# dWb+

$Revision$

# Inhaltsverzeichnis


Anwenderhandbuch dWb+ ..... 1
1. Überblick ..... 1
Datenflussprogrammierung ..... 1
Einleitung ..... 1
Physikalische Größe gegen Botschaften ..... 1
Botschaften ..... 2
Parallelität und Skalierbarkeit ..... 2
Synchronisierte Botschaften ..... 3
Datentypen ..... 3
Generische Module ..... 4
Dynamische Tooltips ..... 5
Skripts für Verbindungen ..... 5
Gliederung von Workspaces ..... 5
Automatisches Erstellen von Verbindungen ..... 6
Fehlermanagement ..... 6
BeanContext Services ..... 7
Anwendungsszenarien ..... 7
Design von Interaktionen zwischen Funktionseinheiten ..... 8
Arbeiten mit der GUI ..... 9
Mandantenfähigkeit ..... 10
Echtzeitanforderungen ..... 11
Analyse und Debugging ..... 12
Verteiltes Rechnen ..... 13
Plugins ..... 13
Skripted Module für noch schnelleres Rapid Prototyping ..... 14
Programmstart ..... 14
System-Properties ..... 14
Umgebungsvariablen ..... 16
Aufbau der Bedienoberfläche ..... 16
Hauptmenü der Anwendung ..... 17
Hauptwerkzeugleiste der Anwendung ..... 19
2. Workspace ..... 20
Überblick ..... 20
Kontextmenü ..... 21
Actions im Editor von Skripts für Verbindungen ..... 32
Hilfslinien ..... 32
Tastaturbedienung ..... 33
3. Dock ..... 35
Überblick ..... 35
Module ..... 35
Aufgabe ..... 35
Bedienung ..... 35
Actions ..... 36
Workspaces ..... 36
Aufgabe ..... 36
Bedienung ..... 36
Actions ..... 36
Globale Variablen ..... 36
Aufgabe ..... 36
Bedienung ..... 37
Favoriten ..... 37

Aufgabe ..... 37
Bedienung ..... 37
Actions ..... 37
Scratch Manager ..... 38
Aufgabe ..... 38
Bedienung ..... 38
Actions ..... 38
Skript-Module ..... 38
Aufgabe ..... 38
Bedienung ..... 39
Actions ..... 39
4. Module ..... 40
Überblick ..... 40
Modulmenü ..... 42
Inputs ..... 48
Doppelklick auf Inputs ..... 48
Module mit variabler Anzahl von Inputs ..... 48
Outputs ..... 48
Menü für Outputs ..... 48
Doppelklick auf Outputs ..... 49
Tooltips für Outputs ..... 49
Automatisches Erstellen von Verbindungen ..... 50
Diagnose ..... 50
Remoting ..... 51
Überblick ..... 51
Funktionsweise ..... 51
Konfiguration ..... 51
Benutzung ..... 52
JMX ..... 53
Überblick ..... 53
Eigenschaften ..... 53
Überblick ..... 53
Benutzung ..... 53
5. Meta-Module ..... 54
Überblick ..... 54
Parametermodul ..... 54
Actions im Quelltexteditor ..... 55
Spaltmodul ..... 57
SVG-Dokument ..... 58
Überblick ..... 58
Instantiierung ..... 59
Parameterdialog ..... 59
Actions ..... 59
Gruppe ..... 60
Instantiierung ..... 60
Parameterdialog ..... 60
Spezifische MetaModule ..... 61
BeanContext Segmentierung ..... 63
Persistenzunterstützung ..... 63
Überblick ..... 63
Instantiierung ..... 64
Parameterdialog ..... 64
Aviator ..... 66
Überblick ..... 66

Instantiierung ..... 66
Parameterdialog ..... 67
Actions ..... 71
Alternative Visualisierung ..... 72
Interaktive Formulare ..... 72
Überblick ..... 72
Der Designer ..... 73
6. Rollen und Rechte ..... 78
Rechte ..... 78
Konfiguration ..... 79
A. Verzeichnis-Layout ..... 80
dWb.accessory.dirs ..... 80
favmod.xml ..... 80
scratchpad.xml ..... 80
macros.template ..... 80
aviator ..... 80
medium ..... 80
small ..... 80
svg ..... 81
lib ..... 81
moduleEmblems ..... 81
modules ..... 81
remoting ..... 81
scripted ..... 81
services ..... 81
stateupdaters ..... 82
workspaces ..... 82
conversionPresets.xml ..... 82
B. Beispieldateien zum Rechtemanagement ..... 84
Policy ..... 84
Login Configuration ..... 89
Anwenderhandbuch Visualisierungs-Client ..... 1
1. Überblick ..... 1
Anwendung ..... 1
Programmstart ..... 1
Weiterführende Literatur ..... 2
Anwenderhandbuch DynamicSVG-Proxy ..... 1
1. Überblick ..... 1
Einsatz ..... 1
Programmstart ..... 1
Servlets ..... 1
XHTMLSnapshotServlet ..... 1
JavaScriptServlet ..... 2
ProxyRotationSpecServlet ..... 2
ProxyOnOffSpecServlet ..... 2
ProxyScaleSpecServlet ..... 2
ProxyTranslationSpecServlet ..... 3
ProxyGraphSpecServlet ..... 3
FSImageProviderServlet ..... 3
ProxyCurrentValuesUpdaterServlet ..... 3
Architektur dWb+ ..... 1
1. Überblick ..... 1
Datenflussprogrammierung ..... 1
Gliederung ..... 1

Programmiermodell ..... 2
Quelltextbearbeitung ..... 2
Änderungen werden nach Neustart wirksam ..... 2
Änderungen werden zur Laufzeit wirksam ..... 2
2. Runtime ..... 3
Workspace ..... 3
Logik ..... 3
Benutzerschnittstelle ..... 4
Modules ..... 4
Beans ..... 4
Benutzerschnittstelle ..... 4
Metamodule ..... 4
Slots ..... 4
Connections ..... 4
Logik ..... 4
Benutzerschnittstelle ..... 4
3. Connections ..... 5
Overview ..... 5
Implementierung ..... 5
Erweiterbarkeit ..... 6
..... 6
4. Tools ..... 9
Docking-Panel ..... 9
Menüs ..... 9
Navigationshilfen ..... 9
Plugins ..... 9
5. De-/Serialisierung ..... 10
Basisformat ..... 10
Alternative Formate ..... 10
6. Verteiltes Rechnen ..... 11
Einordung ..... 11
Remoting ..... 11
Infrastruktur ..... 11
Realisierung ..... 11
7. Class-Loading ..... 13
Einleitung ..... 13
Umsetzung ..... 13
Hierarchische Workspaces ..... 14
Integration neuer Quellen ..... 14
ClassLoader ..... 14
ClassProvider ..... 14
8. Security ..... 15
Überblick ..... 15
Realisierung ..... 15
Spezielle Permissions ..... 16
mw.security.permissions.InsertModulePermission ..... 16
mw.security.permissions.EstablishLinkPermission ..... 16
dwb.prg.security.permissions.InspectParamPermission ..... 17
dwb.prg.security.permissions.ShowParamPanelPermission ..... 17
mw.security.permissions.LoadWorkspacePermission ..... 17
mw.security.permissions.SaveWorkspacePermission ..... 17
mw.security.permissions.RemoveModulePermission ..... 18
mw.security.permissions.RemoveLinkPermission ..... 18
dwb.prg.security.permissions.ToggleLinkActivityPermission ..... 18

dwb.prg.security.permissions.DynamicSVGPermission ..... 18
dwb.prg.security.permissions.AviatorPermission ..... 18
Schaffung neuer spezifischer Permissions ..... 18
Check spezieller Permissions ..... 19
Boolesche Entscheidung ..... 19
PrivilegedAction ..... 20
Pogrammierhandbuch dWb+ ..... 1
1. Einleitung und Überblick ..... 1
dWb+ ..... 1
Module ..... 1
Konfiguration von Modulen ..... 2
Visuelle Entsprechungen in dWb+ ..... 2
Umsetzung in Java ..... 2
Allgemeines ..... 2
Inputs ..... 3
Algorithmus ..... 3
Outputs ..... 3
Konfigurationsparameter ..... 4
Diagnose ..... 4
Zusammenfassung/Schlußfolgerung ..... 5
Entwicklungsschritte ..... 5
2. BeanInfo-Verwendung in dWb+ ..... 6
Internationalisierung ..... 6
Slots ..... 6
Beschriftung ..... 6
Tooltips ..... 6
StateUpdaters ..... 6
Automatisches Erstellen von Verbindungen ..... 6
Maximale Anzahl von Verbindungen ..... 7
Modultitel ..... 7
Modulbeschreibung ..... 7
Icon ..... 7
Layer ..... 7
Verbergen von Properties ..... 8
3. Ein einfaches Beispiel ..... 9
Funktionsbeschreibung ..... 9
Selber machen oder Basisklasse benutzen? ..... 9
Input ..... 9
Konfiguration ..... 10
Output ..... 10
Algorithmus ..... 11
4. Eine JMX MBean als Modul ..... 12
Funktionsbeschreibung ..... 12
Selber machen oder Basisklasse benutzen? ..... 12
MXBean ..... 12
Konfiguration ..... 13
Notification Infrastruktur ..... 13
Output ..... 14
Input ..... 14
5. Arbeiten im Hintergrund (Threads) ..... 16
Warum? ..... 16
Ein einfaches Beispiel ..... 16
Selber machen oder Basisklasse benutzen? ..... 16
Input ..... 18

Konfiguration ..... 18
Output ..... 19
Algorithmus ..... 19
6. Parallele Verarbeitung innerhalb eines Moduls ..... 21
Funktionsbeschreibung ..... 21
Selber machen oder Basisklasse benutzen? ..... 21
BeanContext ..... 22
Workload ..... 23
Input ..... 23
Output ..... 24
7. BeanContext und BeanContextServices konsumieren ..... 25
Warum? ..... 25
Ein einfaches Beispiel ..... 25
Selber machen oder Basisklasse benutzen? ..... 25
Dienst nutzen ..... 26
BeanContext-Mitgliedschaft ..... 27
Modul wird zu Context hinzugefügt ..... 27
Modul wird aus Context entfernt ..... 27
Dienst wird zur Verfügung gestellt ..... 27
Dienst wird zurückgezogen ..... 29
8. BeanContextServices bereitstellen ..... 30
Warum? ..... 30
Ein einfaches Beispiel ..... 30
Selber machen oder Basisklasse benutzen? ..... 30
Dienst definieren ..... 31
BeanContext-Mitgliedschaft ..... 31
Modul wird zu Context hinzugefügt ..... 31
Modul wird aus Context entfernt ..... 31
Dienst wird zur Verfügung gestellt ..... 32
Dienst wird zurückgezogen ..... 32
Dienst bereitstellen ..... 33
Ressourcen freigeben ..... 33
ServiceSelectors definieren ..... 33
9. Generics ..... 35
Warum ..... 35
Ein einfaches Beispiel ..... 36
Selber machen oder Basisklasse benutzen? ..... 36
Input ..... 37
Output ..... 37
Algorithmus ..... 37
BeanInfo ..... 37
10. Variable Anzahl von Inputs ..... 39
Warum ..... 39
Ein einfaches Beispiel ..... 39
Selber machen oder Basisklasse benutzen? ..... 39
Input ..... 40
Output ..... 40
Algorithmus ..... 40
BeanInfo ..... 41
11. Module zur Visualisierung mit einer variablen Anzahl von Inputs ..... 42
Warum ..... 42
Ein einfaches Beispiel ..... 42
Selber machen oder Basisklasse benutzen? ..... 43
Input ..... 43

Hinzufügen neuer Komponenten durch Hinzufügen neuer Input-Slots ..... 43
Schicken neuer Daten an die korrekte Komponente ..... 44
Model ..... 44
BeanInfo ..... 44
Persistenz der Konfiguration für die Komponenten ..... 44
12. Variable Module ..... 46
Warum ..... 46
Ein einfaches Beispiel ..... 46
Selber machen oder Basisklasse benutzen? ..... 46
Input ..... 47
Konfigurationsparameter ..... 47
Output ..... 47
Versenden des PropertyChangeEvents "insAndOuts" ..... 47
Interface VariableBean ..... 48
Ergebnis ..... 48
13. Variable Module für User Eingaben ..... 50
Warum ..... 50
Ein einfaches Beispiel ..... 50
Selber machen oder Basisklasse benutzen? ..... 50
Input ..... 51
Konfigurationsparameter ..... 51
Output ..... 51
Erzeugen der Komponenten zur Konfiguration der Ausgänge ..... 51
Glue ..... 51
14. Module zur Filterung ..... 53
Funktionsbeschreibung ..... 53
Selber machen oder Basisklasse benutzen? ..... 53
Konstruktor ..... 54
15. Eigene Bedienoberflächen für ein Modul ..... 55
Anpassung der Bedienoberfläche für Module ..... 55
Vorarbeit ..... 56
Implementierung ..... 57
16. Actions zur Steuerung eines Moduls ..... 58
Warum ..... 58
Implementierung ..... 58
Actions erzeugen ..... 58
Vorbereitung Anzeige Menü ..... 59
17. Spezielle Properties zur besseren Integration von Modulen ..... 60
Warum ..... 60
18. Module, die mit externen Ressourcen arbeiten müssen ..... 61
Motivation ..... 61
Funktionsbeschreibung ..... 61
Die Basisklasse ..... 61
Input ..... 62
Konfiguration ..... 62
Deklarierte Basisklassenmethoden definieren ..... 63
19. De-/aktivierbare Module ..... 65
Funktionsbeschreibung ..... 65
Selber machen oder Basisklasse benutzen? ..... 65
Output ..... 66
Start des Hintergrundprozesses ..... 66
Workload ..... 66
Erweiterung: Einzelschritt ..... 67
20. Java Api for XML Binding ..... 69

Marshaling ..... 69
Unmarshaling ..... 69
Selber machen oder Basisklasse benutzen? ..... 69
21. Verteiltes Arbeiten (Remoting) ..... 71
Warum? ..... 71
Ein einfaches Beispiel ..... 71
Selber machen oder Basisklasse benutzen? ..... 71
Input ..... 73
Konfiguration ..... 73
Output ..... 73
Algorithmus ..... 73
Deployment ..... 74
22. Getaktete Module ..... 75
Funktionsbeschreibung ..... 75
Selber machen oder Basisklasse benutzen? ..... 75
Input ..... 76
Konfiguration ..... 76
Output ..... 76
Algorithmus ..... 76
23. Mandantenfähigkeit ..... 78
Warum? ..... 78
Selber machen oder Basisklasse benutzen? ..... 78
Daten in den Context einspeisen ..... 78
BeanContextChildModuleBase ..... 78
ThreadingBeanContextChildModuleBase ..... 78
Daten aus dem Context auslesen ..... 79
BeanContextChildModuleBase ..... 79
ThreadingBeanContextChildModuleBase ..... 79
24. Enterprise JavaBeans (EJBs) als Module ..... 80
Funktionsbeschreibung ..... 80
Selber machen oder Basisklasse benutzen? ..... 80
JNDI Lookup ..... 81
Output ..... 82
Input ..... 82
Workload ..... 82
Benutzung als Skripted Module ..... 83
25. Module mit geänderter Kommunikationsmetapher ..... 84
Warum? ..... 84
Ein einfaches Beispiel ..... 84
Selber machen oder Basisklasse benutzen? ..... 84
Input ..... 85
Konfiguration ..... 85
Output ..... 85
Algorithmus ..... 86
26. Packaging ..... 87
27. StateUpdaters ..... 88
Einführung ..... 88
Implementierung ..... 88
Ein einfaches Beispiel ..... 88
Selber machen oder Basisklasse benutzen? ..... 88
Der Konstruktor ..... 89
Das Update ..... 90
Rollout ..... 90
28. Aviator Templates ..... 91

Gestaltung ..... 91
29. Exportieren in eigenen Dateiformaten ..... 93
Warum? ..... 93
Wie? ..... 93
Implementierung ..... 93
Ergebnisse ..... 95
Registrierung/Packaging ..... 95
Weiterführende Informationen ..... 96
30. Graphische Programmierung für beliebige Komponenten ..... 97
Warum? ..... 97
Wie? ..... 97
Formale Spezifikation ..... 97
Metadaten ..... 97
Verbindungsendpunkte ..... 99
VisualComponentSpec ..... 101
Properties ..... 102
Connections ..... 103
Children ..... 103
Technische Spezifikation ..... 104
Simples Beispiel ..... 104
Komplexes Beispiel einschließlich hierarchischer Gliederung ..... 105
Introspector Implementation ..... 111
Metadaten ..... 112
Verbindungsendpunkte ..... 114
Ergebnisse ..... 114
Weiterführende Informationen ..... 115
31. Service Provider Infrastructure ..... 116
Einleitung ..... 116
Module ..... 116
Überblick ..... 116
Interface ..... 116
Beispielimplementierung ..... 116
Verbindungen ..... 117
Überblick ..... 117
Interface ..... 117
Beispielimplementierung ..... 118
32. Modul-Wrapper ..... 120
Einleitung ..... 120
Die Idee ..... 120
Anforderungen an ModuleWidgetWrapper ..... 121
GUI-Integration ..... 121
Link-Management ..... 121
Persistenz ..... 121
Message-Management ..... 121
Die Einbindung ..... 121
Die Basisklasse ..... 122
Abstrakte Methoden, die überschrieben werden müssen ..... 122
Methoden der Basisklasse, die überschrieben werden sollten ..... 123
33. Unter der Haube ..... 127
Kommunikation ..... 127
Grundlegendes ..... 127
Context (Mandantenfähigkeit) ..... 130
Basisklassen ..... 132
ModuleBase ..... 132

ResetableModuleBase ..... 133
BeanContextChildModuleBase ..... 133
CommunicationTemplate ..... 133
VariableNumberofInputsVisualization ..... 133
ThreadingModuleBase ..... 133
ThreadingBeanContextChildModuleBase ..... 134
ThreadingResetableModuleBase ..... 134
ThreadingCommunicationTemplate ..... 134
HostPortCommunicationTemplate ..... 134
StartStopModule ..... 134
StartStopModuleWithDoOnce ..... 135
JaxbBase ..... 135
MapMessageModule ..... 135
RemoteModule ..... 135
A. Quellcode ..... 136
Ein einfaches Modul ..... 136
Eine JMX MBean als Modul ..... 137
Interface ..... 137
Modul ..... 137
Ein Modul mit Algorithmusbearbeitung im eigenen Thread ..... 139
Parallele Verarbeitung innerhalb eines Moduls ..... 140
Benutzung eines Dienstes aus dem BeanContext ..... 142
Bereitstellung eines Dienstes für einen BeanContext ..... 144
Service ..... 144
Implementierung ..... 145
ServiceProvider ..... 145
Generics ..... 147
Variable Anzahl von Inputs ..... 147
Variable Module zur Visualisierung ..... 148
Variable Module ..... 149
Variable Module für User Eingaben ..... 150
Module zur Filterung ..... 152
Implementierung einer alternativen Bedienoberfläche ..... 152
Modul ..... 152
Angepasster JToggleButton ..... 153
Actions für Module ..... 154
SocketOut ..... 155
Start/Stop ..... 157
Start/Stop mit Einzelschritt ..... 158
Enterprise JavaBeans als Module ..... 159
Module mit geänderter Kommunikationsmetapher ..... 160
Verteiltes Arbeiten (Remoting) ..... 161
Modul ..... 161
Interface ..... 162
Implementierung ..... 163
Getaktete Module ..... 163
StateUpdater ..... 164
DemoWorkspaceExporter ..... 165
B. BeanContext Services ..... 167
LoggingConfig ..... 167
Einsatz ..... 167
Methoden ..... 167
DialogParentFrame ..... 167
Einsatz ..... 167

Methoden ..... 167
IDProvider ..... 168
Einsatz ..... 168
Methoden ..... 168
Notification ..... 168
Einsatz ..... 168
Methoden ..... 168
ReportingEngine ..... 169
Einsatz ..... 169
Methoden ..... 169
WorkspaceAPI ..... 170
Einsatz ..... 170
Methoden ..... 170
Bonjour ..... 171
Einsatz ..... 171
Methoden ..... 171
JSMandantManager ..... 171
Einsatz ..... 171
Methoden ..... 172
BackgroundExecutor ..... 172
Einsatz ..... 172
Methoden ..... 172
ApplicationServer ..... 173
Einsatz ..... 173
Methoden ..... 174
Environment ..... 174
Einsatz ..... 174
Methoden ..... 174
BeanContextServices ..... 175
Einsatz ..... 175
..... 175
C. Erstellen von Komponenten für dWb+ mittels Maven ..... 176
Einleitung ..... 176
Module ..... 176
Service Provider Interface ..... 179
OSGI-Bundles ..... 181
D. Fragen und Antworten ..... 183

# Anwenderhandbuch dWb+

**Jürgen Key**

# Anwenderhandbuch dWb+

Jürgen Key

# Inhaltsverzeichnis


1. Überblick ..... 1
    Datenflussprogrammierung ..... 1
        Einleitung ..... 1
        Physikalische Größe gegen Botschaften ..... 1
        Botschaften ..... 2
        Parallelität und Skalierbarkeit ..... 2
        Synchronisierte Botschaften ..... 3
        Datentypen ..... 3
        Generische Module ..... 4
        Dynamische Tooltips ..... 5
        Skripts für Verbindungen ..... 5
        Gliederung von Workspaces ..... 5
        Automatisches Erstellen von Verbindungen ..... 6
        Fehlermanagement ..... 6
        BeanContext Services ..... 7
        Anwendungsszenarien ..... 7
        Design von Interaktionen zwischen Funktionseinheiten ..... 8
        Arbeiten mit der GUI ..... 9
        Mandantenfähigkeit ..... 10
        Echtzeitanforderungen ..... 11
        Analyse und Debugging ..... 12
        Verteiltes Rechnen ..... 13
        Plugins ..... 13
        Skripted Module für noch schnelleres Rapid Prototyping ..... 14
    Programmstart ..... 14
        System-Properties ..... 14
        Umgebungsvariablen ..... 16
    Aufbau der Bedienoberfläche ..... 16
    Hauptmenü der Anwendung ..... 17
    Hauptwerkzeugleiste der Anwendung ..... 19
2. Workspace ..... 20
    Überblick ..... 20
    Kontextmenü ..... 21
    Actions im Editor von Skripts für Verbindungen ..... 32
    Hilfslinien ..... 32
    Tastaturbedienung ..... 33
3. Dock ..... 35
    Überblick ..... 35
    Module ..... 35
        Aufgabe ..... 35
        Bedienung ..... 35
        Actions ..... 36
    Workspaces ..... 36
        Aufgabe ..... 36
        Bedienung ..... 36
        Actions ..... 36
    Globale Variablen ..... 36
        Aufgabe ..... 36
        Bedienung ..... 37
    Favoriten ..... 37
        Aufgabe ..... 37

Bedienung ..... 37
Actions ..... 37
Scratch Manager ..... 38
Aufgabe ..... 38
Bedienung ..... 38
Actions ..... 38
Skript-Module ..... 38
Aufgabe ..... 38
Bedienung ..... 39
Actions ..... 39
4. Module ..... 40
Überblick ..... 40
Modulmenü ..... 42
Inputs ..... 48
Doppelklick auf Inputs ..... 48
Module mit variabler Anzahl von Inputs ..... 48
Outputs ..... 48
Menü für Outputs ..... 48
Doppelklick auf Outputs ..... 49
Tooltips für Outputs ..... 49
Automatisches Erstellen von Verbindungen ..... 50
Diagnose ..... 50
Remoting ..... 51
Überblick ..... 51
Funktionsweise ..... 51
Konfiguration ..... 51
Benutzung ..... 52
JMX ..... 53
Überblick ..... 53
Eigenschaften ..... 53
Überblick ..... 53
Benutzung ..... 53
5. Meta-Module ..... 54
Überblick ..... 54
Parametermodul ..... 54
Actions im Quelltexteditor ..... 55
Spaltmodul ..... 57
SVG-Dokument ..... 58
Überblick ..... 58
Instantiierung ..... 59
Parameterdialog ..... 59
Actions ..... 59
Gruppe ..... 60
Instantiierung ..... 60
Parameterdialog ..... 60
Spezifische MetaModule ..... 61
BeanContext Segmentierung ..... 63
Persistenzunterstützung ..... 63
Überblick ..... 63
Instantiierung ..... 64
Parameterdialog ..... 64
Aviator ..... 66
Überblick ..... 66
Instantiierung ..... 66

Parameterdialog ..... 67
Actions ..... 71
Alternative Visualisierung ..... 72
Interaktive Formulare ..... 72
Überblick ..... 72
Der Designer ..... 73
6. Rollen und Rechte ..... 78
Rechte ..... 78
Konfiguration ..... 79
A. Verzeichnis-Layout ..... 80
dWb.accessory.dirs ..... 80
favmod.xml ..... 80
scratchpad.xml ..... 80
macros.template ..... 80
aviator ..... 80
medium ..... 80
small ..... 80
svg ..... 81
lib ..... 81
moduleEmblems ..... 81
modules ..... 81
remoting ..... 81
scripted ..... 81
services ..... 81
stateupdaters ..... 82
workspaces ..... 82
conversionPresets.xml ..... 82
B. Beispieldateien zum Rechtemanagement ..... 84
Policy ..... 84
Login Configuration ..... 89

# Abbildungsverzeichnis

1.1. Bereiche des Hauptfensters ..... 17
2.1. Dialog zur Verwaltung von Plugins ..... 30
2.2. Dialog zur Bearbeitung des Skripts an einer Verbindung ..... 31
4.1. Aufbau von Modulen ..... 41
4.2. Beispiel für ein Modul mit variabler Anzahl von Eingängen ..... 48
5.1. Dialog zur Eingabe von Code-Fragmenten ..... 54
5.2. Beispiel für ein Parametermodul ..... 55
5.3. Dialog zur Bearbeitung von Code-Templates ..... 57
5.4. Beispiel für ein Spaltmodul ..... 58
5.5. Beispiel für einen nicht hierarchisch aufgebauten Workspace ..... 61
5.6. Beispiel für einen hierarchisch aufgebauten Workspace ..... 63
5.7. Parameterdialog der Persitenzunterstützung mit Beispielworkspace ..... 65
5.8. Parameterdialog Aviator mit zwei Instrumenten ..... 67
5.9. Ausschnitt aus der Palette zur Auswahl der Instrumente ..... 68
5.10. Konfiguration der Instrumente ..... 69
5.11. Formulardesigner ..... 72
5.12. Palette ..... 74
5.13. ..... 74
5.14. Die Properties ..... 75
5.15. Die Events ..... 76
5.16. Die Formulare ..... 76

# Tabellenverzeichnis

1.1. Menü Projekt ..... 17
1.2. Werkzeuge ..... 18
1.3. Menü Steuerung ..... 18
1.4. Menü Information (?) ..... 18
1.5. Actions in der Hauptwerkzeugleiste ..... 19
2.1. Kontextmenü im Workspace ..... 21
2.2. Untermenü zum Ändern der selektierten Elemente ..... 22
2.3. Untermenü zum Umschalten des Modus ..... 22
2.4. Untermenü für Hilfslinien ..... 23
2.5. Untermenü Verbindungen ..... 23
2.6. Untermenü Farbverwaltung ..... 24
2.7. Untermenü Hervorheben ..... 27
2.8. Untermenü Meta-Module ..... 28
2.9. Untermenü Navigation ..... 29
2.10. Untermenü für Ebenen ..... 29
2.11. Aktionen im Dialog zur Verwaltung der Ebenen ..... 29
2.12. Untermenü Plugins ..... 30
2.13. Status von Plugins ..... 30
2.14. Aktionen zur Verwaltung von Plugins ..... 30
2.15. Actions zur Verwaltung von Remoting Servern ..... 31
2.16. Actions im Editor zum Bearbeiten von Skripts für Verbindungen ..... 32
3.1. Actions im Modulbaum ..... 36
3.2. Actions in der Workspace-Liste ..... 36
3.3. Actions für die Verwaltung der Favoriten ..... 37
3.4. Actions im Scratch-Manager ..... 38
3.5. Actions für Skript-Module ..... 39
4.1. Kontextmenü für Module ..... 42
4.2. Untermenü Ausrichtung ..... 42
4.3. Untermenü Sichtbarkeit ..... 43
4.4. Untermenü Refactoring ..... 43
4.5. Untermenü Parameterdialoge ..... 44
4.6. Untermenü Diagnose ..... 45
4.7. Untermenü Reporting (Diagnose) ..... 45
4.8. Untermenü JMX ..... 46
4.9. Untermenü Modul-Eigenschaften ..... 47
4.10. Steuerung des Threads eines Moduls ..... 47
4.11. Pufferstrategie für eingehende Daten ..... 47
4.12. Kontextmenü für Output-Slots ..... 49
4.13. Kennzeichnungen für Module, die automatisch Verbindungen erstellen können ..... 50
5.1. Actions im Editor zum Bearbeiten von Quelltextfragmenten ..... 55
5.2. Actions für das Meta-Modul SVG-Dokument ..... 59
5.3. Werkzeugleiste für das Meta-Modul Persistenzunterstützung ..... 65
5.4. Actions für die Verwaltung dynamischer Spalten ..... 66
5.5. Actions im Parameterdialog der Aviatormodule ..... 67
5.6. Actions zur Konfiguration der Instrumente ..... 68
5.7. Actions zur Bestimmung der Reihenfolge im Stapel ..... 70
5.8. Kontextmenü fürs Layout der Instrumente im Aviator ..... 70
5.9. Ausrichtung ..... 70
5.10. Actions für das Meta-Modul Aviator ..... 71
5.11. Actions in der Werkzeugleiste des Meta-Moduls Dynamische Formulare ..... 73
5.12. Actions in der Werkzeugleiste des Objektbaumes im Metamodul Dynamische Formulare..... 74

5.13. Actions im Kontextmenü der Formulare im Metamodul Dynamische Formulare ..... 76
5.14. Actions zur relativen Anordnung von Formularelementen in Formularen im Metamodul Dynamische Formulare ..... 77

# Kapitel 1. Überblick

## Datenflussprogrammierung

### Einleitung

Datenflussprogrammierung ist eine Technologie, die nunmehr bereits seit 30 bis 40 Jahren angewendet wird, um Probleme in der Informationsverarbeitung zu lösen. Abstrakt gesprochen handelt es sich bei der Datenflussprogrammierung um die Definition und Konfiguration von Datenquellen und Datensenken und Datentransformationen und um die Definition der Pfade, über die Informationen zwischen diesen Akteuren transportiert werden.

Es existieren sowohl traditionelle Programmiersprachen, bei denen diese Definitionen und Konfigurationen in einer Textdatei hinterlegt wird, wie auch graphische Systeme, in denen die Senken, Quellen und Transformationen graphisch repräsentiert werden und die Datenpfade diese graphischen Repräsentationen verbinden, dadurch entsteht ein Netzwerk oder Graph, das visuell die Abläufe repräsentiert - man kann den Daten sozusagen bei ihrem Weg durch die Verarbeitungseinheiten zusehen.

dWb+ ist ein solches graphisches Framework zur Datenflussprogrammierung. Es geht aber noch über die bisher beschriebenen Leistungen hinaus: Außerdem ist es ein Framework, um schnell und einfach neue Transformationen, Quellen und Senken programmieren zu können. Dazu können außer der Haupt-Programmiersprache Java auch Skriptsprachen wie etwa BeanShell (ein interpretierter Java-Dialekt) oder Groovy benutzt werden. Die Menge von Sprachen, mit denen neue Module implementiert werden können, ist erweiterbar. Weiterhin kann man dWb+ sehr einfach dazu benutzen, ganz generell Datenflüsse zu modellieren. Dadurch wird es für Fälle interessant, in denen Sprachen ohne eigene visuelle Designwerkzeuge eingesetzt werden sollen. In diesem Fall kann dWb+ die Modellierung übernehmen und die eigentliche Zielsprache dann die Ausführung.

dWb+ bleibt den prinzipiellen Vorteilen von Systemen zur Datenflussprogrammierung treu: eine inherente Parallelisierbarkeit ist ebenso gegeben, wie die intuitive Bedienbarkeit, die auch Nicht-Programmierern auf einfache Art und Weise gestattet, komplexe Datenflüsse zu konfigurieren, um bestimmte Ziele zu erreichen.

Die folgenden Abschnitte sollen einige der grundlegenden Ideen und Architekturentscheidungen bei der Entwicklung von dWb+ näher erläutern, um Anwendern ohne zu sehr in die Tiefe der Architektur des Systems einzusteigen, ein grundlegendes Verständnis für die Technologie zu vermitteln.

## Physikalische Größe gegen Botschaften

Verbindungen, die visuell als Linie zwischen zwei Modulen dargestellt werden, transportieren Informationen. Diese Informationen kann man zum einen als Botschaften sehen: Es existiert ein diskreter Zeitpunkt, zu dem sie jeweils versendet werden - sind sie abgesendet, sind sie auf der Seite des Senders sozusagen vergessen und wenn danach ein Eingang eines weiteren Moduls mit dem betreffenden Ausgang verbunden wird, wird dieses neu verbundene Modul nicht über die letzte versendete Botschaft informiert. Man kann sich das wie das Abonnement einer Zeitung vorstellen: Am Montagmorgen liefert der Zeitungsbote die Zeitungen an alle Abonnenten aus. Entscheidet sich nun jemand, am Montag die Zeitung zu abonnieren, so erhält er sein erstes Exemplar erst am Dienstag - erst, wenn eine neue Botschaft vom Sender (dem Zeitungsverlag) versendet wird.

Eine zweite Möglichkeiten, diese Informationen zu interpretieren ist, sie als Zustände physikalischer Größen - etwa Druck, Temperatur oder elektrische Spannung - zu sehen. In diesem Fall - speziell, wenn

wir die Analogie mit der elektrischen Spannung benutzen - ist klar, dass, sobald eine neue Verbindung etabliert wird, das Empfängermodul über den aktuellen Status des Senders informiert wird, auch wenn dieser Status sich gar nicht geändert hat.

Für beide Sichtweisen lassen sich gute Argumente finden - bei der Entwicklung von dWb+ musste eine Entscheidung getroffen werden: in dWb+ haben wir uns für die Realisierung der Signale als Botschaften entschieden: Nur die Änderung einer Ausgangsgröße löst die Benachrichtigung der angeschlossenen Empfänger aus. Empfänger an neu hergestellten Verbindungen müssen warten, bis sich am Sender der Zustand ändert, um eine Botschaft zu empfangen.

# Botschaften

## Wiederholte gleiche Werte

Wie bereits im vorhergehenden Abschnitt gesagt, sei hier nochmals betont, dass Botschaften, die von einem sendenden Modul ausgehen und exakt dieselben Information enthalten, wie die unmittelbar vorhergehende Botschaft desselben Moduls nicht an die angeschlossenen Empfänger übertragen werden. Dieses Verhalten kann nur über spezielle Maßnahmen überschrieben werden: das Sender-Modul kann dem Framework über spezielle Gestaltung der Botschaft signalisieren, dass dieser Test unterbleiben soll. Der Gedanke hinter der Wahl dieses Mechanismus ist, dass das Gesamtsystem deterministisch ist, was letztlich bedeutet, dass Operationen mit identischen Inputs immer dasselbe Resultat liefern und daher nicht erneut durchgeführt werden müssen.

## Kopie vs. Referenz

Das Framework unterstützt nicht beim Kopieren der Botschaftsinhalte: die Botschaften enthalten lediglich Referenzen auf Datenobjekte. Da am Empfänger nicht klar ist, ob die empfangene Referenz nicht vielleicht doch im Sender weiterbearbeitet wird, sollte der Empfänger sicherheitshalber eine Kopie der empfangenen Daten anfertigen.

# Parallelität und Skalierbarkeit

Bereits weiter oben wurde angedeutet, dass Datenflussprogrammierung in der reinen Lehre bedeutet, dass damit realisierte Lösungen intrinsich parallelisierbar sind und skalieren. Bei dWb+ muss man wegen der verwendeten Realisierung folgendes dazu anmerken: Im Prinzip ist diese reine Lehre mit dWb+ umsetzbar, wenn man die einzelnen Module so erstelllt, dass sie in einem jeweils eigenen Thread arbeiten. Die einzelnen Threads werden vom System dann auf die zur Verfügung stehenden Verarbeitungseinheiten (Prozessoren) verteilt.

Diese Realisierung in eigenen Threads ist für einen geschulten Programmierer zwar keine große Sache, allerdings stellt das Framework, das Teil von dWb+ ist, eine Basislklasse zur Verfügung, die man benutzen kann, um sich das Leben bei der Erstelllung neuer Module so weit es geht zu erleichtern.

Interessant beim Thema Parallelisierung ist, dass es zu Problemen kommen kann, wenn die Module weiter hinten in der Verarbeitungskette langsamer arbeiten, als die, die ihnen Informationen zuarbeiten: ohne Parallelität würden die schnelleren einfach von den langsamen ausgebremst - damit kann ein solches Problem nicht entstehen. Dadurch, dass in einem parallelen System die schnellen Module die Ergebnisse ihrer Arbeit in eine Botschaft packen und diese versenden, können sie sofort weiterarbeiten und die nächste Botschaft versenden. Bildlich gesprochen sammelt sich ein riesiger Haufen dieser Botschaften im Eingang der langsamen Module und ohne Gegenmaßnahmen ist dadurch irgendwann der Hauptspeicher ausgefüllt und das System kommt zum Stillstand. Gegen dieses potentielle Problem muss man Gegenmaßnahmen einleiten.

dWb+ adressiert dieses Szenario, indem für jeden Eingang eines Moduls ein Puffer mit einer konfigurierbaren Strategie eingerichtet wird: Dadurch ist es zum Beispiel möglich, dass die Bearbeitung einer eingehenden Botschaft A die Bearbeitung startet. Trifft eine Botschaft B ein, wird diese im Puffer vermerkt, falls die Bearbeitung von A noch nicht abgeschlossen wurde. Trifft eine weitere Botschaft C ein, während immer noch A verarbeitet wird, wird B verworfen und deren Platz im Puffer wird durch C eingenommen. Sobald nun die Bearbeitung von A abgeschlossen wurde, sieht das System, dass sich eine weitere Botschaft im Puffer befindet und fährt mit deren Abarbeitung fort. Die beschriebene Strategie ist nur eine von mehreren zur Verfügung stehenden, zwischen denen der Anwender bei der Konfiguration der Module wählen kann.

Näheres dazu und einige Praxisbeispiele sind zum Beispiel im Pogrammierhandbuch dWb+ im Kapitel 5, *Arbeiten im Hintergrund (Threads)* zu finden.

# Synchronisierte Botschaften

Ein Abarbeitungsmodell, wie es die Datenflussprogrammierung darstellt, behandelt Botschaftn als für sich stehend und nicht an andere Botschaften gekoppelt. Insbesondere existiert per se in dWb+ keine Möglichkeit, Botschaften zu synchronisieren. Es ist also nicht so, dass man ein Modul so konfigurieren kann, dass es erst dann an seinem Ausgang Ergebnisbotschaften erzeugt, wenn an spezifischen Eingängen seit der Instantiierung des Moduls oder seit der Erzeugung der letzten Ergebnisbotschaft jeweils mindestens eine Botschaft eingegangen ist.

Ist ein solches Verhalten gewünscht, muss der Ersteller eines solchen Modules dieses implementieren und natürlich auch dokumentieren.

Die einzige Unterstützung, die zur Zeit (entfernt) für die Synchronisierung durch dWb+ zur Verfügung gestellt wird, ist eine Methode, die ein wenig an digitale Schaltungen erinnert: Botschaften gehen an solchen Modulen ganz normal an den Eingängen ein, werden aber erst dann zur Verarbeitung benutzt, wenn an einem bestimmten, speziell gekennzeichneten sogenannten Takteingang ebenfalls eine Botschaft eingeht.

Näheres dazu und einige Praxisbeispiele sind zum Beispiel im Pogrammierhandbuch dWb+ im Kapitel 22, *Getaktete Module* zu finden.

Im Speziellen ist es in dWb+ nicht so, dass die Botschaften selbst Zeitstempel tragen. Der dafür notwendige Overhead wäre gegenüber dem Nutzen nicht vertretbar. Es steht Entwicklern neuer Module für das System natürlich frei, Botschaften zu definieren, die solche Zeitstempel in sich tragen.

# Datentypen

## Botschaften

Botschaften - das heißt Daten, die zwischen Modulen ausgetauscht werden - haben immer einen Datentyp. Ausgänge und Eingänge von Modulen haben ebenfalls einen festgelegten Datentyp. Bei Ausgängen zeigt dieser Typ an, was für Botschaften das Modul am jeweiligen Ausgang produziert, an Eingängen zeigt er an, welcherart Botschaften das jeweilige Modul verarbeiten kann.

## Matching

Das System dWb+ erlaubt die Verbindung zwischen Aus- und Eingängen nur dann, wenn der Ausgang Botschaften eines Typs erzeugt, die der jeweilige Eingang verarbeiten kann. Dabei müssen die Typen von Aus- und Eingang nicht exakt aufeinander passen - das lässt sich vielleicht am besten an einem Beispiel erläutern: Ein Ausgang, der ganze Zahlen erzeugt, kann mit einem Eingang verbunden werden, der Zahlen entgegennimmt, da auch ganze Zahlen Zahlen sind. Umgekehrt ist es aber nicht möglich, einen

Ausgang, der Zahlen erzeugt mit einem Eingang zu verbinden, der nur ganze Zahlen erwartet. Denn die Ausgangsspezifikation allgemeiner Zahlen könnte auch gebrochene Zahlen einschließen, mit denen der Eingang aber nicht umgehen könnte.

Seit Version `4.3pre1` ist es möglich, Ein- und Ausgänge, deren Typen eigentlich nicht zueinander passen dennoch zu verbinden: Hält man während des Drag'n'Drop die Taste **Strg** gedrückt, ist die im vorhergehenden Absatz beschriebene Validierung zueinander passender Datentypen von Ein- und Ausgängen außer Kraft gesetzt: Wenn nach Loslassen der Maustaste festgestellt wird, dass die beiden verbundenen Slots nicht zueinander passen, wird zwischen beiden automatisch ein Konvertermodul in die neue Verbindung eingefügt. Es öffnet sich sofort das Parameterfenster dieses Moduls, in dem der Anwender mittels BeanShell die Konversion des Datenobjekts aus dem Ausgang in ein für den Eingang passendes Objekt spezifizieren kann. Dabei gelten die folgenden Konventionen: In dem BeanShell-Fragment sind zwei Variablen mit besonderen Bedeutungen reserviert: `_input_` hält den vom Quellmodul über den Ausgang versendeten Wert und der am Ende der Ausführung des BeanShell-Fragments in `_output_` gespeicherte Wert wird an den Eingang des Zielmoduls gesendet. Der `_input_` wird ungefiltert in das BeanShell-Fragment übergeben - daher ist es angeraten, `null` explizit zu behandeln! Um die Arbeit einfacher zu gestalten ist es möglich, für Paare von Datentypen Default-Konversionen zu hinterlegen, die dann automatisch im Editor des Parameterfensters als Vorbelegung auftauchen. Das ist in „conversionPresets.xml“ in Anhang A, *Verzeichnis-Layout* ausführlicher beschrieben.

## Arrays

Arrays sind in dieser Beziehung spezielle Datentypen: Überall da, wo ein bestimmter Datentyp erlaubt ist, ist auch ein Array dieses Datentyps erlaubt. Das wird dadurch erreicht, dass während der Übertragung der Daten unter diesem Aspekt in die Botschaften hineingeschaut wird: Versendet der Ausgang, aus dem die Botschaft stammt, ein Array und der Eingang, an den die Botschaft gehen soll erwartet aber skalare Daten, so nimmt die Verbindung die Botschaft auseinander und sendet die einzelnen Skalare an den Eingang. Umgekehrt ist es so, dass wenn der Ausgang Skalare sendet, der Eingang aber ein Array erwartet, die Verbindung jedes Skalar in ein Array der Dimension und Länge eins verwandelt und so an den Eingang weiterreicht.

# Generische Module

Im System dWb+ existieren Module, deren Ein- und Ausgangstypen nicht bereits bei der Erstellung festgelegt sein müssen. Auch hier illustriert das vielleicht ein Beispiel am besten: Man kann sich ein Modul vorstellen, das pro Minute nur eine festgelegte Anzahl an Eingangsbotschaften zu seinem Ausgang durchleiten soll. Diese maximale Anzahl ist konfigurierbar.

Dieses Modul routet die Botschaften nur und verändert sie nicht - es erzeugt auch keine neuen Botschaften. Man kann sich nun vorstellen, dass eine solche Funktionalität für Botschaften, die Zahlen darstellen ebenso nützlich ist, wie für solche, die Texte oder komplexe Datenstrukturen darstellen. Nach dem im letzten Abschnitt Gesagten müsste man nun für jeden Datentyp ein solches Modul erstellen - ein langwieriger, langweiliger und fehlerträchtiger Prozess.

Aus diesem Grund existieren in dWb+ die sogenannen generischen Module: Diese haben die Eigenart, dass ihre Eingänge erst dann einen Typ aufweisen, wenn der erste Ausgang mit ihnen verbunden wurde. Man kann diese Module so gestalten, dass sich zu diesem Zeitpunkt auch der Typ eines oder mehrerer Ausgänge entsprechend ändert. Damit lassen sich Module mit der beschriebenen oder ähnlichen Funktionalitäten ganz einfach umsetzen - und die funktionieren selbst mit Datentypen, die im Moment der Erstellung des Moduls noch nicht existierten!

Näheres dazu und einige Praxisbeispiele sind zum Beispiel im Pogrammierhandbuch dWb+ im Kapitel 9, *Generics* zu finden.

# Dynamische Tooltips

Ein großer Anteil verfügbarer Module lässt sich von der Aufgabe her als Module zur Visualisierung einordnen: Für Fundamentaldatentypen wie auch für komplexere Datenstrukturen existieren bereits Module, die diese Daten visualisieren oder Programmierer können auf einfache Art und Weise neue erstellen.

Um die Anzahl von Modulen nicht zu sehr ansteigen zu lassen, wurden mit Blick auf die Visualisierung der Daten zwei Gegenmaßnahmen implementiert: Es existiert ein Framework innerhalb dWb+, das es sehr einfach macht, Visualisierungsmodule zu schaffen, die zur Laufzeit um weitere Elemente erweitert werden können: Beispielsweise instantiiert man ein Modul, das den historischen Verlauf einer numerischen Größe zeigen kann. Dort eistiert ein Knopf, der diese Visualisierung innerhalb des Moduls vervielfältigt: Das Modul kann dadurch zwei, drei und eigentlich beliebig viele numerische Größen in ebenso vielen voneinander unabhängigen Diagrammen darstellen.

Näheres dazu und einige Praxisbeispiele sind zum Beispiel im im Pogrammierhandbuch dWb+ im Kapitel 10, *Variable Anzahl von Inputs* zu finden.

Die zweite Möglichkeit, die dWb+ bietet, die Anzahl der Module trotz aussagekräftiger Visualisierungen nicht zu sehr ansteigen zu lassen, sind sogenannte dynamische Tooltips: Existiert ein entsprechender dynamischer Tooltip für einen Ausgang eines bestimmten Datentyps, muss zur Visualisierung nicht extra ein Visualisierungsmodul instantiiert und verbunden werden. Es reicht aus, die Maus über den Ausgang zu halten - nach einer gewissen Zeit erscheint der dynamische Tooltip, der auch "festgepinnt" werden kann - damit bleibt die Visualisierung auch dann noch sichtbar, wenn die Maus wegbewegt wird.

Näheres dazu und einige Praxisbeispiele sind zum Beispiel im Anwenderhandbuch dWb+ im Abschnitt „Tooltips für Outputs“auf Seite 49 und im Pogrammierhandbuch dWb+ im Kapitel 27, *StateUpdaters* zu finden.

# Skripts für Verbindungen

Verbindungen in dWb+ sind durch die Verbindungsendpunkte definiert: Der Ausgang des Sendermoduls und der Eingang des Empfangsmoduls beschreiben die Verbindung vollständig. Allerdings existiert eine weitere optionale Eigenschaft für Verbindungen in dWb+: Als weiteren Schritt zur Reduktion der Komplexität von Workspaces - einige wurden bereits im vorhergehenden Abschnitt vorgestellt - existiert die Möglichket, einfache Datentransformationen nicht als Module abbilden zu müssen: Ein Beispiel für so eine einfache Datentransformation wäre das - zugegebenermaßen eher akademische - Szenario, dass das empfangende Modul das Doppelte des numerischen Ergebniswertes des Senders als Eingang benötigt. Statt nun ein Modul schaffen und instantiieren zu müssen, das einen numerischen Eingangswert mit einem konfigurierbaren Faktor multipliziert an andere Module weitergibt, existiert in dWb+ die Möglichkeit, für solche Fälle ein Skript zu definieren. Dieses erhält als Eingangswert das Resultat des Moduls und gibt das Ergebnis an das nächste Modul weiter.

Näheres dazu ist im Anwenderhandbuch dWb+ im Abschnitt „Kontextmenü“ [25] auf Seite 25 zu finden.

# Gliederung von Workspaces

In den vorangegangenen Abschnitten wurden mehrere Wege aufgezeigt, wie man die Komplexität von Workspaces innerhalb von dWb+ reduzieren kann. Wird ein Workspace dennoch zu unübersichtlich, existieren zwei Wege, diese zu erhalten oder wieder herzustellen.

Es existiert die Möglichkeit, Workspaces über sogenannte Gruppenmodule hierarchisch zu gliedern: Die innerhalb der Gruppenmodule platzierten Module können über spezielle Module Daten aus dem

umgebenden Workspace empfangen und auch an diesen senden. Dies ist auch nachträglich möglich: Indem man eine Menge von Modulen selektiert und im Kontextmenü Refactoring (Gruppe) auswählt, werden diese Module in ein automatisch neu erstelltes Gruppenmodul ausgelagert und die benötigten Verbindungen zwischen der Gruppe und den Modulen des umgebenden Workspaces automatisch erstellt.

Näheres dazu ist im Anwenderhandbuch dWb+ im Abschnitt „Gruppe“ auf Seite 60 zu finden.

Eine weitere Möglichkeit ist die Benutzung des Konzeptes der Schichten oder Layers: Es besteht die Möglichkeit, Module in solchen Layers zusammenzufassen. Diese Layers können dann wie aus Graphikprogrammen bekannt, einzeln sichtbar bzw. unsichtbar geschaltet werden.

Näheres dazu ist im Anwenderhandbuch dWb+ im Abschnitt „Überblick“ auf Seite 20 zu finden.

# Automatisches Erstellen von Verbindungen

Um die Erstellung komplexer Workspaces zu vereinfachen und einen möglichst reibungslosen Arbeitsablauf zu gewährleisten, existieren verschiedene Möglichkeiten, Verbindungen schnell zu erstellen: Normalerweise werden Verbindungen zwischen zwei Modulen erstellt, indem der Anwender die linke Maustaste drückt und hält, während der Mauszeiger über einem zu verbindenden Ausgang schwebt. Danach wird der Mauszeiger bei gedrückter Maustaste gezogen, bis er sich über dem intendierten Eingang befindet. Wird die Maustaste nunmehr losgelassen (und sind die beiden Endpunkte vom Typ her kompatibel), wird die Verbindung erstellt. Dieses Vorgehen ist bei größeren Workspaces mit vielen Modulen zeitaufwändig und kompliziert - speziell dann wen Sender und Empfänger so platziert sind, dass nicht beide gleichzeitig im sichtbaren Ausschnitt des Workspace angezeigt werden.

Daher wurden Möglichkeiten geschaffen, Verbindungen schneller zu erstellen: Wird für einen Ausgang das Kontextmenü geöffnet, werden dort alle Module angezeigt, die einen oder mehrere passende Eingänge aufweisen. Man wählt dort einfach das Modul aus, mit dem die neue Verbindung erstellt werden soll (beziehungsweise dessen Eingang, falls es mehrere passende gibt) - und schon ist die Verbindung erstellt.

Man kann in diesem Kontextmenü - gruppiert nach Datentypen - nach einem Modul suchen, das noch gar nicht auf dem Workspace vorhanden ist. Die Auswahl erfolgt wie eben beschrieben, allerdings wird das Modul auch noch automatisch instantiiert und platziert, bevor dann die Verbindung erstellt wird.

Eine weitere Möglichkeit zur schnelleren Erstellung von Verbindungen ist die Methode, zwei Module einfach mit ihren Aus- bzw. Eingangsseiten "zusammenzustoßen". Das sorgt dafür, dass sich passende Eingänge und Ausgänge der beiden Module automatisch verbinden - soll heißen, daß entsprechende Verbindungen automatisch erstellt werden.

Näheres dazu ist im Anwenderhandbuch dWb+ im Abschnitt „Automatisches Erstellen von Verbindungen“ auf Seite 50 zu finden.

# Fehlermanagement

## Logging

Fehler, die in den einzelnen Modulen auftreten, müssen nachverfolgbar sein, damit nicht vorgesehenes oder fehlerhaftes Verhalten des Gesamtsystems analysiert und korrigiert werden kann. dWb+ nutzt zur Protokollierung die Bibliothek log4j. Der Anwender kann das Verhalten in großem Umfang flexibel anpassen. Dazu kann er ein Beispiel für eine Konfigurationsdatei aus der Anwendung heraus in seinem Homeverzeichnis erzeugen und diese dann an seine speziellen Anforderungen anpassen.

Diese Anpassungen können zum Beispiel neue Appender umfassen oder die Konfiguration des Routings zwischen Loggern und Appendern abhängig von den verschiedenen zur Verfügung stehenden Log-Leveln.

Wichtig ist dabei, dass Änderungen in der generierten Beispieldatei keine Auswirkungen haben - die Beispieldatei muss dazu unter einem anderen Namen in demselben Verzeichnis abgelegt werden.

Näheres dazu ist im Anwenderhandbuch dWb+ im Menu Tabelle 1.2, „Werkzeuge“ zu finden.

## Anwenderbenachrichtigung

Für Meldungen über fehlerhafte Zustände innerhalb einer Komponente, die der unmittelbaren Aufmerksamkeit des Anwenders bedürfen, existiert eine Möglichkeit, entsprechende visuelle Hinweise auf dem jeweiligen Modul in Form von Icons anzuzeigen. Programmierer, die diesen Dienst in Anspruch nehmen wollen, können die tun, indem sie eigene Module von den im Framework mitgelieferten Klassen ableiten und die in den Basisklassen vorhandenen Funktionalitäten nutzen oder - falls eine so enge Kopplung an dWb+ nicht gewünscht ist - den entsprechenden BeanContextService benutzen.

Näheres zum Thema Notification Management ist im Anwenderhandbuch dWb+ im Abschnitt „Diagnose“ auf Seite 50 zu finden.

Näheres zum Thema BeanContextServices ist im Pogrammierhandbuch dWb+ im Kapitel 7, *BeanContext und BeanContextServices konsumieren* beziehungsweise im Anhang B, *BeanContext Services* zu finden.

# BeanContext Services

Module können als Stand-alone oder self-sufficient Komponenten gestaltet werden. Manchmal ist es aber so, dass Komponenten auf einen externen Dienst zugreifen möchten oder zentrale Teile mehrerer Module sich gleichen und daher in eine eigene Service-Klasse ausgelagert werden können. Ist dieser extene Dienst über eine eindeutige Schnittstelle beschreibbar, bietet dWb+ die Möglichkeit, die Implementierung des Dienstes zur Laufzeit austauschen zu können.

Interessant dabei ist zu wissen, dass Gruppen-Workspaces hier erlauben, mehrere Implementierungen ein und desselben Interfaces zu nutzen: Innerhalb eines Workspaces - der gleichzeitig der BeanContext ist - kann immer nur eine Implementierung für jedes Service-Interface existieren. Gruppen-Workspaces sind in dieser Beziehung voneinander unabhängig: Sie stellen zwar jeder einen BeanContext dar, aber eben jeder eine eigene Instanz. So kann man für dasselbe Service-Interface in unterschiedlichen Gruppen-Workspaces unterschiedliche Implementierungen nutzen. Ein Beispiel dafür wären Servies, die eine Datenbank-Verbindung über den BeanContext zur Verfügung stellen: Möchte man zeitgleich mit mehreren Datenbanken arbeiten, kann man einfach mehrere Gruppen-Workspaces erstellen und in jedem eine andere Implementierung des Service-Interface zum Zugriff auf die Verbindung registrieren.

Näheres zu diesem Thema ist im Pogrammierhandbuch dWb+ im Kapitel 7, *BeanContext und BeanContextServices konsumieren* beziehungsweise im Anhang B, *BeanContext Services* zu finden.

# Anwendungsszenarien

## Gewöhnliche Differentialgleichungssysteme

Man könnte auf die Idee kommen, dass Datenflussprogrammierung eine naheliegende Technologie für die Modellierung von Differentialgleichungssystemen ist. Das trifft nur unter folgender Bedingung zu: Die Differentialgleichungen dürfen nur mit numerischen Methoden bearbeitet werden, die lediglich eine Auswertung der Funktion f=y'(t) erfordern. Mehrschrittverfahren wie die Klasse der Runke-Kutta-Verfahren oder die Familie der Adams-Verfahren dürfen nicht zur Lösung eingesetzt werden.

Dies daher, weil gekoppelte Differentialgleichungssysteme, die über die aktuellen Zustände miteinander in Verbindung stehen erst nach einem Integrationsschritt über die neuen Zustände informiert werden,

während in einem geschlossenen System diese Informationen innerhalb des Integrationsschrittes zur Verfügung stehen. Da mehrstufige Verfahren die Ableitungen mehrfach und zu unterschiedlichen Zeitpunkten auswerten, fehlen diese Informationen dann, beziehungsweise wird fälschlicherweise unter der Annahme operiert, dass die Inputs von den anderen Differentialgleichungen über den betrachteten Zeitraum des Integrationsschrittes konstant bleiben.

Bereits mit dem einfachen Beispiel des harmonischen, ungedämpften Oszillators lässt sich zeigen, dass diese Herangehensweise der Übertragung der Informationen zwischen Differentialgleichungen außerhalb des Integrationsschrittes zu Verfälschungen führt, die dafür sorgen, dass Verfahren mit adaptiver Schrittweitenregelungen zu vollkommen falschen Ergebnissen führen.

# Design von Interaktionen zwischen Funktionseinheiten

## Überblick

Datenflussprogrammierung bietet unter anderem Möglichkeiten, Agorithmen schnell modellieren und ausprobieren zu können. Dazu werden Komponenten mit standardisierten Schnittstellen visuell auf einem Arbeitstisch arrangiert und ihre Aus- und Eingänge miteinander verbunden. Diese Verbindungen legen den Datenfluss zwischen den Modulen fest.

Die modellierten Workflows als Zusammenfassung der Komponenten und ihrer Verbindungen untereinander können gespeichert werden. Damit kann man sie zu einem späteren Zeitpunkt oder an einer anderen Stelle wiederverwenden. Die vorliegende Anwendung erlaubt es, über eine Programmierschnittstelle Workflows in eigenen Formaten direkt exportieren zu können.

Näheres zum Export in eigenen Formaten ist im Pogrammierhandbuch dWb+ im Kapitel 29, *Exportieren in eigenen Dateiformaten* zu finden.

Die Anwendung geht aber noch weiter; sie kann auch mit Komponentendefinitionen arbeiten, die nicht direkt für sie geschaffen wurden. Innerhalb der Anwendung ist es möglich, verschiedenste Modulspezifikationen zu mischen.

## Moduldefinition

Ein Modul wird durch Metadaten beschrieben, die Auskunft über das Modul, seine Konnektoren zu anderen Modulen und seinen inneren Aufbau geben. Im ersten Schritt wurden folgende Metadaten definiert:

| | |
|---|---|
| Informationen | Dazu gehören der Name des Moduls, eine kurze Beschreibung dessen, was das Modul leistet und eventuell ein Icon zur Visualisierung der Funktion. |
| Slots | Slots liefern Informationen über Kommunikationsschnittstellen, die das Modul zum Informationsaustausch mit anderen Modulen anbietet. Dazu gehören Namen für jeden Slot, Kommunikationsrichtung (Ausgang, Eingang oder beides) und (Daten-)Typ. |

Näheres zu diesem Thema ist im Pogrammierhandbuch dWb+ im Kapitel 30, *Graphische Programmierung für beliebige Komponenten* zu finden

Als Beispiel für die Umsetzung wird hier ein Format genutzt, das heutzutage in aller Munde ist: JSON bietet als simples Format für hierarchische Schlüssel-Wert-Paare alles, was man braucht. Beispiele für

solche Moduldefinitionen ist im Pogrammierhandbuch dWb+ im Abschnitt „Simples Beispiel“ und im Abschnitt „Komplexes Beispiel einschließlich hierarchischer Gliederung“ zu finden.

## Typisierung

Die Anwendung unterstützt den Anwender beim Herstellen von Verbindungen unter anderem dadurch, dass sie die Typen der beiden zu verbindenden Slots auf Kompatibilität prüft: es ist unmöglich, Verbindungen zwischen inkompatiblen Slots herzustellen. Wie funktioniert das nun aber mit vollkommen freien Modul-Spezifikationen? Die hier als Beispiel herangezogene Implementierung geht dazu folgenden Weg: JSON bedeutet ausgeschrieben ja Java Script Object Notation. Javascript ist aber eine typlose Sprache. Daher wurde folgende Lösung ersonnen und implementiert: Fundamentaldatentypen werden auf Java-Typen abgebildet - das bedeutet zum Beispiel, daß ein Slot, für den in der Spezifikation als Typ bool angegeben ist, resultiert in einem Slot des Moduls auf dem Arbeitstisch, das den Java-Typ boolean aufweist. Für Slots, für die keine solche Abbildung bekannt ist, wird einfach transparent zur Laufzeit eine Klasse ohne Inhalt generiert. Diese Klasse wird dann als Typ des Modul-Slots auf der Arbeitsfläche benutzt. Damit kann man auch Slots in so spezifizierten JSON-Spezifikationen nur mit anderen verbinden, wenn sie den gleichen Typ aufweisen.

## Ergebnis

Man kann beliebige Module oder Komponenten in dieser JSON- (oder beliebigen anderen) Spezifikation beschreiben und sie zu einem Workflow kombinieren. Mit einem entsprechenden Export-Modul kann man damit dWb+ als Designwerkzeug für beliebige Workflows benutzen.

Beispiele für Workflows, die mit Spezifikationen im JSON-Format erstellt wurden, lassen sich zum Beispiel im Pogrammierhandbuch dWb+ im Abschnitt „Ergebnisse“ zu finden

# Arbeiten mit der GUI

Die Bedienoberfläche der Anwendung wurde und wird (weiter-) entwickelt immer mit dem Blick auf bedienerfreundliche und effiziente Erstellung komplexer Workspaces. Dabei wurden verschiedene Interaktionsmetaphern implementiert, die helfen sollen, dieses Ziel zu erreichen:

Automatische Verbindungen

Die Anwendung unterstützt das automatische Einrichten von Verbindungen: Normalerweise verbindet man Module, indem man den Ausgang auf den gewünschten Eingang zieht und dort fallenlässt. Öffnet man am Ausgang statt dessen das Kontextmenü, sieht man dort alle Module und alle Eingänge, die mit diesem Ausgang kompatibel sind. Wählt man dort einen aus, wird die entsprechende Verbindung erstellt. Außerdem existiert die Möglichkeit, aus diesem Kontextmenü heraus ein Modul mit einem auf diesen Ausgang passenden Eingang zu instantiieren und direkt zu verbinden. Module, die speziell dafür entwickelt wurden, haben einen bevorzugten Ausgang und oder einen bevorzugten Eingang. Verschiebt man eines dieser Module so, dass sich die Aus- und Eingangsslots berühren, wird - sofern die bevorzugten Aus- und Eingänge vom Typ her kompatibel sind - zwischen diesen eine Verbindung automatisch hergestellt.

Magnetische Hilfslinien und "Besen"

Der Anwender hat die Möglichkeit, ein magnetisches Gitter zu aktivieren, an dem die Module einrasten. Zusätzlich dazu kann man noch vertikale oder horizontale magnetische Hilfslinien hinzufügen. Diese haben die Eigenschaft, wenn sie bewegt werden, während eine Taste auf der Tastatur betätigt wird, zum Besen zu

| | |
|---|---|
| | werden - dann schieben sie alle Module, auf die sie treffen, vor sich her. |
| Automatisches Anordnen relativ zu anderen Modulen | Es existieren verschiedene Möglichkeiten, der automatischen Anordnung von Modulen: Man kann einen simplen Routingalgorithmus nutzen, der versucht, Module so anzuordnen, dass miteinander verbundene stets so positioniert werden, dass sich das Modul mit dem Ausgang links von dem zugehörigen Modul mit dem Eingang befindet. Darüber hinaus sind verschiedene Aktionen verfügbar, wie man sie auch aus Grahikprogrammen kennt: Diese ordnen zum Beispiel alle selektierten Module so an, dass ihre untere Kante auf einer Linie liegt oder die horizontalen Zwischenräume identisch groß sind. |
| | One-Klick Umschalten der Aktivität von Verbindungen |
| Übersichtlichkeit | Die Übersichtlichkeit komplexer Workspaces kann durch hierarchischen Aufbau dieser Workspaces befördert werden. Eine andere Möglichkeit ist es, verschiedene Module unterschiedlichen Ebenen oder Layers zuzuordnen. Die Sichtbarkeit dieser Layer sind selektiv umschaltbar. |

# Mandantenfähigkeit

## Einleitung

Beim Stichwort Mandantenfähigkeit geht es darum, dass es in datenflussorientierten Umgebungen durchaus so sein kann, dass neben den eigentlichen Daten noch Informationen zu ihrer Entstehung für die Verarbeitungseinheiten weiter hinten in der Verarbeitungskette interessant sein könnten. Es soll hier zunächst an einigen Beispielen gezeigt werden, wofür eine solche Mandantenfähigkeit nützlich sein kann.

## Szenarios

### Szenario A - Serverdienst

In diesem ersten Szenario geht es um die Umsetzung eines einfachen HTTP-Servers, der basierend auf der abgeforderten URL eine Datenverarbeitung anstößt und das Ergebnis der Datenverarbeitung an den Anfragenden zurückliefern soll. Dabei stelle man sich den Workflow so vor, dass es einen BeanContextService gibt, der die HTTP-Protokollunterstützung liefert. Auf dem Arbeitstisch werden zwei Module abgelegt: HTTP-Sender und HTTP-Receiver. Das Receiver-Modul gibt die empfangene URL an das nächste Modul weiter. zwischen Sender und Receiver sind beliebig viele andere Module geschaltet, die entsprechend der URL verschiedene Arbeitsschritte durchführen, bis wieder ein String entsteht, der über den Sender an den Client zurückübertragen werden soll.

Da aber HTTP ein Protokoll ist, bei dem sich quasi gleichzeitig beliebig viele Clients mit einem Server verbinden können, existiert im beschriebenen Szenario ein Problem: Wenn sich während der Bearbeitung eines Requests bereits vier neue Clients mit dem Server verbinden, weiß das Sender-Modul am Ende nicht mehr, welchem Client es die Antwort zusenden soll. Das könnte man verhindern, indem man 503 (Service Unavailable) an den Client meldet, solange die Bearbeitung des letzten Requestes nicht abgeschlossen ist. Eine solche Lösung wäre arnselig und nicht mehr zeitgemäß.

Wenn man es aber schaffen könnte, mit den eigentlichen Nutzdaten einen Datenspeicher zu verknüpfen, der Daten zum Kontext der Operation enthält, der die Datenverarbeitung angestoßen hat, und diese

Information über viele verschiedene Komponenten oder Verarbeitungseinheiten hinweg konsistent bliebe, könnte man im Receiver-Modul die Kennung des Client in den Kontext schreiben. Dieser würde bei jeder Kommunikation zweier Module weitergegeben und das Sender-Modul müsste lediglich in den Kontext schauen, um zu wissen, welchem Client es die in Rede stehende Antwortbotschaft übermitteln soll.

### Szenario B - Synchronisation

Das zweite Beispiel ist die Synchronisation, die sich in wenigen Worten wie folgt zusammenfassen lässt: Wenn ein Modul zwei Daten erzeugt und diese über verschiedene Kanäle an unterschiedliche nachfolgende Verarbeitungsketten - also ein oder mehrere gekoppelte Verarbeitungseinheiten - zur Weiterverarbeitung übergibt, ist das ein normales Szenario, das keine besonderen Vorbereitungen bedarf.

Sollen aber die Ergebnisse der beiden Verarbeitungseinheiten am Ende miteinander verschmolzen werden, muss man wissen, welches Ergebnis aus Verarbeitungskette A mit welchem Ergebnis aus der Kette B kombiniert werden soll. Timestamps zur Kennzeichnung der Ergebnisse des ersten Modul fallen aus, da keine zwei Ereignisse in Rechnersystemen wirklich gleichzeitig geschehen. Man muss an die Daten Marker anfügen. Das letzte Modul könnte dann die Marker vergleichen und herausfinden, welche zwei seiner Inputs miteinander kombiniert werden müssten.

Auch hier ist es wieder so, dass diese Marker über die gesamte Kette der Verarbeitungseinheiten konsistent erhalten bleiben müssten.

### Szenario C - Transaktionen

Das dritte Beispiel sind Transaktionen, die sich in datenflussgetriebenen Systemen mittels Mandantenfähigkeit wie folgt umsetzen ließen: In diesem Szenario sind die Verarbeitungen in den einzelnen Modulen einer Verarbeitungskette vergleichbar den atomaren Operationen in einer durch eine Transaktion gekapselten Operation. Das erste Atom eröffnet die Transaktion. Jedes weitere Glied in der Kette muss über die geöffnete Transaktion informiert sein, denn im Fehlerfall muss diese zurückgerollt werden. Das letzte Modul in der Kette muss schließlich dafür sorgen, die Transaktion "zu committen" - also die Ergebnisse persistent zu machen. Auch dazu muss es auf Wissen über die laufende Transaktion zugreifen können.

Dieses Szenario ist ebenfalls ein gutes Beispiel für die Nützlichkeit des Konzeptes der Mandantenfähigkeit oder eines globalen Kontextes:

Diesmal muss die Information über die Transaktion über die gesamte Kette der Verarbeitungseinheiten konsistent erhalten bleiben.

## Umsetzung

Das Framework dWb+ setzt diese Anforderungen so um, dass jedes Modul, das Informationen zu einer auszusendenden Botschaft im Context speichern möchte, dazu einfach eine API-Funktion aufrufen kann. Ein Modul, das zu einer empfangenen Botschaft Informationen aus dem Kontext auslesen möchte, kann dies ebenso wie die Sondierung, ob zugehörige Informationen überhaupt im Context vorliegen, ebenfalls über API-Funktioonsaufrufe realisieren.

# Echtzeitanforderungen

Zum Thema Echtzeitanforderungen in Datenflussprogrammierumgebungen wurde schon viel generelles gesagt - auf berufenem wie unberufenem Munde. Generell ist ein solches System - und das schließt auch dWb+ ein - nicht echtzeitfähig: Sobald mehr Verarbeitungseinheiten zur gleichen Zeit um Prozessorkapazitäten (vulgo: Kerne) konkurrieren, als verfügbar sind, müssen zwangsläufig einige zurückstecken.

Da in einem solchen System aber keine Informationen zentral zur Verfügung steht über Status der einzelnen Verarbeitungseinheiten, kann man auch bur schwerlich einen Scheduler einbauen, der beispielsweise dafür sorgen könnte, bestimmte Verarbeitungseinheiten in festgelegten Intervallen Zugriff auf bestimmte Ressourcen zu garantieren.

Es existieren im System im Auslieferungszustand zum Beispiel Module, die Taktgeber darstellen. Zwei davon sind Clock und HighPrecisionClock. Diese Module senden ein Signal an angeschlosene Konsumenten, wobei der Anwender die zeitlichen Abstände zwischen diesen Signalen angeben kann - bei dem einen in Millisekunden, beim zweiten in Mikrosekunden. Diese Angaben bestimmen aber nicht den Abstand zwischen zwei aufeinanderfolgenden Signalen, sondern nur den *minimalen Abstand zwischen aufeinanderfolgenden Signalen*.

Das Modul Clock kann zur Messung der Abstände aufeinanderfolgender Signale benutzt werden - auf dem Rechner, der Anfang 2017 zur Wartung des Handbuches benutzt wurde, konnte man zum Beispiel sehen, dass in einem Workspace, der nur aus einer HighPrecisionClock und einer Clock bestand, die miteinander verbunden waren, die zeitlichen Abstände der Botschaften bis hinunter zu 50 Mikrosekunden eingehalten wurden. Bei der Einstellung 3 Mikrosekunden kamen die Botschaften in Abständen von rund 6 Mikrosekunden und bei der Einstellung 1 Mikrosekunden kamen sie in Abständen von rund 3 Mikrosekunden. In komplexeren Workspaces können diese Werte natürlich auch noch schlechter ausfallen, auf modernerer und schnellerer Hardware können sie auch besser werden.

# Analyse und Debugging

In Datenfluss-Systemen wie dWb+ existieren selten Möglichkeiten zum Debugging und damit zur Analyse, was momentan im System vorgeht. Die folgenden Abschnitte stellen verschiedene Möglichkeiten der Analyse und Fehlersuche dar, die innerhalb dWb+ zur Verfügung stehen.

## Intelligente Tooltips an Ausgängen

Die Anwendung bietet ein flexibel erweiterbares System von intelligenten Tooltips an, das es erlaubt, für bestimmte Datentypen on Mouseover dedizierte Visualisierungen anzuzeigen. Damit ist es möglich, die Daten, die an einzelnen Ausgängen anliegen, zu inspizieren. Diese Tooltips können fixiert werden, so daß sie nicht sofort wieder verschwinden, wenn die Maus bewegt wird - damit ist es möglich, mehrere Ausgänge zeitgleich zu überwachen. Die Auswahl der jeweiligen Komponente zur Darstellung intelligenter Tooltips für Ausgänge erfolgt anhand des Datentyps des Ausganges. Komponenten zur Änderung der intelligenten Tooltips oder zum neu Hinzufügen von intelligenten Tooltips für bislang noch nicht unterstützte Datentypen werden als Jar-Datei in einen speziellen Ordner des Konfigurationsverzeichnisses gespeichert und stehen nach einem Neustart zur Verfügung.

Näheres zu intelligenten Tooltips findet man im Anwenderhandbuch dWb+ im Abschnitt „Tooltips für Outputs“ auf Seite 49 und im Pogrammierhandbuch dWb+ im Kapitel 27, *StateUpdaters* Informationen zur Struktur des Konfigurationsverzeichnisses sind im Anwenderhandbuch dWb+ im Abschnitt Anhang A, *Verzeichnis-Layout* zu finden.

## Visualisierung

Eine weitere Möglichkeit, den Zustand der im Workspace beteiligten Module zu überwachen, stellen Module dar, die dediziert der Visualisierung dienen. Einige davon werden bereits mit der Anwendung mitgeliefert. Je nach gewünschter Komplexität der Visualisierung ist es einfach neue zu entwickeln, sollten die bereits vorhandenen nicht ausreichen.

Die Standard-Module zur Visualisierung von Daten bieten die Möglichkeit, die Ereignisse über einer echten Zeitskala zu sehen. Darüber hinaus ist es möglich, diese Module über einen Trigger-Eingang zu

steuern: Pre- und Post-Triggering ist möglich. Post-Triggering ist möglich mit History: Man sieht eine einstellbare Anzahl Samples vor dem eigentlichenAuslöser des Triggers.

## Einzelschrittabarbeitung

Relativ neu ist die Möglichkeit, die Übertragung von Botschaften anhalten zu können: Alle Botschaften, die dann übertragen werden sollen, werden statt dessen in eine Warteschlange eingereiht. Der Anwender kann dann mittels Einzelschrittverarbeitung jeweils eine Botschaft freischalten und so die Arbeit des Workspace in Ruhe verfolgen. Die Verbindung, über die in diesem Modus jeweils das nächste Datum übertragen wird, wird visuell hervorgehoben.

# Verteiltes Rechnen

Workspaces können so groß und komplex werden, dass sie mit der geforderten Performanz nicht mehr auf einem einzelnen Rechner abgearbeitet werden können. Dieses Problem wird in datenfluss-basierten Programmierumgebungen dadurch adressiert, dass die Gesamtheit aller zu einem Workspace gehörenden Module über mehrere Rechner verteilt wird. Wie diese Verteilung geschieht, unterscheidet sich jedoch von System zu System.

Bentzt man spezielle Module zum Datenaustausch, kann man mehrere Instanzen von dWb+ zu einem verteilten Workspace zusammenschalten. Allerdings ist dieses Verfahren relativ aufwändig: Zu der Entwicklung der Kopplungsmodule kommt der Aufwand hinzu, auf jedem der beteiligten Rechner die installierten Versionen der Software und der Module konsistent zu halten. Ein weiteres Problem kann in einem solchen Szenario darin bestehen, dass der Start der einzelnen Teilworkspaces in einer bestimmten Reihenfolge vollzogen werden muss.

Daher wurde für das System dWb+ eine weitere Lösung zur Erschaffung verteilter Workspaces implementiert: Der Name der Lösung lautet Remoting, da er auf der Java-Technologie der Remote Method Invocation (RMI) beruht. Module müssen einigen geringfügigen Anforderungen genügen, damit folgendes möglich wird: Nach Instantiierung der Module in einer zentralen dWb+ Instanz kann man über das Kontextmenü der Module deren Funktionalität auf Compute-Server auslagern. Auf diesen Compute-Servern läuft ein Execution Environment, das auf Informationen von der zentralen dWb+ Instanz warten. Diese sendet Informationen an die Compute-Server, wann immer Module Eingangssignale empfangen, die ausgelagert sind. Der Compute-Server erhält alle relevanten Daten (auch die benötigten Java-Klassen!) von der zentralen Instanz, führt die geforderten Berechnungen aus und sendet die Ergebnisse zurück. Das Modul auf der zentrlen Instanz wiederum versendet diese Ergebnisse als seine eigenen.

Die Nutzung eines Compute-Servers auf den anderen beteiligten Rechnern sorgt für sparsamen Ressourcen-Einsatz (der Rechner, auf dem der Compute-Server läuft, kann sogar headless sein) und für automatische Auflösung eventueller Versionskonflikte durch Nachladen der benötigten Klassen von der zentralen dWb+ Instanz.

Für den Einsatz von Modulen im Remoting musste immer eine Implementierung und ein entsprechendes Interface geschaffen werden. Dieser Aufwand konnte durch Nutzung einer weiteren technischen Maßnahme weiter reduziert werden: Durch Benutzung von Skript-Sprachen in Java war es möglich, ein Modul zu schaffen, das Java-Code interpretieren kann. Diese Interpretation kann per Remoting auf andere Rechner verlagert werden.

# Plugins

Die Anwendung dWb+ selbst ist durch Plugins flexibel erweiterbar. Es existiert eine API-Beschreibung zur Dokumentation der für Plugins zur Verfügung stehenden Anknüpfungspunkte. Damit ist es zum Beispiel möglich, mehrere Module synchron zu steuern oder neue Komponenten zum Dock hinzuzufügen.

Geplant ist, diese Schnittstelle so zu erweitern, dass darüber neue Metamodule definiert und zur Anwendung hinzugefügt werden können.

## Skripted Module für noch schnelleres Rapid Prototyping

Der Entwicklungszyklus für die Erstellung neuer Module ist normalerweise Editieren, Compilieren, Ausrollen. Die Anwendung dWb+ bietet darüber hinaus einen schnelleren Entwicklungszyklus, der zwar PErformance-einbußen mit sich bringt, dafür aber sehr viel schnellere Turnaround-Zeiten mit sich bringt:

Die Anwendung dWb+ erlaubt es, Module in Scriptsprachen zu definieren und wie kompilierte zu benutzen. Dazu existiert beispielsweise ein Plugin, das die Sprache Groovy dafür benutzbar macht. Fest eingebaut ist diese Fähigkeit für die Skriptsprache BeanShell - ein interpretiertes Java. Dieses Java kann zwar noch nicht mit dem in Java 5 eingebauten Sprachfeature Generics (und allen in höheren Versionen hinzugekommenen Sprachfeatures) umgehen, ist dafür aber in der Lage, JavaBeans als Klassen zu behandeln.

Damit ist es möglich, eine Klasse für ein neues Modul, das gerade entwickelt wird, über diesen Weg als Modul zu nutzen und zu testen. Wenn die Funktionalität augereift ist, kann man es dann compilieren undauf herkömmlichem Wege ausrollen, was dann zu einem Performanceschub führt.

Informationen zur Struktur des Konfigurationsverzeichnisses sind im Anwenderhandbuch dWb+ im Abschnitt „Skript-Module“ zu finden.

# Programmstart

## System-Properties

Der Start der Anwendung wird wie der jeder anderen Java-Anwendung auch vollzogen. Es existieren einige Parameter, die beim Programmstart spezifiziert werden können/müssen. Diese werden als System-Parameter - auch System-Properties genannt - spezifiziert. Das bedeutet, dass sie mittels vorangestelltem -D in der Kommandozeile beim Programmstart als Schlüssel=Wert eingefügt werden.

| | |
|---|---|
| java.security.auth.login.config | Diese System-Property gibt an, aus welcher Datei die Konfiguration für den Java Authentication and Authorization Service (JAAS) gelesen werden soll. Das ist wichtig für die Verwaltung der Rollen und Rechte in der Anwendung. Genauere Informationen dazu finden sich im Kapitel Kapitel 6, *Rollen und Rechte*.<br><br>Die Angabe ist nur dann notwendig, wenn die Anwendung das Rollen- und Rechtesystem benutzen soll. In diesem Fall ist mittels `-Djava.security.manager` ein SecurityManager anzugeben. |
| java.security.policy | Diese System-Property legt den Namen der Datei fest, aus der die Berechtigungen für die einzelnen Rollen und Rechte gelesen werden sollen. Das ist wichtig für die Verwaltung der Rollen und Rechte in der Anwendung. Genauere Informationen dazu finden sich im Kapitel 6, *Rollen und Rechte*.<br><br>Die Angabe ist nur dann notwendig, wenn die Anwendung das Rollen- und Rechtesystem benutzen soll. In diesem Fall ist mittels `-Djava.security.manager` ein SecurityManager anzugeben. |
| java.security.manager | Diese System Property legt den Klassennamen des zu benutzenden Security-Managers fest. Damit die Verwaltung |

| | |
|---|---|
| | der Rollen und Rechte wie in Kapitel 6, *Rollen und Rechte* beschrieben funktioniert, muss ein Security-Manager aktiv sein. Enthält diese System-Property keine Angaben (`-Djava.security.manager=""`), wird der Standard-Security-Manager benutzt.<br><br>Die Angabe ist nur dann notwendig, wenn die Anwendung das Rollen- und Rechtesystem benutzen soll. In diesem Fall muß die Konfiguration durch Angeba von `-Djava.security.auth.login.config` und `-Djava.security.policy` vervollständigt werden.. |
| DynamicSVG.proxy.uri | Diese URL beschreibt die Kontaktadresse des Proxy für den Aviator wie in Kapitel 5, *Meta-Module* in „Actions“ beschrieben. Diese URL enthält die Angaben einschließlich des Kontextes - also zum Beispiel http://www.netsys-it.de/servlet/dynamicsvgproxy. Wird dieser Parameter nicht spezifiziert, kann die Publikation über Proxy nicht ausgewählt werden. |
| dwb.config.dir | Diese System-Property spezifiziert den Namen des zu benutzenden Datenverzeichnisses, das unter anderem die Module, die Konfiguration und die verschiedenen Ressourcen zum Beispiel für das Meta-Modul Aviator enthält. Die genauen Inhalte dieses Verzeichnisses sind in Anhang A, *Verzeichnis-Layout* beschrieben. Ebenfalls dort findet man Informationen über den Standort dieses Verzeichnisses, falls dieser Parameter nicht explizit angegeben wird. |
| SVGChooserPanel.updateBitmaps | Diese System-Property entscheidet darüber, ob der Dialog zur Auswahl der hinzuzufügenden Instrumente im Meta-Modul Aviator (beschrieben in Kapitel 5, *Meta-Module* in „Konfiguration“) die Vorschau-Darstellungen der Instrumente bei Bedarf (nicht vorhanden oder SVG neuer als vorhandene) neu erzeugen soll (-DSVGChooserPanel.updateBitmaps=true) oder nicht (-DSVGChooserPanel.updateBitmaps=false). Wird dieser Parameter beim Start nicht angegeben, wirkt das genauso wie die Angabe -DSVGChooserPanel.updateBitmaps=true. |
| dWb.service.ApplicationServer.port | Diese System-Property konfiguriert den Dienst zur Publikation von Informationen mittels Servlet-Engine. Manche Module können mittels dieses Dienstes Informationen im Web bereitstellen. Ein Beispiel für ein solches Modul ist das in Kapitel 5, *Meta-Module* in „Aviator“ beschriebene Meta-Modul Aviator. Wird dieser Parameter beim Start nicht angegeben, steht dieser Dienst nicht zur Verfügung. Nähere Informationen zu diesem Dienst sind im Programmierhandbuch in Anhang B, *BeanContext Services* in „ApplicationServer“ zu finden.<br><br>Diese Einstellung kann auch per Umgebungsvariable gesetzt werden. Das ist speziell für Windows-Anwender wichtig, die die Anwendung über die Exe-Datei starten und daher keine Java-System-Properties angeben können |
| dWb.PropagationManager.use | Diese System-Property die Debugging-Hilfen in Workspaces wie in „Einzelschrittabarbeitung“ im Anwenderhandbuch dWb+ |

beschrieben. Wird dieser Parameter beim Start nicht angegeben, stehen die Debugging-Hilfsmittel nicht zur Verfügung. Dies ist die Defaulteinstellung: Die Verfügbarkeit der Hilfsmittel führt zu einem geringfügigen Performanceverlust - daher sollte man sie nur dann aktivieren, wenn sie wirklich benötigt werden. Nähere Informationen dazu sind im Anwenderhandbuch dWb+ in Tabelle 1.3, „Menü Steuerung“ zu finden.

Diese Einstellung kann auch per Umgebungsvariable gesetzt werden. Das ist speziell für Windows-Anwender wichtig, die die Anwendung über die Exe-Datei starten und daher keine Java-System-Properties angeben können

## Umgebungsvariablen

Alternativ können auch für einige dieser Parameter Umgebungsvariablen gesetzt werden, die dann die gleichen Auswirkungen haben:

dWb.service.ApplicationServer.port — Diese Umgebungsvariable konfiguriert den Dienst zur Publikation von Informationen mittels Servlet-Engine. Manche Module können mittels dieses Dienstes Informationen im Web bereitstellen. Ein Beispiel für ein solches Modul ist das in Kapitel 5, *Meta-Module* in „Aviator“ beschriebene Meta-Modul Aviator. Wird diese Parameter nicht konfiguriert, steht dieser Dienst nicht zur Verfügung. Nähere Informationen zu diesem Dienst sind im Programmierhandbuch in Anhang B, *BeanContext Services* in „ApplicationServer“ zu finden.

Diese Einstellung kann alternativ auch als Java-System-Property angegeben werden.

dWb.PropagationManager.use — Diese System-Property die Debugging-Hilfen in Workspaces wie in „Einzelschrittabarbeitung“ im Anwenderhandbuch dWb+ beschrieben. Wird dieser Parameter beim Start nicht angegeben, stehen die Debugging-Hilfsmittel nicht zur Verfügung. Dies ist die Defaulteinstellung: Die Verfügbarkeit der Hilfsmittel führt zu einem geringfügigen Performanceverlust - daher sollte man sie nur dann aktivieren, wenn sie wirklich benötigt werden. Nähere Informationen dazu sind im Anwenderhandbuch dWb+ in Tabelle 1.3, „Menü Steuerung“ zu finden.

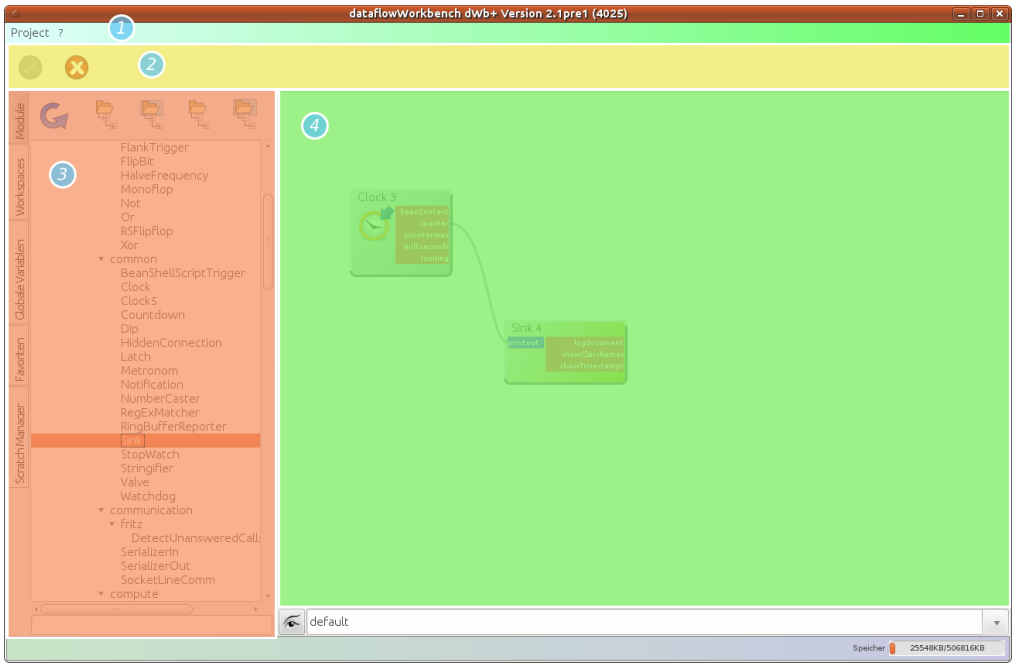
Diese Einstellung kann alternativ auch als Java-System-Property angegeben werden.

# Aufbau der Bedienoberfläche

Nach dem Start der Anwendung erscheint zunächst ein Splashscreen, der die Zeit bis zum Öffnen des Hauptfensters überbrückt.

Das Hauptfenster gliedert sich in vier große Bereiche: Oben sind das Menü und eine Werkzeugleiste angeordnet. Unten wird das Fenster durch eine Statusleiste abgeschlossen, in der der aktuelle Speicherbedarf der Anwendung angezeigt wird. Die Mitte des Fensters schließlich teilen sich das Dock und der Workspace.

### Abbildung 1.1. Bereiche des Hauptfensters




Die aktuelle Größe und Position des Hauptfensters werden beim Beenden der Anwendung automatisch gespeichert, so dass man das Hauptfenster nach dem erneuten Start exakt an dieser Position und in dieser Größe wiederfindet.

Die nächsten beiden Abschnitte stellen die im Hauptmenü und in der Werkzeugleiste verfügbaren Actions - in Abbildung 1.1, „Bereiche des Hauptfensters“ die Abschnitte 1 und 2 - genauer vor. Für die Komponenten in der Mitte sind jeweils eigene Abschnitte reserviert: Die im Dock zusammengefassten Komponenten werden im Kapitel 3, *Dock* beschrieben. Mit den Möglichkeiten und der Funktion des Workspaces an sich befasst sich Kapitel 2, *Workspace*.

# Hauptmenü der Anwendung

Das Hauptmenü der Anwendung befindet sich im Hauptfenster oben wie in Abbildung 1.1, „Bereiche des Hauptfensters“ (Abschnitt 1) zu sehen.

### Tabelle 1.1. Menü Projekt

Öffnet einen Workspace aus einer Datei, deren Inhalte die des aktuell bearbeiteten Workspaces ersetzen.

Bereits vorhandene Workspace-Elemente wie etwa Module, Verbindungen und Skripts werden aus dem Workspace entfernt. Dabei unter Umständen noch nicht fertig berechnete Algorithmen der entfernten Module werden unterbrochen. Sind die Module korrekt implementiert, werden auch alle von den zu entfernenden Modulen allokierten Ressourcen wieder freigegeben.

Importiert einen Workspace aus einer Datei und fügt die darin enthaltenen Module und Verbindungen zum aktuellen hinzu.

Bereits vorhandene Workspace-Elemente wie etwa Module, Verbindungen und Skripts werden nicht aus dem Workspace entfernt. Algorithmen werden nicht

unterbrochen. Von bereits vorhandenen Modulen allokierte Ressourcen werden nicht freigegeben.

Speichert den aktuellen Inhalt des Workspaces einschließlich aller Module und Verbindungen in eine XML-Datei.

Die Datei wird - falls sie schon existiert - nur nach nochmaliger Frage an den Anwender überschrieben.

Speichert den Workspace in einer SVG- oder PDF-Graphik.

Diese Action dient Dokumentationszwecken: Der gesamte Inhalt des Workspaces wird als Graphik in einer Datei gespeichert. Dabei entscheidet die Dateinamensendung über das Format. Wird keine Endung angegeben, wird immer svg angenommen.

Die Datei wird - falls sie schon existiert - nur nach nochmaliger Frage an den Anwender überschrieben.

Speichert den ausgewählten Teil des Workspace in einer SVG- oder PDF-Graphik

Diese Action dient Dokumentationszwecken: Der selektierte Inhalt des Workspaces wird als Graphik in einer Datei gespeichert. Selektierter Inhalt sind dabei alle selektierten Module sowie alle Verbindungen, die sowohl in einem selektierten Modul beginnen wie auch enden. Dabei entscheidet die Dateinamensendung über das Format. Wird keine Endung angegeben, wird immer svg angenommen.

Die Datei wird - falls sie schon existiert - nur nach nochmaliger Frage an den Anwender überschrieben.

### Tabelle 1.2. Werkzeuge

Schreibt ein Beispiel für eine Logging Konfigurationsdatei in das Konfigurationsverzeichnis der Anwendung.

Diese Konfigurationsdatei (deren genauer Name einschließlich des Verzeichnisses in einem Dialog angezeigt wird) namens dffw.template kann als Ausgangspunkt eigener Logging-Konfigurationen dienen. Solche Konfigurationen werden im selben Verzeichnis gesucht, wo auch das Beispiel erzeugt wurde - allerdings unter dem Namen dffw.props.

### Tabelle 1.3. Menü Steuerung

Im Einzelschrittmodus werden Daten nur über Verbindungen übertragen, wenn dies angefordert wird; mit jeder Anforderung wird genau eine Botschaft übertragen. Der Einzelschrittmodus ist immer dann aktiv, wenn dieser Menüpunkt ausgewählt ist.

Fordert die Übertragung einer Botschaft im Einzelschrittmodus an. Dieser Menüpunkt ist nur verfügbar, wenn der Einzelschrittmodus aktiviert ist und mindestens eine Botschaft auf Übertragung wartet.

### Tabelle 1.4. Menü Information (?)

Informationen zum Programm

Diese Action öffnet einen Dialog, in dem unter anderem folgende Informationen zu finden sind:

- Der genaue Name und die genaue Versionsnummer der Anwendung

- Informationen über die benutzten Komponenten
- Der Ort und Name des Konfigurations- und Datenverzeichnisses - Inhalt und Struktur dieses Verzeichnisses sind im Anhang A, *Verzeichnis-Layout* zu finden.

Öffnet einen Browser, der die neuesten Informationen zur Anwendung präsentiert

Dieses Untermenü enthält die verschiedenen Themes, die der Anwender für die Darstellung der Benutzeroberfläche auswählen kann. Die hier getroffene Einstellung ist persistent: Beim Neustart der Anwendung wird das letzte ausgewählte Theme automatisch geladen und benutzt.

# Hauptwerkzeugleiste der Anwendung

Die Hauptwerkzeugleiste der Anwendung befindet sich im Hauptfenster direkt unterhalb des Hauptmenüs wie in Abbildung 1.1, „Bereiche des Hauptfensters“ (Abschnitt 2) zu sehen.

### Tabelle 1.5. Actions in der Hauptwerkzeugleiste

Login

Diese Action öffnet einen Dialog, in dem der Anwender sich für die Anwendung mittels Login und Passwort authentifizieren kann. Ohne Authentifizierung kann ein Anwender mit der Anwendung nur sehr eingeschränkt interagieren. Je nach der Identität, die er durch den Login-Vorgang annimmt, werden ihm mehr oder weniger Rechte eingeräumt. Genauere Informationen zu Rollen und Rechten sind in Kapitel 6, *Rollen und Rechte* zu finden.

DieAction steht nur dann zur Verfügung, wenn die Anwendung das Rollen- und Rechtesystem benutzen soll. In diesem Fall ist mittels `-Djava.security.manager` ein SecurityManager anzugeben.

Logout

Diese Action entzieht dem Anwender alle Rechte, die er durch ein zuvor erfolgtes erfolgreiches Login eventuell erlangt hat.

DieAction steht nur dann zur Verfügung, wenn die Anwendung das Rollen- und Rechtesystem benutzen soll. In diesem Fall ist mittels `-Djava.security.manager` ein SecurityManager anzugeben.

# Kapitel 2. Workspace

## Überblick

Die Anwendung dWb+ erlaubt es, Datenverarbeitungen visuell zu modellieren. Dazu werden Funktionseinheiten - die sogenannten Module - auf die Arbeitsfläche platziert und die Ein- und Ausgänge der Module so miteinander verbunden, dass die so entstehende Verarbeitungskette das gewünschte Ergebnis erzeugt.

Die Gesamtheit der Module und Verbindungen - alles, was auf der Arbeitsfläche angeordnet ist - bezeichnet man als Workspace. Der Workspace ist in Abbildung 1.1, „Bereiche des Hauptfensters“ als Abschnitt 3 gekennzeichnet. Workspaces können gespeichert und später wieder eingeladen werden. Die in ihnen produzierten und verarbeiteten Daten können protokolliert werden. Es ist möglich, Vektor- und Bitmapgraphiken vom Aussehen beliebiger Workspaces zu Dokumentationszwecken zu erzeugen. Workspaces können hierarchisch organisiert werden: Workspaces können andere Workspaces - sogenannte Gruppen - enthalten. Zwischen Workspaces und in ihnen enthaltenen Gruppen können Daten ausgetauscht werden: Es ist ebenso möglich, dass ein Workspace Daten an in ihm enthaltene Gruppen sendet, wie im Workspace Daten aus enthaltenen Gruppen zu empfangen.

Workspaces sind wie moderne Graphikprogramme in Ebenen oder Schichten organisiert: Module gehören immer zu genau einer Schicht, Verbindungen können zu einer oder zwei Schichten gehören[1]. Man kann beliebig viele Schichten innerhalb eines Workspace definieren. Beliebig viele Schichten eines Workspace können ausgeblendet werden. Ist eine Schicht ausgeblendet, werden alle Elemente, die zu dieser Schicht gehören, unsichtbar. Eine Verbindung, die zu zwei Schichten gehört, wird dann unsichtbar, wenn mindestens eine der Schichten ausgeblendet ist, zu denen sie gehört.

Zur Erleichterung der Organisation und zur Erhöhung der Übersichtlichkeit kann man Gruppen selektierter Module (und die zugehörigen Verbindungen)[2] per Refactoring über das im Kapitel 4, *Module* beschriebene Modul-Kontextmenü in Gruppen oder Ebenen verlagern.

Workspaces können größer als die sichtbare Arbeitsfläche sein. Sobald sich mindestens eins der Module im Workspace außerhalb des sichtbaren Bereiches der Arbeitsfläche befindet, werden Scrollbalken eingeblendet, mit denen man diesen sichtbaren Ausschnitt verschieben kann. Darüber hinaus existieren weitere Navigationshilfen für den Workspace, die über das im nächsten Abschnitt vorgestellte Kontextmenü aufgerufen werden können.

Dieses Kontextmenü enthält auch Actions zur Anwendung auf Verbindungen. Diese Actions sind nur dann verfügbar, wenn der Mauszeiger über einer Verbindung schwebt. Diese Verbindung wird dann von der Action betroffen. Verbindungen können in zwei verschiedenen Stati existieren: aktiv und inaktiv. Je nach Status unterscheidet sich die visuelle Darstellung: Aktive Verbindungen werden als durchgezogene Linien dargestellt, inaktive als durchbrochene Linien. Die Verbindung, auf die die jeweilige Action wirken wird, kann der Anwender ebenfalls anhand der visuellen Darstellung identifizieren: diese wird dicker dargestellt. Außer den im Kontextmenü zusammengefassten Actions, die auf Verbindungen wirken können, existieren noch zwei weitere Interaktionsmöglichkeiten: Ein Klick mit der linken Maustaste auf eine solche Verbindung schaltet sie zwischen den Stati aktiv und inaktiv um. Klickt man bei gehaltener Feststelltaste auf eine Verbindung, wird diese sofort entfernt.

[1] Verbindungen gehören zu zwei Schichten genau dann, wenn die beteiligten Module zu unterschiedlichen Schichten gehören
[2] Gruppen selektierter Module samt ihrer Verbindungen werden auch als Workspacefragmente bezeichnet

# Kontextmenü

### Tabelle 2.1. Kontextmenü im Workspace

| | |
|---|---|
| | Entfernt alle Module und Verbindungen aus diesem Workspace. |
| | Diese Action ist nur für den Stammworkspace verfügbar. Gruppen-Workspaces wie in „Gruppe“ in Kapitel 5, *Meta-Module* beschrieben bieten diese Action nicht an. |
| | Importiert einen Workspace aus einer Datei und fügt die darin enthaltenen Module und Verbindungen zum aktuellen hinzu. |
| | Bereits vorhandene Workspace-Elemente wie etwa Module, Verbindungen und Skripts werden nicht aus dem Workspace entfernt. Algorithmen werden nicht unterbrochen. Von bereits vorhandenen Modulen allokierte Ressourcen werden nicht freigegeben. |
| | Diese Action ist nur für Gruppen-Workspaces wie in „Gruppe“ in Kapitel 5, *Meta-Module* beschrieben verfügbar. |
| | Importiert einen Workspace aus einer Datei als Unterworkspace für den aktuellen. |
| | Diese Action erstellt zunächst ein neues Meta-Modul vom Typ Gruppe wie in „Gruppe“ in Kapitel 5, *Meta-Module* beschrieben. Anschließend wird im Workspace dieses Moduls der Workspace aus der gewählten Datei eingeladen. Diese Action ist besonders dann nützlich, wenn man einen größeren Workspace geplant, diesen dann partitioniert hat, um die einzelnen Komponenten zu entwickeln. |
| | Mittels dieser Action ist es möglich, die Komponenten als Gruppen in einem Workspace zusammenzuführen und diese Gruppen dann zu verschalten. |
| | Speichert den aktuellen Inhalt des Workspaces einschließlich aller Module und Verbindungen in eine XML-Datei. |
| | Die Datei wird - falls sie schon existiert - nur nach nochmaliger Frage an den Anwender überschrieben. |
| | Diese Action ist nur für Gruppen-Workspaces wie in „Gruppe“ in Kapitel 5, *Meta-Module* beschrieben verfügbar. |
| | Speichert den aktuellen Inhalt des Workspaces. |
| | Diese Action speichert den Inhalt des aktuellen Workspaces zum schnellen Zugriff in einem extra dafür angelegten Verzeichnis. Die in „Workspaces“ in Kapitel 3, *Dock* beschriebene Komponente dient der Verwaltung dieser Workspaces. |
| | Die Action fragt dafür zunächst den Namen ab, unter dem der Inhalt des aktuellen Workspaces gespeichert werden soll. Bei Namensgleichheit mit einem bestehenden wird nochmals nachgefragt, ob der bestehende mit den neuen Daten überschrieben werden soll. |
| | Damit der neue Workspace in der Komponente zur Verwaltung der Workspaces auftaucht, muss in dieser Komponente die Action ausgeführt werden, die die Liste der verfügbaren Workspaces neu lädt. |
| | Exportiert die Inhalte dieses Workspace - das Format hängt dabei von der Dateinamensendung ab. |

Speichert den Workspace in einer SVG- oder PDF-Graphik.

Diese Action dient Dokumentationszwecken: Der gesamte Inhalt des Workspaces wird als Graphik in einer Datei gespeichert. Dabei entscheidet die Dateinamensendung über das Format. Wird keine Endung angegeben, wird immer svg angenommen.

Die Datei wird - falls sie schon existiert - nur nach nochmaliger Frage an den Anwender überschrieben.

Diese Action ist nur für Gruppenworkspaces wie in „Gruppe“ in Kapitel 5, *Meta-Module* beschrieben verfügbar.

Speichert den ausgewählten Teil des Workspace in einer SVG- oder PDF-Graphik

Diese Action dient Dokumentationszwecken: Der selektierte Inhalt des Workspaces wird als Graphik in einer Datei gespeichert. Selektierter Inhalt sind dabei alle selektierten Module sowie alle Verbindungen, die sowohl in einem selektierten Modul beginnen wie auch enden. Dabei entscheidet die Dateinamensendung über das Format. Wird keine Endung angegeben, wird immer svg angenommen.

Die Datei wird - falls sie schon existiert - nur nach nochmaliger Frage an den Anwender überschrieben.

Diese Action ist nur für Gruppenworkspaces wie in „Gruppe“ in Kapitel 5, *Meta-Module* beschrieben verfügbar.

Verbirgt alle unbenutzen Ein- und Ausgänge in allen Modulen.

Alle Module verfügen über Ein- und Ausgänge - sogenannte Slots, über die sie Daten von anderen Modulen empfangen oder an andere Module senden. Sind in einem Workspace alle benötigten Verbindungen zwischen den beteiligten Modulen angelegt, kann man die Übersichtlichkeit durch Ausführen dieser Action verbessern: Alle Slots, an die keine Verbindungen angeschlossen sind, werden ausgeblendet.

Falls versehentlich ein Slot ausgeblendet wurde, der noch für eine Verbindung benötigt wird, ist das nicht schlimm: Die Module verfügen über Actions, mit denen man alle oder nur ausgewählte unsichtbare Slots wieder sichtbar machen kann.

Auswahl

**Tabelle 2.2. Untermenü zum Ändern der selektierten Elemente**

| |
|---|
| Alles wird selektiert |
| Alles, was nicht ausgeblendet wurde wird selektiert |
| Selektiertes wird deselektiert, Deselektiertes wird selektiert |
| Alles im aktuellen Layer wird selektiert |
| Selektiertes wird deselektiert, Deselektiertes wird selektiert |

**Tabelle 2.3. Untermenü zum Umschalten des Modus**

| |
|---|
| Dieser Modus erlaubt es zum Beispiel, Verbindungen zu de-/aktivieren |
| Dieser Modus erlaubt es, den Verlauf der Verbindungen zwischen Modulen anzupassen |
| Dieser Modus erlaubt das Anpassen der relativen Abstände aller Module mittels Mausrad. |

Dieses Untermenü enthält Aktionen zur Erzeugung von Hilfslinien. Module rasten während des Verschiebens an vertikalen und horizontalen Hilfslinien ein, so daß sie leichter in einem frei definierbaren Raster ausrichtbar sind. Hilfslinien können mit der Maus verschoben werden. Hilfslinien werden gelöscht, indem sie über den Rand des Workspace hinaus verschoben werden.

Darüber hinaus ist es möglich, ein regelmäßiges Gitter von feststehenden Hilfslinien ein- und auszublenden.

### Tabelle 2.4. Untermenü für Hilfslinien

| | |
|---|---|
| | Fügt an der Position, an der das Popup geöffnet wurde, eine neue horizontale Hilfslinie hinzu |
| | Fügt an der Position, an der das Popup geöffnet wurde, eine neue horizontale und vertikale Hilfslinie hinzu |
| | Fügt an der Position, an der das Popup geöffnet wurde, eine neue vertikale Hilfslinie hinzu |
| | Schaltet das Hilfsliniengitter an und aus |
| | Entfernt selektierte Hilfslinien |
| | Module leisten Widerstand, wenn sie übereinander gezogen werden |

Fügt kopierte Module in diesen Workspace ein.

Diese Action versucht, den aktuellen Inhalt der Zwischenablage als Menge von Modulen und Verbindungen zu interpretieren. Falls das nicht geht, macht diese Action einfach gar nichts. Falls sich ein Workspacefragment in der Zwischenablage befindet, wird es rekonstruiert und die enthaltenen Module und Verbindungen zum aktuellen Workspace hinzugefügt.

Dieses Untermenü gruppiert Actions im Zusammenhang mit Verbindungen. Diese Actions sind nur dann verfügbar, wenn sich der Mauszeiger beim Öffnen des Kontextmenüs über einer Verbindung befunden hat - das erkennt man auch daran, dass die Verbindung anders eingefärbt wird, sobald sich der Mauszeiger über ihr befindet.

Zusätzlich zu den Actions im Kontextmenü existieren auch folgende weitere Möglichkeiten zur Interaktion mit Verbindungen unter dem Mauszeiger: ein Klick mit der linken Maustaste auf eine solche Verbindung schaltet sie zwischen den Stati aktiv und inaktiv um. Klickt man bei gehaltener Feststelltaste auf eine Verbindung, wird diese sofort entfernt.

### Tabelle 2.5. Untermenü Verbindungen

| | |
|---|---|
| | Entfernt die ausgewählte Verbindung aus dem Workspace. |
| | Die Verbindung wird tatsächlich entfernt. |
| | Gleiches erreicht man mit einem Klick auf die Verbindung mit der linken Maustaste bei gleichzeitig gehaltener Feststelltaste. |
| | Umschalten des Verbindungsstatus zwischen inaktiv/aktiv. |
| | Die Verbindung bleibt erhalten, überträgt aber im Zustand inaktiv keine Daten. Visuell ist das an der Darstellung der Verbindung |

zu erkennen: aktibe Verbindungen werden mittels durchgezogener Linien dargestellt, inaktive mittels gepunkteter Linien.

Gleiches erreicht man mit einem Klick auf die Verbindung mit der linken Maustaste.

Das Unterdrücken der Weiterleitung der zu übertragenden Daten kann man auch abhängig von den zu übertragenden Daten über ein entsprechendes Skript an der Verbindung erreichen, wie im Anwenderhandbuch dWb+ im Abschnitt „Kontextmenü“ [25] auf Seite 25 zu ersehen ist.

### Tabelle 2.6. Untermenü Farbverwaltung

| | |
|---|---|
| Ausgewählte Verbindung einfärben | Erlaubt es, die Farbe für diese Verbindung zu ändern. Diese Aktion öffnet einen Dialog, in dem der Anwender eine Farbe auswählen kann. Wurde die Aktion für eine bestimmte Verbindung ausgeführt, wird diese mit der ausgewählten Farbe eingefärbt, falls der Dialog bestätigt wurde. Wurde die Aktion nicht für eine spezielle Verbindung ausgeführt, sieht man zunächst keine unmittelbare Auswirkung. Statt dessen wurde in diesem Fall die voreingestellte Farbe für neue Verbindungen konfiguriert. |
| Zufällige Farben für alle Verbindungen | Färbt alle Verbindungen unterschiedlich ein |
| Alle Verbindungen auf Voreinstellung zurücksetzen | Färbt alle Verbindungen mit der Standardfarbe ein |
| Zufällige Farben für alle Verbindungen mit derselben Quelle | Färbt alle Verbindungen mit derselben Quelle wie die ausgewählte unterschiedlich ein |
| Zufällige Farben für alle Verbindungen mit demselben Ziel | Färbt alle Verbindungen mit demselben Ziel wie die ausgewählte unterschiedlich ein |
| Alle Verbindungen mit | Färbt alle Verbindungen mit derselben Quelle wie die ausgewählte mit der Standardfarbe ein |

derselben Quelle auf Voreinstellung zurücksetzen

Alle Verbindungen mit demselben Ziel auf Voreinstellung zurücksetzen
: Färbt alle Verbindungen mit demselben Ziel wie die ausgewählte mit der Standardfarbe ein

Verbindung mit derselben Quelle einfärben
: Öffnet einen Farbauswahldialog und färbt die ausgewählte Verbindung und alle mit derselben Quelle mit der dort gewählten Farbe ein

Verbindung mit demselben Ziel einfärben
: Öffnet einen Farbauswahldialog und färbt die ausgewählte Verbindung und alle mit demselben Ziel mit der dort gewählten Farbe ein

Jede neue Verbindung verschieden einfärben
: Alle Verbindungen bekommen eine individuelle zufällige Farbe zugewiesen

Öffnet einen Dialog zur Anpassung des zugehörigen Skripts. Abbildung 2.2, „Dialog zur Bearbeitung des Skripts an einer Verbindung“ zeigt ein Beispiel für einen solchen Dialog. Die in diesem Editor zur Verfügung stehenden Actions sind in Abschnitt „Actions im Editor von Skripts für Verbindungen“ näher erläutert.

Diese Action gestattet es, Datentransformationen während der Übertragung zu definieren. Dazu ein einfaches Beispiel: Ein Modul A liefert Zahlen, die an ein anderes Modul übertragen werden sollen. Allerdings benötigt man eine Datenanpassung dahingehend, dass an Modul B der doppelte Wert erwartet wird. Man kann dafür ein Modul implementieren wie in Kapitel beschrieben. In einem solchen Fall ist es aber einfacher, die Transformation direkt an der Verbindung festzulegen. Das sieht für das beschriebene Beispiel wie folgt aus:

```
_data_=2*_data_;
```

Die Variable _data_ enthält zu Beginn der Ausführung des Skripts den Wert, den Modul A versenden möchte. Die Transformation muss diesen Wert ändern - nur so kommt das Resultat an Modul B an.

Der Code, der für die Transformation geschrieben wird, kann den vollen Syntaxumfang von Java in der Sprachversion 1.4 benutzen.

Man kann in solchen Skripts auch globale Variablen nutzen, wie das im Anwenderhandbuch dWb+ im Abschnitt „Globale Variablen“ auf Seite 36 beschrieben wird.

Eine weitere Möglichkeit der Beeinflussung des Verhaltens von Verbindungen ist, die Übertragung des Datums komplett zu verhindern: Dazu setzt man innerhalb des Skriptes einfach die Variable `_suppressed_` auf einen beliebigen Wert- beispielsweise so:

```
if(_data_>5)
    _suppress_=true;
```

Die Verbindung wird aufgetrennt - statt dessen werden zwei Module zwischengeschaltet, die den gleichen inneren Zustand haben.

Diese Action ist in Workspaces sinnvoll, die nicht einen streng linearen Ablauf von Datenquelle zu Datensenke aufweisen: wenn sich zu viele Verbindungslinien kreuzen und der Workspace dadurch unübersichtlich wird, kann man mit dieser Action Verbindungen visuell trennen.

Man kann sich das so vorstellen, dass ein Draht, der die Verbindung symbolisiert, durch zwei Löcher - die neuen Module - gesteckt wird und "unter" dem Workspace verläuft.

## Anmerkung

Dadurch entstehen zwei neue Verbindungen. Ein Skript, das der Verbindung zugeordnet war, die aufgetrennt wird, wird der neuen Verbindung zugeordnet, die mit der Datensenke verbunden ist.

Holt die letzte Botschaft vom Sender und überträgt sie nochmals an den Empfänger.

Diese Action kann nützlich werden, wenn man gerade einen Workspace aufbaut und mit unterschiedlichen Modulen experimentieren möchte, allerdings das Modul, das die Daten liefert, nur sehr selten arbeitet. Statt dann eine halbe Stunde auf jede Botschaft warten zu müssen, kann man mittels dieser Action einfach die letzte Botschaft nochmals versenden lassen. Näheres dazu in „Physikalische Größe gegen Botschaften“ und „Wiederholte gleiche Werte“.

Allerdings sollte man dabei beachten, dass diese neu geschaffenen Botschaften keinerlei Context haben (können). Näheres dazu in „Mandantenfähigkeit“.

Zeigt das Modul, von dem die Verbindung ausgeht.

Damit ist es möglich, in komplexen Workspaces schnell die an einer Verbindung beteiligten Module zu finden - diese Action hebt das Modul hervor, von dessen Output diese Verbindung ausgeht.

Zeigt das Modul, an dem die Verbindung endet.

Damit ist es möglich, in komplexen Workspaces schnell die an einer Verbindung beteiligten Module zu finden - diese Action hebt das Modul hervor, an dessen Input diese Verbindung endet.

Fügt einen weiteren Kontrollpunkt in der Mitte des Verbindungsabschnitts hinzu

Löscht alle manuell vorgenommenen Einstellungen und hinzugefügten Kontrollpunkte

Dieses Untermenü gruppiert Actions, mit denen man schneller einen Überblick in komplexen Workspaces gewinnen kann. Die verschiedenen Hervorhebungen werden aktiv, wenn der Mauszeiger sich über einem Modul befindet.

### Tabelle 2.7. Untermenü Hervorheben

| | |
|---|---|
| Nichts | Nichts wird hervorgehoben |
| Gleichartige | Module desselben Typs wie unter dem Mauszeiger werden hervorgehoben. |
| | Der Typ entspricht dabei der Klasse - alle Instanzen ein und derselben Klasse werden also hervorgehoben, sobald der Mauszeiger über einem Modul platziert wird. |
| Verbundene | Direkt mit dem Modul unter dem Mauszeiger verbundene Module werden hervorgehoben. |
| | Alle Module, die Daten an das Modul unter dem Mauszeiger senden, oder die Daten von diesem empfangen, werden hervorgehoben dargestellt. |
| Gleicher Remote Server | Falls das Modul unter dem Mauszeiger Remoting unterstützt, werden alle Module, die ebenfalls Remoting unterstützen und den gleichen Remoting Server konfiguriert haben, hervorgehoben. Genaueres zum Thema Remoting findet man in „Remoting“ in Kapitel 4, *Module*. |

Ordnet die Module entsprechend ihrer Verbindungen an. Damit kann der Anwender versuchen, einen unübersichtlichen Workspace übersichtlicher zu machen. Die Regeln zur automatischen Anordnung sind einfach: Module werden so geordnet, dass der Output-Slot oder Ausgang einer Verbindung links vom Input-Slot oder Eingang einer Verbindung positioniert wird. Damit werden die Module automatisch so angeordnet, dass die Daten von links nach rechts fließen. Kreise werden aufgespaltet, indem automatisch entsprechende HiddenConnection-Module eingefügt

Dieses Menü dient der Verwaltung verschiedener sogenannter Meta-Module. Darunter werden Softwarekomponenten zusammengefasst, die im Workspace genau so agieren wie Module, jedoch von der Implementierung völlig unterschiedlich dazu sind.

Meta-Module kapseln meist komplexe Funktionalitäten und müssen immer vor der Instantiierung parametriert werden. Näheres dazu ist in Kapitel 5, *Meta-Module* zu finden.

### Tabelle 2.8. Untermenü Meta-Module

| | |
|---|---|
| Parametermodul erzeugen | Erzeugt ein Modul zum Einschleusen beliebiger Daten in den Workspace. Die genaue Funktionsweise dieser Art von Modulen und die Szenarien, in denen ihr Einsatz vorteilhaft ist, wird in „Parametermodul“ in Kapitel 5, *Meta-Module* beschrieben. |
| Spaltmodul erzeugen | Erzeugt ein Modul zum Aufspalten beliebiger JavaBeans in ihre Properties. Die genaue Funktionsweise dieser Art von Modulen und die Szenarien, in denen ihr Einsatz vorteilhaft ist, wird in „Spaltmodul“ in Kapitel 5, *Meta-Module* beschrieben. |
| SVG-Dokument integrieren | Analysiert die Struktur eines SVG-Dokument auf steuerbare Elemente hin. Die genaue Funktionsweise dieser Art von Modulen und die Szenarien, in denen ihr Einsatz vorteilhaft ist, wird in „SVG-Dokument“ in Kapitel 5, *Meta-Module* beschrieben. |
| Gruppe anlegen | Erzeugt einen "Unter"-Workspace. Die genaue Funktionsweise dieser Art von Modulen und die Szenarien, in denen ihr Einsatz vorteilhaft ist, wird in „Gruppe“ in Kapitel 5, *Meta-Module* beschrieben. |
| Persistenzunterstützung | Fügt ein Module zum Persistenzmanagement ein (Speicherung und Berichtswesen). Die genaue Funktionsweise dieser Art von Modulen und die Szenarien, in denen ihr Einsatz vorteilhaft ist, wird in „Persistenzunterstützung“ in Kapitel 5, *Meta-Module* beschrieben. |
| Aviator | Erstellen des Aviator-Designs. Die genaue Funktionsweise dieser Art von Modulen und die Szenarien, in denen ihr Einsatz vorteilhaft ist, wird in „Aviator“ in Kapitel 5, *Meta-Module* beschrieben. |
| Dynamische Formulare | Erzeugt ein Modul, das es erlaubt, interaktive GUIs einschließlich Programmlogik zu erstellen. Die genaue Funktionsweise dieser Art von Modulen und die Szenarien, in denen ihr Einsatz vorteilhaft ist, wird in „Interaktive Formulare“ in Kapitel 5, *Meta-Module* beschrieben. |
| Eingang hinzufügen | Erstellen eines Einganges, mittels dessen Gruppenworkspaces Daten von ihrem übergeordneten Workspace empfangen können. Gruppenworkspaces werden in „Gruppe“ in Kapitel 5, *Meta-Module* beschrieben.<br><br>Diese Action ist nur für Gruppenworkspaces wie in „Gruppe“ in Kapitel 5, *Meta-Module* beschrieben verfügbar. |
| Ausgang hinzufügen | Erstellen eines Ausganges, mittels dessen Gruppenworkspaces Daten an ihren übergeordneten Workspace senden können. Gruppenworkspaces werden in „Gruppe“ in Kapitel 5, *Meta-Module* beschrieben.<br><br>Diese Action ist nur für Gruppenworkspaces wie in „Gruppe“ in Kapitel 5, *Meta-Module* beschrieben verfügbar. |

Die Actions in diesem Untermenü dienen der Anzeige verschiedener Navigationshilfen für besonders große oder komplexe Workspaces.

### Tabelle 2.9. Untermenü Navigation

Zeigt eine verkleinerte Ansicht der Arbeitsfläche zur leichteren Navigation an.

Ein Fenster wird geöffnet, das eine Miniaturansicht des aktuell bearbeiteten Workspaces darstellt. Ein farbiges Rechteck zeigt dabei die aktuell sichtbaren Teil des Workspaces. Mittels der Maus kann man dieses Rechteck verschieben und auf diese Art und Weise den sichtbaren Bereich oder Viewport sehr schnell ändern.

Module anhand ihrer Namen auffinden.

Diese Action öffnet ein Fenster mit einer Liste, die alle Modultitel des aktuell bearbeiteten Workspaces anzeigt. Mittels Klick auf einen der Namen wird der sichtbare Bereich oder Viewport so verschoben, dass dieses Modul sichtbar wird.

Untermenü zum Umschalten der Sichtbarkeit einzelner Ebenen.

Module können - vergleichbar einem Graphikprogramm - verschiedenen Ebenen angehören. Damit kann man zum Beispiel einen Workspace logisch gliedern. Dieses Untermenü enthält für jede Ebene einen Menüpunkt, der es erlaubt, die jeweils zugehörige Ebene sichtbar oder unsichtbar zu machen.

Werden Module unsichtbar, von denen Verbindungen ausgehen oder zu denen Verbindungen hinführen, so werden diese Verbindungen ebenfalls unsichtbar. Werden solche Module wieder sichtbar gemacht, werden die zugehörigen Verbindungen ebenfalls automatisch wieder sichtbar.

Der erste Menüpunkt in diesem Untermenü öffnet einen Dialog zur verwaltung der Ebenen:

### Tabelle 2.10. Untermenü für Ebenen

Öffnet einen Dialog zur Steuerung der Sichtbarkeit und zur Kombination von Ebenen. Kombinierte Ebenen dienen dazu, die Sichtbarkeit mehrerer zusammengehöriger Ebenen auf einmal zu ändern:

#### Tabelle 2.11. Aktionen im Dialog zur Verwaltung der Ebenen

Macht alle ausgewählten Ebenen unsichtbar - bereits unsichtbare Ebenen werden nicht geändert. Diese Aktion ist nur dann verfügbar, wenn in der Liste unterhalb der Werkzeugleiste mindestens eine Ebene selektiert ist.

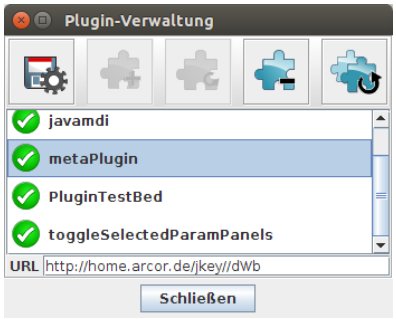
Macht alle ausgewählten Ebenen sichtbar - bereits sichtbare Ebenen werden nicht geändert. Diese Aktion ist nur dann verfügbar, wenn in der Liste unterhalb der Werkzeugleiste mindestens eine Ebene selektiert ist.

Erstellt eine Kombination aus allen ausgewählten Ebenen. Diese Aktion ist nur dann verfügbar, wenn in der Liste unterhalb der Werkzeugleiste mehr als eine Ebene selektiert ist.

Dieses Untermenü enthält immer einen Menüpunkt. Dieser dient dazu, einen Dialog zur Verwaltung der Plugins zu öffnen. Daran schließen sich verschiedene Untermenüs und Menüpunkte an. Deren Aufgaben und Anordnung wird durch die vom Anwender installierten Plugins bestimmt.

### Tabelle 2.12. Untermenü Plugins

### Abbildung 2.1. Dialog zur Verwaltung von Plugins




Öffnet einen Dialog zur Verwaltung von Plugins wie er in Abbildung Abbildung 2.1, „Dialog zur Verwaltung von Plugins“ examplarisch dargestellt ist. In diesem Dialog sind Aktionen zur verwaltung von Plugins in einer Werkzeugleiste am oberen Fensterrand zusammengefasst. Am unteren Fensterrand befindet sich ein Texteingabefeld, das eine URL zu einem Plugin-Repository enthält. Den Hauptteil des Fensters nimmt eine Liste ein, in der alle Plugins aufgeführt werden, die installiert sind, und/oder die das aktuelle Repository zur Installationoder zur Aktualisierung anbietet. Der Status eines Plugins wird durch folgende Piktogramme veranschaulicht:

### Tabelle 2.13. Status von Plugins

Nicht installiert.

Installiert.

Installiert und Update verfügbar.

In diesem Dialog stehen folgende Aktionen zur Verfügung:

### Tabelle 2.14. Aktionen zur Verwaltung von Plugins

Öffnet einen Dialog zur Konfiguration eines Proxy.

Installieren des ausgewählten Plugins. Diese Aktion ist nur verfügbar, wenn mindestens ein Plugin in der Liste selektiert wurde.

Aktualisiert das ausgewählte Plugin wenn eine neuere Version verfügbar ist. Diese Aktion ist nur verfügbar, wenn mindestens ein Plugin in der Liste selektiert wurde.

Entfernt das ausgewählte Plugin. Diese Aktion ist nur verfügbar, wenn mindestens ein Plugin in der Liste selektiert wurde.

Alle Plugins, für die neuere Versionen verfügbar sind, werden aktualisiert. Die Ausführung kann - je nach Anzahl zur Verfügung stehender Plugin-Aktualisierungen - einige Zeit in Anspruch nehmen.

Konfigurieren des gewählten Plugins. Diese Aktion ist nur dann verfügbar, wenn das gewählte Plugin installiert ist und über eine Aktion zur Anpassung der Konfiguration verfügt.

Zeigt die prozentuale Auslastung für jedes Modul mit eigenem Thread an.

Diese Action öffnet einen Dialog, der alle Modulinstanzen, die von ThreadingModuleBase abgeleitet sind, auf dem Workspace in einer Tabelle darstellt, die in jeder Zeile den Modulnamen und daneben die Auslastung des Moduls anzeigt.

Verwaltung der Remoting Server. Genaueres zum Thema Remoting findet man in „Remoting“ in Kapitel 4, *Module*. Die Remoting Server werden über einen Dialog verwaltet, der sich über diese Action öffnen läßt.

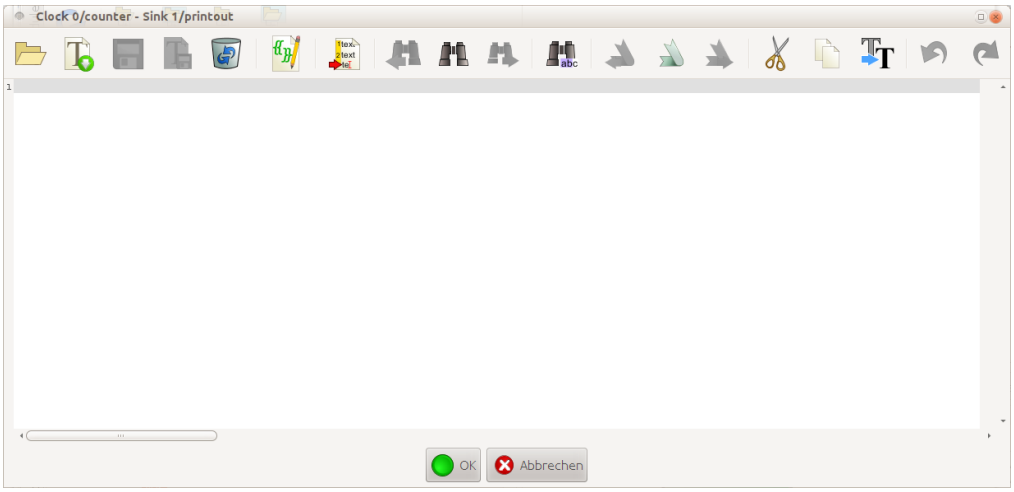
Der Dialog enthält eine Liste der aktuell verfügbaren Remoting Server. Oberhalb der Liste befindet sich eine Werkzeugleiste mit Actions zum Hinzufügen und Entfernen von Servern:

### Tabelle 2.15. Actions zur Verwaltung von Remoting Servern

Hinzufügen eines Elements zu der Liste

Entfernt die ausgewählten Elemente aus der Liste. Diese Action steht nur zur Verfügung, wenn mindestens ein Element selektiert wurde. Nach Audsführen der Action erscheint eine Sicherheitsabfrage. Elemente werden erst dann wirklich entfernt, wenn diese Sicherheitsabfrage nochmals bestätigt wird.

### Abbildung 2.2. Dialog zur Bearbeitung des Skripts an einer Verbindung



# Actions im Editor von Skripts für Verbindungen

**Tabelle 2.16. Actions im Editor zum Bearbeiten von Skripts für Verbindungen**

| | |
|---|---|
| | Ersetzt eingegebenen Text mit dem Inhalt einer Textdatei |
| | Hängt den Inhalt einer Textdatei an den bereits eingegebenen Text an |
| | Speichert den Inhalt der Datei |
| | Speichert den eingegebenen Text in einer Datei |
| | Löscht sämtlichen bereits eingegebenen Text |
| | Öffnet einen Dialog zur Verwaltung der Code-Templates, wie er beispielhaft in Abbildung 5.3, „Dialog zur Bearbeitung von Code-Templates“ dargestellt ist. Code-Templates sind Textfragmente mit einem Namen, die auf Anforderung - Voreinstellung ist **Ctrl**+**F12** - in den Text eingefügt werden. Dabei wird das Wort unter dem Cursor als Name eines einzufügenden Code-Templates interpretiert. Wird ein Code-Template diesen Namens gefunden, wird das Wort unter dem Cursor durch den Text des gefundenen Code-Templates ersetzt.<br><br>Namen für Code-Templates können aus Buchstaben und Zahlen bestehen und müssen mit einem Buchstaben beginnen. Enthalten Code-Templates Template-Parameter, kann der Anwender nach Einfügen des Templates mittels **Tabulator** zwischen den einzelnen Parametern umschalten - dabei springt der Cursor jeweils zur nächsten Position eines solchen Platzhalters und selektiert ihn. Template-Parameter werden durch eine grüne Unterstreichung hervorgehoben. **Esc** beendet diesen Modus. Template-Parameter können aus Buchstaben und Zahlen bestehen und müssen mit einem Buchstaben beginnen. Sie müssen mittels ${ und } eingeschlossen sein. |
| | Setzt den Cursor in die gewählte Zeile |
| | Vorheriges Auftreten finden |
| | Text im Dokument finden |
| | Nächstes Auftreten finden |
| | Hebt jedes Auftreten des aktuell markierten Textes hervor |
| | Setzt den Cursor auf die Zeile mit dem vorhergehenden Lesezeichen |
| | Setzt (löscht) ein Lesezeichen in der aktuellen Zeile |
| | Setzt den Cursor auf die Zeile mit dem nächsten Lesezeichen |
| | Schneidet den markierten Text aus und kopiert ihn in die Zwischenablage |
| | Kopiert den markierten Text in die Zwischenablage |
| | Fügt Text aus der Zwischenablage an der Cursorposition ein |
| | Undo |
| | Redo |

# Hilfslinien

Hilfslinien dienen der leichteren übersichtlichen Anordnung der Module. Es gibt sie in zwei unterschiedlichen Ausprägungen: Einmal existiert die Möglichkeit, ein Gitter mit fester "Maschenweite"

einzublenden. Weiterhin kann der Anwender einzelne Hilfslinien an beliebigen Positionen einfügen. Die Aktionen dazu, wie zum An- und Abschalten des Gitters finden sich im Kontextmenü.

Alle Arten von Hilfslinien wirken auf die gleiche Art und Weise: Module, deren Kanten beim Verschieben in die Nähe von Hilfslinien geraten, werden von diesen Hilfslinien eingefangen. Möchte man Module von Hilfslinien lösen, muß man bei gedrückter linker Maustaste die Maus eine gewisse Strecke verschieben, bevor die Hilfslinien, an denen das jeweilige Modul angedockt ist, das Modul wieder freigeben.

Frei positionierbare Hilfslinien können nachträglich verschoben werden: Wenn die Maus auf eine Hilfslinie wirkt, wird sie etwas dicker gezeichnet als der Rest. Ist eine Hilfslinie solcherart gekennzeichnet, kann man sie mittels der Maus verschieben, indem man die linke Maustaste gedrückt hält und die Maus bewegt. Befindet sich der Mauszeiger über dem Kreuzungspunkt einer vertikalen und einer horizontalen Hilfslinie, werden auf diese Art und Weise beide verschoben. Frei positionierbare Hilfslinien werden gelöscht, indem sie mit der Maus aus dem Workspace heraus gezogen werden.

Die frei positionierbaren Hilfslinien haben darüber hinaus noch zwei Besonderheiten: Zum einen ist es möglich, sie in der Länge beziehungsweise Breite zu ändern: das geschieht, indem man den Mauszeiger über der jeweiligen Hilfslinie positioniert und am Mausrad dreht: nach oben wird die Hilfslinie länger, nach unten wird sie kürzer. Die zweite Besonderheit ist der "Kehr"-Modus. Dieser Modus wird aktiv, wenn während des Verschiebens einer frei positionierbaren Hilfslinie die Taste **Ctrl** gedrückt und festgehalten wird: Dann wird diese Hilfslinie zu einem "Besen" und schiebt Module vor sich her. Damit ist es möglich, Module sehr schnell in Reih' und Glied zu bringen. Diesen Modus erkennt man daran, daß die aktive(n) Hilfslinie(n) dick gepunktet dargestellt werden.

# Tastaturbedienung

Die aktuell unterstützten Tastenkombinationen lauten:

| | |
|---|---|
| **Cursortaste Links** | Bewegen aller aktuell selektierter Module um 10 Pixel nach links |
| **Cursortaste Rechts** | Bewegen aller aktuell selektierter Module um 10 Pixel nach rechts |
| **Cursortaste Hoch** | Bewegen aller aktuell selektierter Module um 10 Pixel nach oben |
| **Cursortaste Runter** | Bewegen aller aktuell selektierter Module um 10 Pixel nach unten |
| **Umschalt+Cursortaste Links** | Bewegen aller aktuell selektierter Module um 50 Pixel nach links |
| **Umschalt+Cursortaste Rechts** | Bewegen aller aktuell selektierter Module um 50 Pixel nach rechts |
| **Umschalt+Cursortaste Hoch** | Bewegen aller aktuell selektierter Module um 50 Pixel nach oben |
| **Umschalt+Cursortaste Runter** | Bewegen aller aktuell selektierter Module um 50 Pixel nach unten |
| **Alt+Cursortaste Links** | Bewegen aller aktuell selektierter Module um 1 Pixel nach links |
| **Alt+Cursortaste Rechts** | Bewegen aller aktuell selektierter Module um 1 Pixel nach rechts |
| **Alt+Cursortaste Hoch** | Bewegen aller aktuell selektierter Module um 1 Pixel nach oben |
| **Alt+Cursortaste Runter** | Bewegen aller aktuell selektierter Module um 1 Pixel nach unten |
| **Taste Entfernen** | Löschen aller selektierten Module aus dem Workspace |
| **Strg+C** | Kopieren aller aktuell selektierter Module in die System-Zwischenablage |

| | |
|---|---|
| **Strg**+**X** | Ausschneiden aller aktuell selektierter Module und Kopieren in die System-Zwischenablage |
| **Strg**+**V** | Einfügen der in der System-Zwischenablage befindlichen Module |
| **Taste Enter** | Öffnen des Parameter-Dialogs |

# Kapitel 3. Dock

## Überblick

Die Komponente, die mit der Arbeitsoberfläche zusammen die Mitte des Hauptfensters einnimmt, nimmt Komponenten auf, die Module oder ganzen Workspace-Fragmente enthalten. Diese Komponenten stellen die Werkzeugpalette für einen Anwender zur Gestaltung eigener Workspaces dar.

Im Dock selbst sind mehrere Komponenten enthalten, die in den folgenden Abschnitten noch genauer beschrieben werden. Von den enthaltenen Komponenten ist immer nur eine sichtbar. Die Umschaltung zwischen den einzelnen Komponenten erfolgt durch Klick mit der linken Maustaste auf die Knöpfe am linken Rand des Docks. Klickt man auf den Knopf der aktuell gerade sichtbaren Komponente nochmals mit der linken Maustaste, wird das Dock ausgeblendet. Das erleichtert die Modulanordnung, da dadurch nochmals ein wenig mehr Platz auf der Arbeitsfläche zur Verfügung steht. Ist das Dock ausgeblendet, genügt ein Klick mit der linken Maustaste auf einen der den Komponenten zugeordneten Knöpfe um es wieder einzublenden.

Die Aufteilung, wieviel Platz das Dock und die Arbeitsfläche einnehmen kann auch angepasst werden, indem man den Trenner zwischen beiden Bereichen verschiebt. Dazu stellt man den Mauszeiger auf den Trenner (der Mauszeiger ändert sich), drückt die linke Maustaste und bewegt die Maus dann bei festgehaltener linker Maustaste.

## Module

### Aufgabe

Diese Komponente dient dem Management der Module, aus denen sich die Workspaces zusammensetzen. Die Module werden aus den installierten Modulquellen extrahiert und in einem Baum dargestellt. Die Module werden in einem Baum organisiert. Dabei sind die Modulquellen als oberste Ebene im Baum angeordnet. Darunter ordnen sich die Module anhand der Paketnamen.

### Bedienung

Module werden aus dem Baum per Drag'n'Drop auf den Workspace gezogen. So lange die Module im Baum dargestellt werden, werden keine Instanzen gebildet. Die Erzeugung der Instanzen geschieht erst, wenn ein Modul auf den Workspace fallen gelassen wird.

Der Baum verfügt über mehrere Actions in einer Werkzeugleiste, die über dem Baum zu finden ist. Unter dem Baum befindet sich ein Texteingabefeld, in das man beliebige Texte eingeben kann. Sobald das Texteingabefeld nicht mehr leer ist, werden im Baum nur noch Module angezeigt, die in ihrem Namen den eingegebenen Text enthalten. Dabei wird nicht nur direkt im Modulnamen gesucht, sondern auch in den Paketnamen. Wird das Texteingabefeld geleert, werden wieder alle Module im Baum sichtbar.

Ist der erste Buchstabe in diesem Textfeld ein < oder ein >, so bezieht sich der eingegebene Text nicht auf den Namen des Moduls, sondern auf die Datentypen, die es als Inputs akzeptiert (<) oder an seinen Ausgängen liefert (>). Das bedeutet, daß nach einer Eingabe von <numb nur noch die Module im Baum angezeigt werden, die zum Beispiel den Typ java.lang.Number an einem ihrer Eingänge entgegennehmen. Zu beachten ist hierbei, daß Module, die ein int als Input erwarten, trotz der Tatsache, daß es sich dabei ebenfalls um numerische Daten handelt ebenfalls verborgen sind: Es geht beim Vergleich um den reinen Text der Typnamen, nicht um etwa vorhandene Oberklassenbezeichnungen oder ähnliches. Gleiches gilt für die Filterung nach Ausgangsdatentypen.

## Actions

**Tabelle 3.1. Actions im Modulbaum**

| | |
|---|---|
| | Durchsucht das Modulverzeichnis nach neu hinzugekommenen Modulen. Dabei werden alle Module neu geladen. Wurden währenddessen Änderungen an den Funktionalitäten bestehender Module vorgenommen, stehen diese geänderten Funktionalitäten nach dem Neu Laden zur Verfügung. Bestehende Klassen funktionieren allerdings wie gehabt - nur neu erstellte Modulinstanzen verfügen über die geänderten Funktionalitäten. |
| | Alle Knoten anzeigen |
| | Zeigt alle Knoten unter dem selektierten |
| | Ausblenden aller Knoten |
| | Ausblenden aller Knoten unterhalb des ausgewählten |

# Workspaces

## Aufgabe

Workspaces können an beliebigen Orten im Dateisystem gespeichert werden. Bei oft benötigten Workspaces existiert aber womöglich der Wunsch, diese schneller zugreifbar zu haben.

Dazu existiert ein spezielles Verzeichnis für jeden Nutzer, in dem Workspaces gespeichert werden. Die Workspaces in diesem Verzeichnis werden in dieser Liste verwaltet.

## Bedienung

Das Hinzufügen eines neuen Workspaces in die Liste geschieht über die entsprechende Action im Workspace-Menü. Der Anwender muss dazu lediglich einen Namen vergeben. Existiert bereits ein Workspace mit diesem Namen in der Liste, wird der Anwender gefragt, ob er den bereits vorhandenen überschreiben möchte.

Die Workspaces in der Liste werden wie einzelne Module im Modulbaum per Drag'n'Drop in den gerade bearbeiteten Workspace eingefügt. Sie verhalten sich also ähnlich wie Makromodule.

## Actions

**Tabelle 3.2. Actions in der Workspace-Liste**

| | |
|---|---|
| | Neu laden |

# Globale Variablen

## Aufgabe

Diese Liste dient der Verwaltung globaler Variablen. Für jede Verbindung lässt sich ein Skript definieren, das jedesmal ausgeführt wird, wenn das zugehörige Sendermodul Daten an die Empfänger versendet. In solchen Skripten ist beliebiger Java-Quelltext erlaubt. Das gerade versendete Datum ist in der Variable `_data_` innerhalb des Skripts verfügbar.

Möchte der Entwickler des aktuell bearbeiteten Workspaces das Datum ändern, muss er einfach dieser Variable einen neuen Wert zuweisen - also zum Beispiel

```
_data_ = _data_*2;
```

schreiben.

Ein Entwickler kann aber durchaus auch ein Skript schreiben, das aus dem Wert in _data_ mehrere Resultate berechnet. In diesem Fall sind die globalen Variablen äußerst hilfreich: die Anweisung

```
GlobalVariableManager.getSharedInstance().set("huhu",_data_*3);
```

etwa würde eine globale Variable definieren, die den dreifachen Zahlenwert des zuletzt in `_data_` gespeicherten Wertes vorhält.

Näheres dazu ist im Anwenderhandbuch dWb+ im Abschnitt „Kontextmenü“ [25] auf Seite 25 zu finden.

## Bedienung

Die Benutzung der globalen Variablen entspricht der Benutzung der Module im Modulbaum - sie werden per Drag'n'Drop auf den bearbeiteten Workspace gezogen und dort fallengelassen.

Dadurch entstehen Pseudomodule mit nur einem Output, der den Typ der zugehörigen Variable hat. Jede Änderung der Variable sorgt dafür, dass das Pseudomodul den entsprechenden Wert an alle angeschlossenen Module weitergibt.

# Favoriten

## Aufgabe

Diese Komponente dient der Erstellung einer Kollektion oft benötigter Module. Besonders bei mehreren Modulquellen kann der Modulbaum sehr schnell unübersichtlich werden. Daher wurde mit den Favoriten eine Möglichkeit geschaffen, auf oft benutzte Module schneller zugreifen zu können.

## Bedienung

Um ein Modul zu den Favoriten hinzuzufügen, muss das entsprechende Modul im Modulbaum angefasst und auf die Favoriten gezogen werden. Da die beiden Komponenten nie gleichzeitig sichtbar sind, wird das Modul auf den Knopf mit der Aufschrift Favoriten in der Dock-Leiste gezogen. Nun kann man es entweder auf dem Knopf fallen lassen oder ein wenig warten - dWb+ schaltet nach einer geringen Zeitspanne, in der ein Modul über dem Knopf schwebt, auf die Komponente Favoriten um und man kann das Modul dann einfach direkt auf der Liste fallenlassen.

Die Benutzung der Favoriten entspricht der Benutzung der Module im Modulbaum - sie werden per Drag'n'Drop auf den bearbeiteten Workspace gezogen und dort fallengelassen. Erst beim Fallenlassen werden Instanzen der jeweiligen Klasse gebildet.

## Actions

**Tabelle 3.3. Actions für die Verwaltung der Favoriten**

| | |
|---|---|
| | Entfernt die ausgewählten Elemente aus der Liste |

# Scratch Manager

## Aufgabe

Diese Komponente dient als Sammelbecken für oft benötigte Modulkonfigurationen und Workspacefragmente, die öfter nachgenutzt werden sollen. Generell erfüllt sie denselben Zweck wie die Komponente Workspaces - allerdings ist die Trennung eine logische: die Komponente Workspaces sollte man benutzen, wenn fertige Workspaces schnell zugreifbar sein sollen.

Diese Komponente sollte eher dazu dienen , einen Baukasten von halb- oder teilweise fertiggestellten Bausteinen für Workspaces zu verwalten.

In dieser Komponente kann man natürlich auch einzelne Module verwalten - letztlich ist ein einzelnes Modul ja nur ein Spezialfall eines Workspacefragments. Das bietet den Vorteil gegenüber der Komponente namens Favoriten, dass man dort die Module nur in ihrer Standardkonfiguration hinzufügen kann. Der Scratch Manager hingegen erlaubt es, ein Modul, das durch den Anwender umkonfiguriert wurde, in genau dieser Konfiguration zur späteren Verwendung zu hinterlegen.

## Bedienung

Neue Workspacefragmente werden zur Liste hinzugefügt, indem die erste der Actions in der Werkzeugleiste über der Liste ausgeführt wird. Diese Action prüft, ob der gegenwärtige Inhalt der Zwischenablage ein gültiges Workspacefragment beschreibt. Falls ja, wird ein Dialog geöffnet, in dem man diesem Fragment einen Namen geben kann. Darüber hinaus ist es möglich, hier eine Beschreibung einzugeben. dWb+ erzeugt eine Beschreibung, die aus einer Aufzählungsliste mit den in dem Fragment enthaltenen Modulen besteht. Man kann diese Beschreibung durch beliebige eigene Texte ersetzen. Diese Angaben können auch später über eine entsprechende Action in der Werkzeugleiste angepasst werden.

Die Benutzung der Favoriten entspricht der Benutzung der Module im Modulbaum - sie werden per Drag'n'Drop auf den bearbeiteten Workspace gezogen und dort fallengelassen

## Actions

**Tabelle 3.4. Actions im Scratch-Manager**

| | |
|---|---|
| | Hinzufügen eines Elements zu der Liste |
| | Erlaubt die Details und den Namen dieses Bausteins zu ändern |
| | Entfernt die ausgewählten Elemente aus der Liste |
| | Bewegt das ausgewählte Element in der Liste eine Position nach unten |
| | Bewegt das ausgewählte Element in der Liste eine Position nach oben |

# Skript-Module

## Aufgabe

Diese Komponente dient der Verwaltung sogenannter Skripted Module. Diese Module liegen als nicht kompilierte Java-Klassen im Quelltext in einem Unterverzeichnis des Datenverzeichnisses, das auch die Module, die Konfiguration und verschiedene Ressourcen enthält. Der Name des Unterverzeichnisses für die Skripted Module lautet `scripted`. Die genauen Inhalte dieses Verzeichnisses sind in Anhang A, *Verzeichnis-Layout* beschrieben.

Die Dateien sind in diesem Verzeichnis entsprechend der Namespaces der in ihnen definierten Java-Klassen anzuordnen. Die Datei, in der die Klasse Example im package de.exam definiert ist, muß also unter dem Namen Example.java im Unterverzeichnis de/exam gespeichert werden.

# Bedienung

Die Skripted Module werden genau so benutzt, wie auch normale kompilierte Module: Sie werden mittels Drag'n'Drop auf den Workspace gezogen. Dort können sie kopiert und wieder eingefügt werden. Ihre Konfiguration wird wie bei anderen Modulen dabei wie auch beim Speichern und wieder Laden eines Workspace nicht vergessen, sondern konserviert. Auch Refactoring in Gruppen ist möglich.

Zur schnelleren Arbeit bei der Entwicklung von neuen Modulen kann man Änderungen an Skripted Modulen vornehmen, ohne die Anwendung beenden zu müssen. Allerdings sind die Änderungen nur für Modulinstanzen realisiert, die nach der Änderung auf dem Workspace platziert wurden. Bereits bestehende Instanzen werden nicht aktualisiert. Beim Speichern und Kopieren in die Zwischenablage wird immer die jeweils aktuellste Klassendefinition benutzt: Auch wenn eine Instanz mittels einer älteren Version der Klasse erzeugt wurde, ist sie nach dem Einfügen oder Laden auf dem neuesten Stand.

# Actions

### Tabelle 3.5. Actions für Skript-Module

| | |
|---|---|
| | Durchsucht das Verzeichnis der Skripted Module nach neu hinzugekommenen Modulen und aktualisiert gegebenenfalls die Baumansicht. |
| | Alle Knoten anzeigen |
| | Zeigt alle Knoten unter dem selektierten |
| | Ausblenden aller Knoten |
| | Ausblenden aller Knoten unterhalb des ausgewählten |

# Kapitel 4. Module

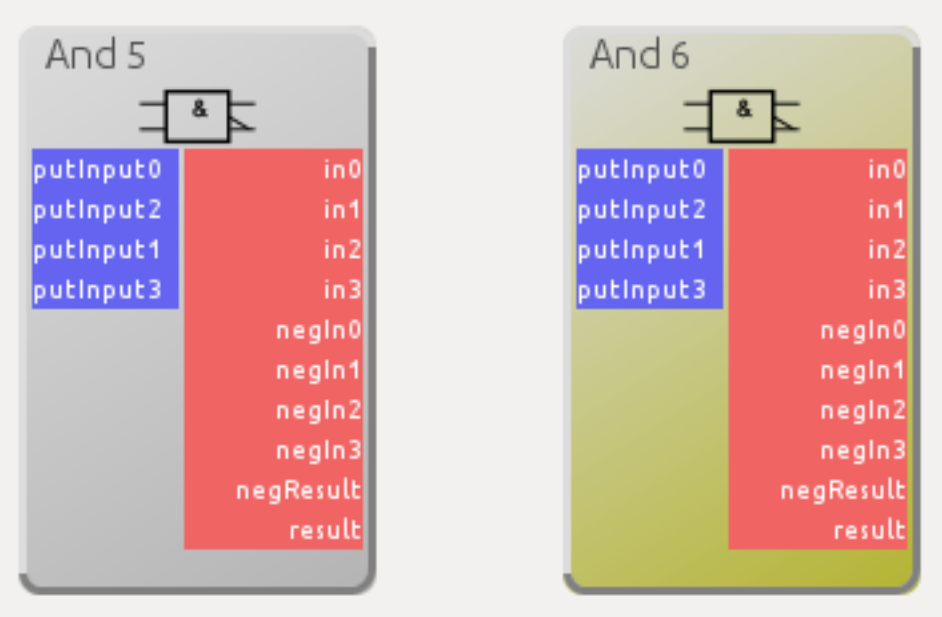
## Überblick

Module bilden die Verarbeitungseinheiten innerhalb der Anwendung dWb+. Verbindungen zwischen den Modulen leiten die vom Sender erzeugten Daten zum Empfänger, wo sie weiterverarbeitet werden.

Module erscheinen im Workspace als graphische Repräsentationen der durch das Modul verkörperten Algorithmen und Verarbeitungseinheiten. Diese benötigen Daten zur Verarbeitung und produzieren Ergebnisse. In Abbildung 4.1, „Aufbau von Modulen“ ist ein Beispiel für zwei Module auf dem Workspace zu sehen, das (fast) alle Elemente zeigt, die ein Modul aufweisen kann: Oben vefindet sich der Titel des Moduls, der durch den Anwender frei wählbar ist. Neben dem Titel können durch das Modul weitere Informationen eingeblendet werden. Dies geschieht über Symbole, die über einen Tooltip verfügen, so dass der Anwender mittels MouseOver die jeweilige Bedeutung erfragen kann. Auf der linken Seite in roter Farbe sind die Eingänge in das Modul in roter Farbe gekennzeichnet. Die Daten, die über diese Eingänge in das Modul fließen, stellen die Eingangsdaten für die Verarbeitung innerhalb des Moduls dar. Auf der rechten Seite in blauer Farbe findet man die Ausgänge des Moduls, über die die Ergebnisse der Verarbeitung das Modul verlassen. Das Modul selbst kann - über oder zwischen den Ein- und Ausgängen - noch ein Symbol aufweisen, das idealerweise die erbrachte Verarbeitungsleistung darstellt. Dort werden auch Symbole eingeblendet, wenn das Modul den Anwender über geringfügige (Warnungen ) oder schwerwiegende (Fehler ) Probleme informieren möchte. Der Rest des Moduls liefert Informationen darüber, ob es gerade selektiert ist (siehe unten) oder nicht: Selektierte Module werden gelb dargestellt wie in der Abbildung rechts zu sehen, Module, die nicht selektiert sind, werden dagegen wie in der Abbildung links grau dargestellt.

Die verschiedenen Abschnitte und Bereiche eines Moduls stellen darüber hinaus verschiedene Interaktionsmethoden bereit. Diese Interaktionsmethoden werden hier nur kurz benannt - die genauere Vorstellung erfolgt weiter unten in diesem Abschnitt. Der Korpus des Moduls ist der Bereich in der Abbildung 4.1, „Aufbau von Modulen“ - dieser hat abhängig von seinem Selektionsstatus eine andere Farbe: Standard ist grau für nicht selektiert und gelb für selektiert. Hier ist es möglich, die Position des Moduls anzupassen, indem die linke Maustaste gedrückt und festgehalten und die Maus anschließend bewegt wird. Ein Doppelklick in diesem Bereich öffnet den Parameterdialog für dieses Modul (falls es keinen Parameterdialog hat, bleibt der Doppelklick wirkungslos). Der Druck auf die rechte Maustaste öffnet ein Kontextmenü, in dem sich verschiedene nützliche Funktionen verbergen. Ein Klick mit der linken Maustaste selektiert das Modul, falls es noch nicht selektiert ist - falls es bereits selektiert ist, hebt ein Klick mit der linken Maustaste die Selektion auf, sofern gleichzeitig die Taste STRG festgehalten wird.

### Abbildung 4.1. Aufbau von Modulen




Die Module enthalten einen oder mehrere Eingänge, durch die sie Daten von anderen Modulen empfangen können und einen oder mehrere Ausgänge, durch die sie Daten an andere Module senden können. Diese Ein- und Ausgänge werden visuell durch die sogenannten Slots dargestellt: auf der linken Seite eines Modules sind die blau dargestellten Eingänge zu finden, auf der rechten Seite die rot dargestellten Ausgänge. Ein Modul hat immer einen Titel, der sich aus dem Namen und einer laufenden Nummer ergibt. Manche Module verfügen darüber hinaus noch über ein Icon, das idealerweise in Beziehung zur Aufgabe des Moduls steht.

Verbindungen zwischen Modulen werden angelegt, indem der Ausgang des Quell-Moduls mit dem Eingang des Zielmoduls verbunden wird. Das geschieht meist durch Drag'n'Drop - Klick der linken Maustaste während der Mauszeiger sich über einem Modulausgang befindet und anschließendes Ziehen der Maus bei festgehaltener linker Maustaste bis sich der Mauszeiger über dem gewünschten Eingang befindet. Wird die Maustaste losgelassen, wird die neue Verbindung angelegt. Darüber hinaus existieren weitere Wege zur Etablierung von Verbindungen, die weiter hinten in diesem Kapitel noch näher beleuchtet werden.

Bei der Verbindung mittels Drag'n'Drop lassen sich nicht beliebig Verbindungen anlegen: Der Typ des Eingangs muss zu dem des Ausganges passen. Diese Information wird über kleine Symbole neben dem Mauszeiger während des Vorgangs angezeigt. Prinzipiell passen Ausgänge zu Eingängen, wenn sie entweder den gleichen Typ haben oder wenn der Eingang einen Typ hat, der eine Generalisierung des Typs des Ausgangs darstellt. So kann etwa ein `Integer`-Ausgang mit einem `Number`-Eingang verbunden werden, da jede Ganzzahl auch eine Zahl ist - jedoch wäre umgekehrt die Verbindung eines `Number`-Ausgangs mit einem `Integer`-Eingang nicht möglich, da Zahlen auch Gleitkommazahlen sein können, der Eingang aber explizit nur Ganzzahlen (ohne Nachkommastellen) verlangt.

Seit Version `4.3pre1` ist es möglich, Ein- und Ausgänge, deren Typen eigentlich nicht zueinander passen dennoch zu verbinden: Hält man während des Drag'n'Drop die Taste **Strg** gedrückt, ist die im vorhergehenden Absatz beschriebene Validierung zueinander passender Datentypen von Ein- und Ausgängen außer Kraft gesetzt: Wenn nach Loslassen der Maustaste festgestellt wird, dass die beiden verbundenen Slots nicht zueinander passen, wird zwischen beiden automatisch ein Konvertermodul in die neue Verbindung eingefügt. Es öffnet sich sofort das Parameterfenster dieses Moduls, in dem der Anwender mittels BeanShell die Konversion des Datenobjekts aus dem Ausgang in ein

für den Eingang passendes Objekt spezifizieren kann. Dabei gelten die folgenden Konventionen: In dem BeanShell-Fragment sind zwei Variablen mit besonderen Bedeutungen reserviert: `_input_` hält den vom Quellmodul über den Ausgang versendeten Wert und der am Ende der Ausführung des BeanShell-Fragments in `_output_` gespeicherte Wert wird an den Eingang des Zielmoduls gesendet. Der `_input_` wird ungefiltert in das BeanShell-Fragment übergeben - daher ist es angeraten, `null` explizit zu behandeln! Um die Arbeit einfacher zu gestalten ist es möglich, für Paare von Datentypen Default-Konversionen zu hinterlegen, die dann automatisch im Editor des Parameterfensters als Vorbelegung auftauchen. Das ist in „conversionPresets.xml“ in Anhang A, *Verzeichnis-Layout* ausführlicher beschrieben.

Module können beliebig mit der Maus angeordnet und verschoben werden. Mehrere Module gleichzeitig zu bewegen ist möglich, wenn die betroffenen Module selektiert sind. Wenn eines der selektierten Module bewegt wird, werden alle selektierten Module diese Bewegung ebenfalls vollziehen. Weiterhin ist es möglich, die Menge der selektierten Module nach bestimmten Kriterien automatisch anzuordnen. So kann man zum Beispiel die Oberkanten aller selektierten Module auf eine Linie bringen oder den vertikalen oder horizontalen Zwischenraum zwischen allen selektierten Modulen ausgleichen.

Um mehrere Module zu selektieren kann man entweder, indem man mit der Maus durch Drücken der linken Maustaste und anschließendes Bewegen der Maus bei gehaltener linker Maustaste ein Viereck markiert - alle Module, die teilweise oder vollständig innerhalb des Rechtecks liegen, werden bei Loslassen der Maustaste selektiert. Selektierte Module erkennt man an der Färbung: Module sind grau, selektierte Module gelb. Eine weitere Möglichkeit, Module zu selektieren ist ein einfacher Klick mit der linken Maustaste während sich der Mauszeiger innerhalb eines Moduls befindet (nicht über einem der Aus- oder Eingänge). In diesem Fall werden alle anderen selektierten Module automatisch deselektiert. Hält man während des Klicks die Taste Steuerung (Strg oder Ctrl) gedrückt, wird das Modul, wenn es vorher nicht selektiert war, selektiert. Gehörte das Modul dagegen bereits zur Menge der selektierten Module, wird es in diesem Falle deselektiert.

# Modulmenü

### Tabelle 4.1. Kontextmenü für Module

Löschen von Modulen.

Diese Action löscht das Modul, für welches das Kontextmenü geöffnet wurde, beziehungsweise alle aktuell selektierten Module. Achtung: das Löschen wird ohne Sicherheitsabfrage direkt ausgeführt.

Weiterhin werden alle Verbindungen, die an den zu löschenden Modulen enden oder von ihnen ausgehen ebenfalls gelöscht.

Dieses Untermenü beinhaltet Actions zur Ausrichtung mehrerer Module nach verschiedensten Kriterien. Dieses Untermenü und die darin enthaltenen Actions haben nur dann Auswirkungen, wenn mindestens zwei Module selektiert sind.

Alle Actions beziehen sich jeweils nur auf die selektierten Module.

### Tabelle 4.2. Untermenü Ausrichtung

Richtet die Module so aus, dass die linken Kanten alle auf einer gedachten Linie liegen

Richtet die Module so aus, dass die rechten Kanten alle auf einer gedachten Linie liegen

| | |
|---|---|
| | Richtet die Module so aus, dass die oberen Kanten alle auf einer gedachten Linie liegen |
| | Richtet die Module so aus, dass die unteren Kanten alle auf einer gedachten Linie liegen |
| | Richtet die Module so aus, dass die horizontalen Mitten alle auf einer gedachten Linie liegen |
| | Richtet die Module so aus, dass die vertikalen Mitten alle auf einer gedachten Linie liegen |
| | Verteilt die Module so, dass der horizontale Zwischenraum zwischen je zwei benachbarten überall gleich ist |
| | Verteilt die Module so, dass der vertikale Zwischenraum zwischen je zwei benachbarten überall gleich ist |

Dieses Untermenü dient der Konfiguration der Sichtbarkeit unbeschalteter Ein- und Ausgänge.

Die Actions in diesem Menü beziehen sich nur auf das Modul, für das das Kontextmenü geöffnet wurde. Etwaige weitere selektierte Module werden davon nicht betroffen.

### Tabelle 4.3. Untermenü Sichtbarkeit

| | |
|---|---|
| | Öffnet einen Dialog zur Konfiguration, welche Ein- und Ausgänge sichtbar sein sollen. |
| | Es ist nicht möglich, Ein- und Ausgänge, die über Verbindungen mit anderen Modulen verbinden sind, auszublenden. Nur unbeschaltete Ein- und Ausgänge können ausgeblendet werden. |
| | Verbirgt alle unbenutzen Ein- und Ausgänge in diesem Modul. |
| | Diese Action entspricht der gleichartigen Action, die auch für den Workspace zur Verfügung steht. |
| | Blendet alle ausgeblendeten Slots dieses Moduls ein |

### Tabelle 4.4. Untermenü Refactoring

| | |
|---|---|
| | Verschiebt die ausgewählten Module in eine eigene Gruppe. |
| | Diese Action dient der Verbesserung der Übersichtlichkeit komplexer Workspaces: Man kann damit logische Funktionseinheiten definieren und daraus einen Unterworkspace erstellen. Damit werden beliebig viele Module zu einer Gruppe wie in „Gruppe“ in Kapitel 5, *Meta-Module* beschrieben zusammengefasst. |
| | Alle selektierten Module und alle Verbindungen, die zwischen jeweils zwei selektierten Modulen bestehen werden Teil der Gruppe. Verbindungen, die auf nur einer Seite in einem selektierten Modul enden, erstellen automatisch einen Ein- oder Ausgang für die Gruppe. Dieser Ausgang wird dann entsprechend außer- und innerhalb der Gruppe mit den entsprechenden Modulen so verbunden, dass der Workspace nach dem Refactoring wieder genauso funktioniert, wie davor. |

Verschiebt die ausgewählten Module in die ausgewählte Ebene.

Diese Action öffnet einen Dialog mit einer ComboBox, die die bereits definierten Ebenen enthält. Wählt man hier eine aus, so werden alle selektierten Module dieser Ebene zugeordnet. Ist diese ausgewählte Ebene zu diesem Zeitpunkt ausgeblendet, werden die neu zu dieser Ebene hinzugefügten ebenfalls sofort automatisch ausgeblendet.

Neue Ebenen werden erstellt, indem nicht einer der bereits vorhandenen Namen in der ComboBox ausgewählt wird, sondern ein Name, der noch nicht vergeben wurde in die ComboBox eingegeben wird. In diesem Fall wird eine neue Ebene mit diesem Namen angelegt und alle selektierten Module dieser neuen Ebene zugeschlagen. Neu angelegte Ebenen sind direkt nach der Erzeugung zunächst sichtbar.

Führt die selektierten Module in einer Gruppe zusammen.

Löst eine bestehende Gruppe wieder auf.

Kopiert die selektierten Module und die Links zwischen ihnen in die Zwischenablage.

Alle selektierten Module werden in die Zwischenablege kopiert. Dazu kommen noch alle Verbindungen, die sowohl von einem selektierten Modul ausgehen als auch an einem selektierten Modul enden.

Kopiert die selektierten Module und die Links zwischen ihnen in die Zwischenablage und löscht sie anschließend aus dem Workspace.

Alle selektierten Module werden in die Zwischenablege kopiert. Dazu kommen noch alle Verbindungen, die sowohl von einem selektierten Modul ausgehen als auch an einem selektierten Modul enden. Die Module und Verbindungen, auf die das oben Gesagte zutrifft, werden gelöscht.

### Tabelle 4.5. Untermenü Parameterdialoge

Schließt alle geöffneten Parameterdialoge außer dem aktiven.

Der aktive Parameterdialog ist derjenige des Moduls, für das das Kontextmenü geöffnet wurde - die Parameterdialoge anderer Module - auch wenn diese selektiert sind - werden geschlossen.

Schließt alle geöffneten Parameterdialoge.

Öffnet den Parameterdialog.

Schließt den Parameterdialog.

Sendet den Inhalt des Parameterdialogs zum zentralen Parameter-Manager. Damit lässt sich die Anzahl von Fenstern minimieren: Alle Parameterdialoge - beziehungsweise deren Inhalte - können innerhalb eines einzigen Fensters gruppiert werden. Der Nachteil dieser Methode ist, dass nur noch ein Parameterdialog gleichzeitig angezeigt werden kann.

Möchte man einen Parameterdialog wieder in sein eigenes Fenster legen, muss man ihn einfach nochmals öffnen: entweder durch

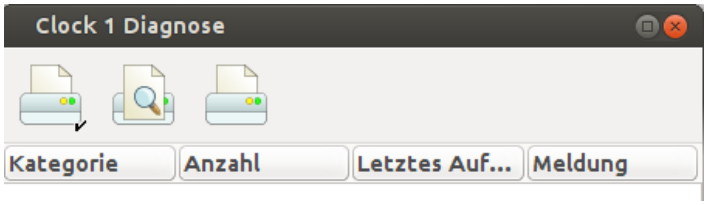
Doppelklick auf das Modul oder durch die entsprechende Action im Kontextmenü.

Wurde ein Parameterdialog in die Verantwortung des Parameter-Managers übergeben, wird dieser Fakt gespeichert, so daß nach Laden eines solchen Workspaces der Parameterdialog sofort wieder unter der Verwaltung durch den Parameter-Manager steht.

Dieses Untermenü dient dem Umgang mit wichtigen Meldungen aus dem Modul heraus. Module können Warnungen und Fehlermeldungen für den Anwender produzieren. Diese Meldungen werden durch das Modul so lange gespeichert, bis der Anwender selbst das Kommando gibt, diesen Speicher zu löschen. Sich wiederholende Meldungen werden nicht gespeichert, sondern nur gezählt. Jede Meldung wird sekundengenau mit dem Zeitpunkt ihres Auftretens protokolliert. Bei sich wiederholenden Meldungen wird nur der Zeitpunkt des letzten Auftretens zusammen mit der Gesamtanzahl protokolliert.

### Tabelle 4.6. Untermenü Diagnose

Zeigt alle Warnungen und Fehler dieses Moduls seit dem der Anwender das letzte Mal den Fehlerspeicher geleert hat (oder seit Programmstart, falls der Fehlerspeicher seither noch nie geleert wurde). In dem sich öffnenden Dialog befindet sich die tabellarische Darstellung aller Meldungen mit darüber befindlichen Aktionen zum Reporting:




Die Aktionen dienen zur Steuerung des Reportings: Reports können in verschiedenen Formaten erzeugt und ihre Erscheinung kann angepasst werden. Die dazu zur Verfügung stehenden Aktionen sind im Einzelnen:

### Tabelle 4.7. Untermenü Reporting (Diagnose)

Öffnen des Konfigurationsdialoges

Vorschau des Berichtes am Bildschirm

Speichern des erzeugten Berichtes in einer Datei

Verwirft die gespeicherte Liste der Fehler und Warnungen

Kopiert die eindeutige ID dieses Moduls in die Zwischenablage

Dieses Untermenü dient der Verwaltung der JMX-Funktionalitäten in dWb +. Module, deren zugrundeliegende Beans MBeans im Sinne von JMX sind, erhalten zusätzlich dieses Untermenü. Module, auf die das nicht zutrifft, erhalten dieses Menü nicht. MBeans werden durch dWb+ integriert, indem sie einfach über das Kontextmenü im MBean Server publiziert werden können. Diese Veröffentlichung kann ebenfalls über dieses Kontextmenü jederzeit rückgängig

gemacht werden. Nach der Veröffentlichung verhalten sie sich wie jede andere MBean auch - über einen entsprechenden Agenten, der mit dem MBean Server verbunden ist, können die exponierten Eigenschaften abgefragt und geändert, sowie Notifications empfangen werden. Der Publikationsstatus der MBean wird beim Speichern des Workspace genau so mit übernommen, wie beim Kopieren über die Zwischenablage: Nach erneutem Einladen eines Moduls, das im MBean Server publiziert war, wird es sofort wieder publiziert. Die Kopie eines Moduls, das im MBean-Server publiziert war, wird nach Einfügen sofort ebenfalls publiziert.

### Tabelle 4.8. Untermenü JMX

| | |
|---|---|
| | Veröffenlicht die diesem Modul zugrundeliegende Bean über den Platform Server |
| | Beendet die Veröffentlichung der diesem Modul zugrundeliegende Bean |
| | Öffnet einen Dialog zur Eingabe eines dedizierten Object Name statt des automatisch generierten. Der Object Name wird dabei als Eigenschaft der betreffenden Modulinstanz („Eigenschaften“) unter dem Namen de.netsysit.dataflowframework.ui.ModuleWidget.ObjectName gespeichert. |
| | Kopiert den Objektnamen, unter dem die diesem Modul zugrundeliegende Bean als MBean publiziert wurde, in die Zwischenablage |
| Modul aktiv | Umschalten des Status aller angeschlossenen Verbindungen zwischen inaktiv/aktiv. Visuell ist das an der Darstellung der Verbindung zu erkennen: aktibe Verbindungen werden mittels durchgezogener Linien dargestellt, inaktive mittels gepunkteter Linien. |
| | Diese Aktion ist nicht verfügbar, wenn mehr als ein Modul selektiert ist. Die Aktion wirkt nur auf das Modul, für das das Kontextmenü geöffnet wurde. |
| | Erlaubt es, die Farbe sämtlicher aktueller Verbindungen dieses Moduls zu ändern. Diese Aktion öffnet einen Dialog, in dem der Anwender eine Farbe auswählen kann. Wird der Dialog bestätigt, werden anschließend alle Verbindungen, die von diesem Modul ausgehen oder an diesem Modul enden, mit der neuen Farbe eingefärbt. |
| | Diese Aktion ist immer verfügbar. Sie wirkt nur auf das Modul, für das das Kontextmenü geöffnet wurde. |
| | Wurden über das Kontextmenü des Workspace Regeln zum Hervorheben von Modulen ausgewählt, wird diese Aktion für den Anwender auswählbar. |
| | Diese Aktion ist bewirkt, daß alle vorher selektierten Module deselektiert werden. Im Anschluß daran werden alle Module selektiert, die zum Zeitpunkt der Ausführung der Aktion hervorgehoben dargestellt wurden. Näheres zum Hervorheben von Modulen findet sich in „Kontextmenü“ in Kapitel 2, *Workspace* |
| | Öffnet einen Dialog zur Eingabe des neuen Namens |
| | Dieses Untermenü bietet die Möglichkeit, jedem Modul eines der verfügbaren Embleme hinzuzufügen. Das Emblem wird jeweils links über dem zugehörigen Modul angezeigt. Die Aktionen dieses Untermenüs sind nur verfügbar, wenn genau ein Modul selektiert ist. |
| | Die einzelnen zur Verfügung stehenden Embleme oder Symbole können durch den Anweder konfiguriert werden: Dazu ist einfach ein Verzeichnis namens `moduleEmblems` im Konfigurationsverzeichnis der Anwendung anzulegen |

und die entsprechenden Graphiken dort hineinzukopieren (vergleiche dazu „moduleEmblems“). Zum Konfigurationsverzeichnis der Anwendung vergleiche bitte Anhang A, *Verzeichnis-Layout* im Anwenderhandbuch dWb+.

Die Bemerkung kann beliebigen Text enthalten

Erzeugt ein Kind-Modul zum Anpassen der Eigenschaften dieser Instanz. Näheres zu Eigenschaften unter „Eigenschaften“. Das Kind-Modul enthält eine Tabelle, die pro Zeile eine Eigenschaft anzeigt. Dabei steht der Name der Eigenschaft links, ihr Wert rechts. Beide Aspekte einer Eigenschaft können in-place modifiziert werden: Dazu ist lediglich ein Doppelklick auf die jeweilige Tabellenzelle notwendig. Zum Hinzufügen und Entfernen von Eigenschaften existiert ein Kontextmenü, das per Mausklick mit der rechten Maustaste erreicht werden kann:

### Tabelle 4.9. Untermenü Modul-Eigenschaften

Fügt eine neue Eigenschaft hinzu

Entfernt die Eigenschaft unter dem Mauszeiger

Konfiguriert Log-Level und Ziel für die protokollierten Meldungen

Threads Ist ein Modul so konzipiert, daß die eigentliche Bearbeitung der Funktionalität in einem eigenen Thread passiert, verfügt das zugehörige Widget über ein Steuerungsmenü:

### Tabelle 4.10. Steuerung des Threads eines Moduls

Dieses Untermenü erlaubt die Konfiguration der Prorität des zugehörigen Threads

Startet die Funktionalität dieses Moduls

Stoppt die Ausführung dieses Moduls

Dieses Untermenü existiert nur bei Modulen, deren Funktionalität in einem eigenen Thread ausgelagert ist und dessen Pufferstrategie konfigurierbar ist. Die Pufferstrategie bestimmt darüber, wie sich das Modul verhalten soll, wenn die Abarbeitung des letzten Eingangsdatums noch nicht abgeschlossen ist und bereits das nächste Eingangsdatum eintrifft.

### Tabelle 4.11. Pufferstrategie für eingehende Daten

| Pufferstrategie für eingehende Daten | |
|---|---|
| Blockierend | Falls ein neues Datum am Eingang eingeht, bevor die Bearbeitung des letzten abgeschlossen ist, wird das sendende Modul so lange blockiert, bis das empfangende für die Abarbeitung wieder zur Verfügung steht. |
| Nicht Blockierend | Falls ein neues Datum am Eingang eingeht, bevor die Bearbeitung des letzten abgeschlossen ist, wird das neue Datum verworfen. |
| Gepuffert | Falls ein neues Datum am Eingang eingeht, bevor die Bearbeitung des letzten abgeschlossen ist, wird das neue Datum an das Ende einer Warteschlange angereiht. Sobald das empfangende Modul für die Abarbeitung wieder zur Verfügung steht, fährt es mit der Bearbeitung des ersten Elements der Warteschlange fort, sofern diese nicht leer ist. |

Modulmenü Verfügt ein Modul zusätzlich über eigene Actions, so werden diese in ein Untermenü integriert, das an dieser Stelle eingefügt wird. Das Untermenü trägt dabei den konkreten Modulnamen.

Module, die Remoting unterstützen, zeigen darüber hinaus hier ein Untermenü namens Remoting Server, das die verschiedenen Remoting Server zur Auswahl stellt. Genaueres zum Thema Remoting findet man in „Remoting“

# Inputs

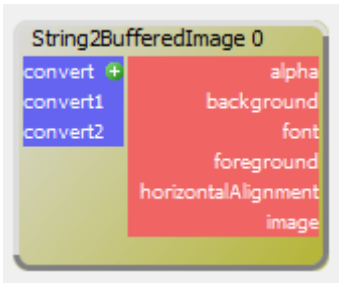
## Doppelklick auf Inputs

Klickt man auf einen Input-Slot mit der linken Maustaste doppelt, so wird die Erzeugung eines in „Parametermodul“ in Kapitel 5, *Meta-Module* beschriebenen Meta-Modules initiiert.

## Module mit variabler Anzahl von Inputs

Es existieren Module, bei denen die Anzahl von Input-Slots nicht festgelegt ist. Ein einfaches Beispiel für ein solches Modul wäre etwa ein Modul, das die Summe der Zahlen bilden soll, die es an seinen Eingängen empfängt. Es wäre unflexibel, die Menge der zu summierenden Zahlen auf 2 oder irgendeine andere Zahl festzulegen. Statt dessen wird eine clevere Implementierung es erlauben, weitere Eingänge hinzuzufügen, so sie benötigt werden. Solche Eingänge, die durch den Anwender vervielfältigt werden können, sind durch ein kleines grünes Plus am Input-Slot erkennbar. Ein Beispiel dafür, wie ein solches Modul auf dem Workspace aussieht, sieht man in Abbildung 4.2, „Beispiel für ein Modul mit variabler Anzahl von Eingängen“. Ein Klick mit der linken Maustaste während der Mauszeiger über diesem grünen Plus-Symbol positioniert ist, erzeugt einen weiteren Eingang des jeweiligen Typs. In der Abbilddung wurde der Eingang bereits zweimal vervielfältigt.

**Abbildung 4.2. Beispiel für ein Modul mit variabler Anzahl von Eingängen**




# Outputs

## Menü für Outputs

Dieses Kontextmenü enthält über den in der nachfolgenden Tabelle beschriebenen Punkten abhängig vom Inhalt des Workspaces weitere Untermenüs und Menüpunkt: Bei Öffnen des Kontextmenüs wird der Workspace daraufhin untersucht, ob sich darin Module befinden, die einen Input aufweisen, mit dem man den Output verbinden könnte, für den das Kontextmenü geöffnet wurde.

Werden solche Module gefunden, gibt es zwei Möglichkeiten des Verfahrens: Enthält ein Modul genau einen passenden Eingang, wird ein Menüpunkt mit dem Namen des Moduls angelegt: dieser Menüpunkt sorgt dann für die Einrichtung einer Entsprechenden Verbindung. Existieren mehr als ein passender

Eingang, wird ein Untermenü mit dem Namen des Moduls angelegt, das als Menüpunkte die Namen der Kandidaten enthalten. Jeder dieser Untermenüpunkte richtet dann die jeweilige Verbindung ein.

**Tabelle 4.12. Kontextmenü für Output-Slots**

| | |
|---|---|
| | Öffnet einen Dialog zur Bearbeitung von Name und Beschreibung |
| Kandidaten | Dieses Untermenü wird abhängig vom Typ des Outputs aufgebaut, für den das Kontextmenü geöffnet wurde. Es enthält alle Module, die über mindestens einen Eingang verfügen, der mit diesem Output verbunden werden könnte. Dabei sind die Module in Untermenüs gruppiert, die den Typen entsprechen, die auf den ausgewählten Output passen würden. Ist der Output vom Typ Number, würden also gegebenenfalls Untermenüs vom Typ int, double,... entstehen.<br><br>Bei Auswahl eines dieser Menüpunkte wird das entsprechende Modul auf den Workspace gelegt. Existiert nur ein passender Input in dem neuen Modul, wird auch die entsprechende Verbindung automatisch eingerichtet. |

# Doppelklick auf Outputs

Klickt man mit der linken Maustaste auf einen Output-Slot, wird automatisch eines der in „Spaltmodul“ in Kapitel 5, *Meta-Module* beschriebenen Meta-Module erzeugt und mit diesem Output-Slot verbunden.

# Tooltips für Outputs

Die Output-Slots der einzelnen Module tragen zunächst den Typ der Daten, die sie versenden wollen als Tooltip, wenn nicht in der zugehörigen BeanInfo-Klasse eine ShortDescription für die entsprechende Property vereinbart ist. Diese Informationen ändern sich, sobald das erste Datum aus einem Slot versendet wird, falls sogenannte StateUpdater für den Typ des Output-Slots vereinbart sind.

StateUpdaters sind Komponenten, denen in der Anwendung dWb+ die Aufgabe zukommt, die Verfolgung der Datenverarbeitung zu erleichtern: Sie ersetzen bei bekannten Datentypen die Tooltips der Slots durch Komponenten, die jeweils die zuletzt von diesem Slot versendeten Daten visualisieren. Dazu muss für den Slot-Typ eine StateUpdater-Komponente registriert sein. Beispiele für bereits registrierte Datentypen sind unter anderem:

| | |
|---|---|
| String | Komponente zeigt eine Historie der letzten versendeten Strings aus diesem Slot |
| Number | Komponente zeigt die jeweils letzte aus diesem Slot versendete Zahl als String an |
| java.awt.Image | Komponente zeigt eine Miniaturansicht des letzten aus diesem Slot versendeten Bildes an |

Diese Tooltips haben einige Eigenschaften, die über die normaler Tooltips hinausgehen: man kann sie durch Setzen des Häkchens in der Checkbox oben rechts an den Workspace heften, so dass sie nicht verschwinden, wenn der Mauszeiger den zugehörigen Slot wieder verläßt. Der Tootip verschwindet nicht, wenn der Mauszeiger zwar die zugehörige Komponente verläßt, aber über dem Tooltip verbleibt. Man kann den Tooltip mittels Ziehen mit der Maus vergrößern und verkleinern (besonders nützlich bei der Vorschau von Bildern).

Man kann eigene StateUpdater für beliebige Klassen erstellen und diese dann einfach in der Anwendung benutzen. Dazu reicht es aus, eine Jar-Datei an die richtige Stelle des Konfigurationsverzeichnisses zu kopieren. Die Organisation der Dateien und Verzeichnisse des Konfigurationsverzeichnisses ist im Anhang A, *Verzeichnis-Layout* zu finden. Damit ist es auch möglich, bereits mitgelieferte StateUpdater zu überschreiben - man könnte also zum Beispiel die Darstellung von Zahlen als Text durch eine Komponente ersetzen, die beispielsweise die letzten zwanzig Werte als Diagramm darstellt.

# Automatisches Erstellen von Verbindungen

Es ist möglich, Verbindungen zwischen Modulen automatisch erstellen zu lassen: Manche Module sind durch Embleme oder Symbole in der rechten oberen Ecke gekennzeichnet, die anzeigen, dass sie über Ein- oder Ausgänge verfügen, die automatisch Verbindungen zu denen anderer Module aufbauen. Diese Kennzeichnungen sehen wie folgt aus:

**Tabelle 4.13. Kennzeichnungen für Module, die automatisch Verbindungen erstellen können**

| | |
|---|---|
| | Dieses Modul verfügt über mindestens einen Eingang, der sich automatisch mit geeigneten Ausgängen verbindet. |
| | Dieses Modul verfügt über mindestens einen Ausgang, der sich automatisch mit geeigneten Eingängen verbindet. |
| | Dieses Modul verfügt über mindestens einen Ausgang und mindestens einen Eingang, die sich automatisch mit geeigneten Gegenstücken verbinden. |

Die tatsächliche Etablierung solcher Verbindungen geschieht, indem die jeweiligen Module so bewegt werden, dass sich die Kanten berühren: wird ein Modul, das über mindestens einen Eingang verfügt, über den automatische Verbindungen etabliert werden können, so bewegt, dass seine rechte Kante die linke eines Moduls berührt, das über mindestens einen Ausgang verfügt, über den automatische Verbindungen etabliert werden können, wird zwischen diesen automatisch eine Verbindung etabliert (falls dies nicht bereits geschehen ist).

# Diagnose

Zwischen Inputs und Outputs befindet sich bei jedem Modul ein Bereich, der eigentlich für die Anzeige des in der zugehörigen BeanInfo-Klasse festgelegten Symbols reserviert ist. Ist kein Symbol festgelegt, bleibt dieser Bereich zunächst leer.

Module, die dafür ausgelegt sind, können den Anwender über Probleme informieren, indem sie in diesem Bereich bei Problemen oder kritischen Fehlern entsprechende Symbole einblenden. Verfügt das Modul über eine BeanInfo-Graphik, wird das entsprechende Symbol darübergeblendet:

Solange im Untersuchungszeitraum nur Probleme und keine kritischen Fehler gemeldet wurde, wird das Symbol für Warnungen (links, gelb) angezeigt, sobald mindestens ein kritischer Fehler aufgetreten ist, wird das Symbol für Fehler angezeigt. Dieses bleibt bestehen, auch wenn danach wieder neue Warnungen aufgelaufen sind. Die letzte Meldung wird als Tooltip am diesem Emblem dargestellt. Auch hier ist es so, daß die Meldung zum letzten kritischen Fehler stehen bleibt, auch wenn inzwischen wieder neue Warnungen generiert wurden.

Im Kontextmenü jedes Moduls findet man Aktionen, die es erlauben, genauere Informationen über die Meldungen zu erhalten - unter anderem zur Anzahl des Auftretens. Weiterhin kann man dort Berichte über die aufgelaufenen Ereignisse exportieren und den Speicher für die Meldungen leeren.

# Remoting

## Überblick

Remoting bezeichnet die Möglichkeit, Aufgaben auf andere Rechner auszulagern, die dafür besser geeignet sind. Das kann zum einen die Rechenpower betreffen - wenn Rechner zur Verfügung stehen, die dedizierte Rechenleistung anbieten, sind rechenaufwendige Aufgaben dort wahrscheinlich besser aufgehoben weil schneller erledigt. Zum anderen könnten es Aufgaben sein, die spezielle Ressourcen erfordern - etwa Spezialhardware.

Für den Anwender ist dies alles verborgen. Er muß lediglich wissen, welchen Rechner er zur Erledigung der Aufgaben einteilen möchte und die entsprechende Konfiguration vornehmen.

Nicht alle Module können auf andere Rechner verschoben werden. Ein Modul, das dazu in der Lage sein soll, muß vom Entwickler speziell angepasst worden sein.

## Funktionsweise

Die Verlagerung von Funktionalitäten funktioniert kurz zusammengefasst wie folgt: Der Rechner, auf dem dWb+ ausgeführt wird, wird im Folgenden Master genannt. Neben dem Master existieren mehrere Remoting Server. Auf diesen wird ein minimaler Server ausgeführt, der nichts anderes tut, als auf Anweisung vom Master Objekte zu instantiieren, Operationen mit ihnen auszuführen und die Ergebnisse dieser Operationen an den Master zurückzugeben.

Im einzelnen funktioniert das wie folgt: Die Anwendung dWb+ transferiert eine Funktionalität auf einen Remoting Server, indem sie die Klassendefinition der Funktionalität zusammen mit allen Interfaces, die dWb+ davon in Anspruch nehmen möchte, an diesen sendet. Der Remoting Server lädt alle Klassendefinitionen, die zur Instantiierung benötigt werden, per HTTP von dem von dWb+ bereitgestellten Klassen-Server herunter und instantiiert die gewünschte Funktionalität. Die entstandene Instanz wird unter einem Namen, den dWb+ beim Transfer spezifiziert, in einer zentralen Registrierung gespeichert.

dWb+ kann nun auf den Remoting Server zugreifen, indem die Anwendung bei einer zentralen Registrierung nach Zugriff auf eine Instanz, die das gewünschte Interface implementiert, nachsucht. Dazu muß dieses Interface und der Name der Instanz und der Name des Remoting Servers angegeben werden. dWb+ erhält eine Instanz, die das gewünschte Interface implementiert zur weiteren Verwendung von der zentralen Registrierung geliefert.

Werden an diesem Interface Methoden aufgerufen, werden die Argumente und Parameter serialisiert und über das Netzwerk an den verantwortlichen Remoting Server versendet. Dieser führt die eigentliche Arbeit durch, indem in seiner Instanz die gewünschte Methode abgearbeitet wird. Das Ergebnis wird wieder serialisert und zurück an den Master versendet, wo das Ergebnis zur weiteren Verwendung an den ursprünglichen Aufrufer übergeben wird.

## Konfiguration

### dWb+

Um das Feature Remoting benutzen zu können, ist ein wenig Konfigurationsaufwand zu betreiben: Zunächst müssen die Rechner identifiziert werden, die zur Auslagerung benutzt werden sollen.

Die Anwendung selbst muß mit zwei System-Properties konfiguriert werden. Nur wenn diese beiden korrekte Werte beinhalten, kann Remoting benutzt werden:

### de.elbosso.dataflowframework.ClassLoaderServer.port

Der Port, auf dem den Rechnern die benötigten Klassen bereitgestellt werden sollen. Die Rechner, auf die die Module verlagert werden sollen, müssen die Funktionalität in Gestalt von Klassen irgendwie erhalten. Das ginge auch durch Kopieren der benötgiten Klassen. dWb+ macht sich eine Eigenart von Java zunutze: Auf allen Rechnern, auf die Module ausgelagert werden sollen, wird ein minimaler Server gestartet, der die Module empfängt und deren Arbeit steuert. Dieser Server lädt die benötigten Klassen über das HTTP-Protokoll nach. Der zuständige Server wird gestartet, wenn diese Property auf eine Zahl zwischen 1025 und 65535 gesetzt wird. Der angegebene Port darf natürlich nicht von einem anderen Programm bereits belegt sein.

### de.elbosso.util.beans.context.service.provider.RemotingServiceProvider.hosts

Eine Liste der Namen der Rechner, auf die Module verlagert werden dürfen. Die einzelnen Namen werden surch Semikolon (;) getrennt. Es ist geplant, daß sich die Rechner in späteren Versionen dynamisch mekden, sobald sie für Modulverlagerungen bereitstehen - im Moment muß die Liste bei Start von dWb+ stehen und kann nur geändert werden, wenn dWn+ neu gestartet wird.

## Remoting Server

Remoting Server werden die Rechner genannt, die verlagerte Module aufnehmen können. Voraussetzung dafür, einen Rechner als Remoting Server benutzen zu können, ist die Installation einer Java-Laufzeitumgebung in derselben Version wie auch auf dem Rechner, auf dem die Anwendung dWb+ ausgeführt wird. Auf all diesen Rechnern müssen Dienste gestartet werden, die dazu dienen, die verlagerten Module zu empfangen und ihre Ausführung zu steuern. Diese Dienste sind als kleine Anwendung realisiert, die mittels Skriptdateien gestartet werden. Der Name der Skriptdateien beginnt mit startrmiserver und trägt eine Betriebssystemspezifische Endung (.sh oder .bat). Zur Zeit ist nur Ubuntu-Linux als Betriebssystem für Remoting Server zertifiziert. Andere Linux-Varianten sollten auch funktionieren - gegebenenfall muß das Startskript an die zur Verfügung stehende Shell angepasst werden. Das Startskript erwartet zwei Parameter: Den Namen des Rechners, auf dem dWb+ ausgeführt wird und den Port, an dem der in dWb + integrierte HTTP-Server die Klassen anbietet.

# Benutzung

Die Benutzung von Modulen, für die Remoting möglich ist, unterscheidet sich zunächst nicht von der normaler Module. Sie werden ebenfalls per Drag'n'Drop auf den Workspaces platziert. Man kann sie ebenfalls Kopieren und an anderer Stelle wieder einfügen. Sie werden ganz normal in Workspaces gespeichert und stehen nach dem Einladen wieder zur Verfügung. Ihr Zustand wird bei diesen Aktionen wie der aller anderer Module konserviert und wiederhergestellt.

Die Zuordnung, auf welchem Rechner das Modul seine Arbeit verrichten soll, findet über das Untermenü RMI Server unten im Kontextmenü des Moduls statt. Dieses Untermenü stellt einen Eintrag pro Rechner dar, auf den man das Modul verlagern kann. Bei Auswahl eines dieser Menüpunkte wandert das Modul dorthin.

Der Rechner, dem das Modul zugeordnet wurde, ist selbstverständlich ebenfalls Teil des Zustandes des Moduls. Nach dem Einladen eines Workspaces werden die Module selbstständig wieder auf die Rechner verlagert, denen sie vor dem Speichern oder Kopieren zugeordnet waren.

# JMX

## Überblick

Module, deren zugrundeliegende Beans MBeans im Sinne von JMX sind, erhalten ein zusätzliches Untermenü (Tabelle 4.8, „Untermenü JMX“). Module, auf die das nicht zutrifft, erhalten dieses Menü nicht. MBeans werden durch dWb+ integriert, indem sie einfach über das Kontextmenü im MBean Server publiziert werden können. Diese Veröffentlichung kann ebenfalls über dieses Kontextmenü jederzeit rückgängig gemacht werden. Nach der Veröffentlichung verhalten sie sich wie jede andere MBean auch - über einen entsprechenden Agenten, der mit dem MBean Server verbunden ist, können die exponierten Eigenschaften abgefragt und geändert, sowie Notifications empfangen werden. Der Publikationsstatus der MBean wird beim Speichern des Workspace genau so mit übernommen, wie beim Kopieren über die Zwischenablage: Nach erneutem Einladen eines Moduls, das im MBean Server publiziert war, wird es sofort wieder publiziert. Die Kopie eines Moduls, das im MBean-Server publiziert war, wird nach Einfügen sofort ebenfalls publiziert.

# Eigenschaften

## Überblick

Modulen kann eine beliebige Menge von Eigenschaften zugeordnet werden. Diese Eigenschaften werden nicht zwischen Instanzen einer Modulklasse geteilt. Sie sind nur für die jeweilige Instanz gültig, für die sie spezifiziert wurden. Eigenschaften sind Tupel aus z Strings, wobei einer der beiden den Namen der Eigenschaft und der andere ihren Wert darstellt. Der Name ist ein eindeutiger Schlüssel - innerhalb einer Modulinstanz darf es keinen zweiten gleichen geben.

Eigenschaften sind neben den Attributen, die über die Parameterdialoge eingestellt werden können, eine weitere Möglichkeit, das Verhalten einer Modulinstanz anzupassen.

## Benutzung

Module können auf die gesetzten Eigenschaften über einen der BeanContext-Services zugreifen, die dWb + anbietet. Genaueres dazu findet sich in Abschnitt „Environment“

# Kapitel 5. Meta-Module

## Überblick

Meta-Module sind Komponenten, die sich innerhalb von Workspaces wie normale Module verhalten - sie verfügen über Input- und Output-Slots und können Daten anderer Module verarbeiten und ihre Ergebnisse an andere Module weitergeben.

Allerdings unterscheiden sie sich in wesentlichen Punkten von den Modulen, aus denen Workspaces normalerweise bestehen: Einige dieser Module sind lediglich eine visuelle Ausprägung eines Dienstes, den dWb+ als Anwendung bietet.

Manche dieser Module können auch nicht instantiiert werden, ohne dass der Anwender zuvor einige Fragen zu ihrer Konfiguration beantwortet.

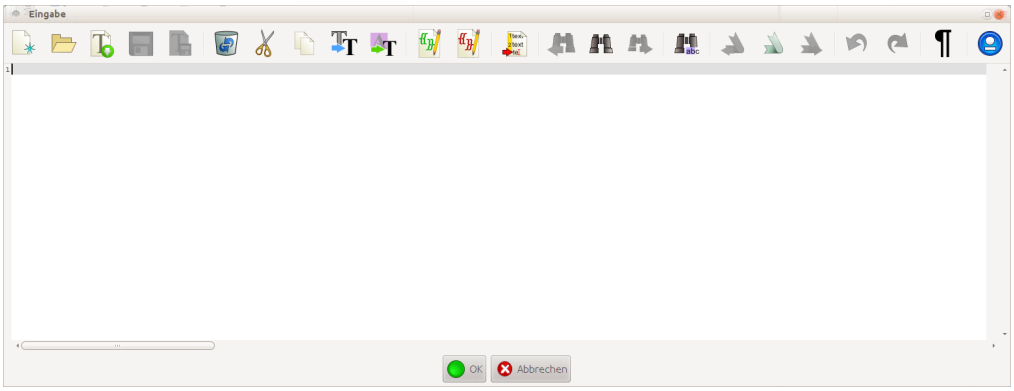
Allen Meta-Modulen ist aber eines gemeinsam: Sie können - nicht wie die anderen Module - nicht problemlos in andere Umgebungen migriert werden. Sie stellen Aspekte der Anwendung dWb+ dar und sind im Gegensatz zu den sogenannten normalen Modulen keine JavaBeans in der Hinsicht, dass sie "self-contained" sind und in jeder Umgebung funktionieren. Meta-Module benötigen als Umgebung zum Funktionieren dWb+.

In den folgenden Abschnitten werden die durch die Anwendung dWb+ angebotenen Meta-Module beschrieben und auf die jeweiligen Besonderheiten hingewiesen.

## Parametermodul

Parametermodule erlauben es, Instanzen beliebiger Klassen in einen Workspace einzubringen, wenn es zum Beispiel zu aufwendig wäre, extra ein Modul zu schreiben, nur damit einmalig eine Instanz einer bestimmten Klasse einen Algorithmus startet.

### Abbildung 5.1. Dialog zur Eingabe von Code-Fragmenten

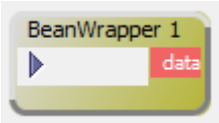



Wenn die entsprechende Action im Kontextmenü eines Workspaces ausgeführt wird, öffnet sich ein Dialog mit einem Quelltexteditor im Innern wie er in Abbildung 5.1, „Dialog zur Eingabe von Code-Fragmenten“ dargestellt ist. In diesen Editor kann man beliebigen Java-Quelltext eingeben. Die letzte Zuweisung innerhalb dieses Skripts wird als Wert und Instanz betrachtet, für die ein Parametermodul erstellt werden soll.

Anschließend wird das Modul erstellt, das genau einen Output hat. Dieser Output hat den Typ der zuletzt erstellten Instanz innerhalb des Skripts. Das Modul verfügt über einen Knopf. Mittels dieses Knopfes kann

man den aktuellen Status der Instanz, die das Parametermodul umhüllt, wiederholt absenden. Ein Beispiel dafür ist in Abbildung 5.2, „Beispiel für ein Parametermodul“ dargestellt.

### Abbildung 5.2. Beispiel für ein Parametermodul




Das Parametermodul hat - wie alle anderen auch - einen Parameterdialog. In diesem Fall zeigt der Parameterdialog eine Bedienoberfläche zur Modifikation der Properties der Instanz, die im Parametermodul enthalten ist.

Das Parametermodul kann nicht nur einfache Instanzen enthalten, sondern auch Arrays. In diesem Fall wird die Bedienoberfläche eine Liste sein, in der jeder Eintrag einer Instanz oder einem Arrayelement entspricht. Diese Elemente lassen sich dann wieder mittels Doppelklick modifizieren - daraufhin wird ein Dialog geöffnet, der es erlaubt, die Properties des zugehörigen Arrayelements anzupassen.

Parametermodule lassen sich nicht nur über das bereits erwähnte Kontextmenü des Workspaces öffnen. Bei einem Doppelklick auf einen Input-Slot eines Moduls analysiert dWb+ den Typ des Inputs. Kann dWb+ automatisch eine Instanz dieses Typs erzeugen, wird das getan und anschließend automatisch ein Parametermodul erzeugt, dessen Ausgang mit dem Input verbunden wird, auf den doppelt geklickt wurde.

Kann dWb+ keine Möglichkeit finden, den entsprechenden Typ automatisch zu instantiieren, öffnet sich ein Dialog wie bei der Erzeugung eines Parametermoduls über das Workspace-Kontextmenü, in dem man die Instantiierung mittels Java-Quelltext spezifizieren kann. Nach Schließen des Dialogs wird das Parametermodul auf dem Workspace platziert und sein Output mit dem Input verbunden, der doppelt angeklickt worden war.

# Actions im Quelltexteditor

### Tabelle 5.1. Actions im Editor zum Bearbeiten von Quelltextfragmenten

| | |
|---|---|
| Neues Dokument anlegen | |
| Ersetzt eingegebenen Text mit dem Inhalt einer Textdatei | |
| Hängt den Inhalt einer Textdatei an den bereits eingegebenen Text an | |
| Speichert den Inhalt der Datei | |
| Speichert den eingegebenen Text in einer Datei | |
| Löscht sämtlichen bereits eingegebenen Text | |
| Schneidet den markierten Text aus und kopiert ihn in die Zwischenablage | |
| Kopiert den markierten Text in die Zwischenablage | |
| Fügt Text aus der Zwischenablage an der Cursorposition ein | |
| Ersetzt bereits eingegebenen Text mit dem Inhalt der Zwischenablage | |
| Öffnet einen Dialog zur Verwaltung der Code-Templates, wie er beispielhaft in Abbildung 5.3, „Dialog zur Bearbeitung von Code-Templates“ dargestellt ist. Code-Templates sind Textfragmente mit einem Namen, die auf Anforderung - Voreinstellung ist **Ctrl**+**F12** - in den Text eingefügt werden. Dabei wird das Wort unter dem Cursor als Name eines einzufügenden Code-Templates interpretiert. Wird ein Code-Template diesen Namens gefunden, wird das Wort unter dem Cursor durch den Text des gefundenen Code-Templates ersetzt. | |

Namen für Code-Templates können aus Buchstaben und Zahlen bestehen und müssen mit einem Buchstaben beginnen. Enthalten Code-Templates Template-Parameter, kann der Anwender nach Einfügen des Templates mittels **Tabulator** zwischen den einzelnen Parametern umschalten - dabei springt der Cursor jeweils zur nächsten Position eines solchen Platzhalters und selektiert ihn. Template-Parameter werden durch eine grüne Unterstreichung hervorgehoben. **Esc** beendet diesen Modus. Template-Parameter können aus Buchstaben und Zahlen bestehen und müssen mit einem Buchstaben beginnen. Sie müssen mittels ${ und } eingeschlossen sein.

Öffnet einen Editor, mit dem der Anwender die Makros definieren kann, die bei Druck auf **Alt**+**F12** angewendet werden.

Diese Makros folgen der Syntax des Frameworks Velocity (`http://velocity.apache.org/`).

Setzt den Cursor in die gewählte Zeile

Vorheriges Auftreten finden

Text im Dokument finden

Nächstes Auftreten finden

Hebt jedes Auftreten des aktuell markierten Textes hervor

Setzt den Cursor auf die Zeile mit dem vorhergehenden Lesezeichen

Setzt (löscht) ein Lesezeichen in der aktuellen Zeile

Setzt den Cursor auf die Zeile mit dem nächsten Lesezeichen

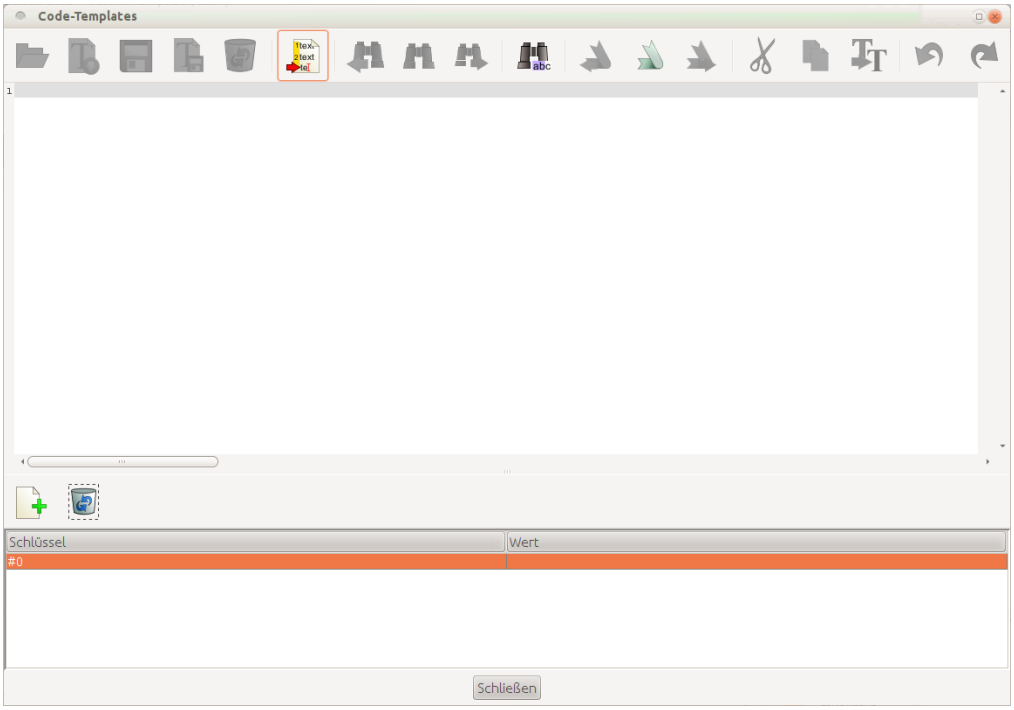
Undo

Redo

Zeilenende sichtbar / unsichtbar

null

**Abbildung 5.3. Dialog zur Bearbeitung von Code-Templates**




# Spaltmodul

Spaltmodule erlauben den einfachen Zugriff auf Elemente einer Datenstruktur. Erzeugen Module Instanzen einer Klasse und versenden diese, existiert manchmal das Problem, dass als Input für das nächste Modul in der Verarbeitungskette dieser Typ nicht benötigt wird, sondern nur eines seiner Elemente.

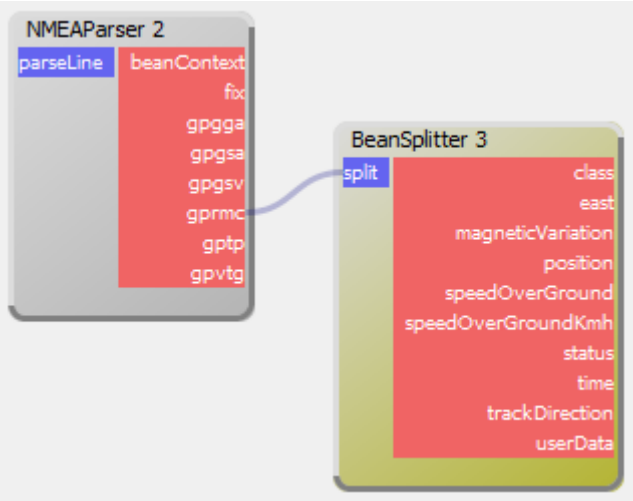
Ein Beispiel dafür wäre ein Modul, das Daten vom Typ Rectangle erzeugt. Sein Nachfolger benötigt aber nur die Information über die Breite des Rechtecks und erwartet daher konsequenterweise als Input nicht den Typ Rectangle, sondern Number.

Um für solche Fälle nicht immer neue Adaptermodule schreiben zu müssen, existiert das Konstrukt Spaltmodul: Damit ist es möglich, für beliebige Typen ein Modul zu erstellen, das den spezifizierten Typ als Eingabe erwartet und alle lesbaren Properties (alle Properties, die über eine get-Methode verfügen), als Ausgänge bereitstellt. In unserem Beispiel würde also ein Parametermodul mit einem Eingang vom Typ Rectangle und vier Ausgängen vom Typ Number (x, y, Breite, Höhe) entstehen.

Wenn die entsprechende Action im Kontextmenü eines Workspaces ausgeführt wird, öffnet sich ein Dialog mit einem Quelltexteditor im Innern wie er in Abbildung 5.1, „Dialog zur Eingabe von Code-Fragmenten“ dargestellt ist. In diesen Editor kann man beliebigen Java-Quelltext eingeben. Die letzte Zuweisung innerhalb dieses Skripts wird als Wert und Instanz betrachtet, für die ein Spaltmodul erstellt werden soll.

Anschließend wird das Modul erstellt, das genau einen Input hat. Dieser Input hat den Typ der zuletzt erstellten Instanz innerhalb des Skripts. Ein Beispiel für ein solches Modul ist in Abbildung 5.4, „Beispiel für ein Spaltmodul“ dargestellt.

**Abbildung 5.4. Beispiel für ein Spaltmodul**




Spaltmodule haben keinen Parameterdialog.

Spaltmodule lassen sich nicht nur über das bereits erwähnte Kontextmenü des Workspaces öffnen. Bei einem Doppelklick auf einen Output-Slot eines Moduls analysiert dWb+ den Typ des Outputs. Kann dWb+ automatisch eine Instanz dieses Typs erzeugen, wird das getan und anschließend automatisch ein Spaltmodul erzeugt, dessen Eingang mit dem Output verbunden wird, auf den doppelt geklickt wurde.

Kann dWb+ keine Möglichkeit finden, den entsprechenden Typ automatisch zu instantiieren, öffnet sich ein Dialog wie bei der Erzeugung eines Spaltmoduls über das Workspace-Kontextmenü, in dem man die Instantiierung mittels Java-Quelltext spezifizieren kann. Nach Schließen des Dialogs wird das Spaltmodul auf dem Workspace platziert und sein Input mit dem Output verbunden, der doppelt angeklickt worden war.

Es existiert eine Sonderform des Spaltmoduls: Wird beim Doppelklick auf den Output-Slot eines Moduls festgestellt, daß es sich beim Typ des Outputs um ein Feld oder Array handelt, öffnet sich ein Dialog, indem man - wie im Dialog zur Festlegung der zu druckenden Seiten in einer Textverarbeitung - Indices angeben kann. Wäre der Typ des Outputs also ein Integer-Array und der im Dialog angegebene Text zum Beispiel 0, 2-4, so würde ein Modul erzeugt, das einen Eingang vom Typ Integer-Array hätte. Es hätte 4 Ausgänge vom Typ Integer. Jedes Mal, wenn der Eingang ein Array empfinge, würden die Elemente mit den entsprechenden Indices aus dem Array extrahiert und über die zugehörigen Ausgänge an nachfolgende Module weitergegeben.

# SVG-Dokument

## Überblick

Dieses Metamodul dient der entfernten Überwachung der Schlüsselparameter eines Workspace. Es steuert die Darstellung einer SVG-Graphik auf einem Bildschirm (oder einem Projektor), die von einem DynamicSVG-Client angezeigt wird. dWb+ und dieser DynamicSVG-Client kommunizieren dabei über das Netzwerk: dWb+ sendet darzustellende Daten an den Visualisierungs-Client und dieser wiederum sendet Informationen über Ereignisse wie zum Beispiel Mausklicks und Tastaturbedienung an dWb+.

Es ist prinzipiell möglich, mehrere solcher Module in einem Workspace zu instantiieren. Diese verschiedenen Module können durchaus auch denselben Visualisierungs-Client steuern.

# Instantiierung

Nach Ausführen der entsprechenden Action im Kontextmenü des Workspaces fragt die Anwendung nach der SVG-Datei, die eingebunden werden soll. Wird hier eine entsprechende Graphikdatei ausgewählt, wird sie geladen und analysiert. Bei dieser Analyse werden alle semantischen Marker, die keine Ereignisse darstellen, extrahiert und in Input-Slots des entstehenden Moduls umgewandelt. Die semantischen Marker, die Ereignisse darstellen, werden als Output-Slots zum Modul hinzugefügt. Darüber hinaus erhält jedes so entstehende Modul noch je einen speziellen Input- und Output-Slot. Dieser spezielle Input-Slot erlaubt es, die SVG-Graphik zu ändern - eine entsprechende Botschaft wird auch an den Visualisierungs-Client gesendet. So kann man auf ein und demselben Endgerät zwischen verschiedenen Prozeßsichten umschalten. Wichtig dabei ist, dass diese Umschaltung nur auf dem Client stattfindet - das Modul selbst bleibt nach wie vor für die Steuerung der eingangs gewählten Datei verantwortlich. Zur Steuerung der Datei, auf die umgeschaltet wurde, kann man aber natürlich ein anderes SVG-Modul in den Workspace integrieren. Der spezielle Output-Slot dient ebenfalls der Signalisierung von Ereignissen - allerdings ist hier der Typ der versendeten Signale String - bei eintreten eines Signals wird durch diesen Slot der Name des Elements versendet, an dem das Ereignis eingetreten ist.

Die Output-Slots für Ereignisse haben alle den Typ Boolean. Der Typ für die Input-Slots wird durch den entsprechenden semantischen Marker bestimmt: für semantische Marker, die Text verarbeiten ist das zum Beispiel String, für semantische Marker, die Zahlen verarbeiten ist es demzufolge Number und für semantische Marker, die Farben verarbeiten, dementsprechend java.awt.Color.

# Parameterdialog

Der Parameterdialog dient der Parametrierung der Verbindung zum Visualisierungs-Client. Die rechte Karteikarte dient nur der Information und kann ignoriert werden - lediglich die Einstellungen auf der Karteikarte namens Atomare Properties müssen durch den Anwender angepasst werden. Wichtig sind hierbei die Eigenschaften host und port. Diese beiden spezifizieren die Verbindung auf Seiten des Visualisierungs-Clients. Die Verbindung wird nach Eingabe von Werten für die beiden Eigenschaften nicht sofort aufgebaut - dies geschieht über eine Action im Popupmenü des Moduls.

Die Eigenschaft servicename ist dafür da, über Bonjour oder Zeroconf einen Visualisierungs-Client zu finden, damit man den Host und den Port nicht mühsam einprägen und von Hand eingeben muss. Wird hier ein gültiger Wert angegeben, kann man über eine der Actions im Popupmenü Kandidaten finden, die diesen Dienst anbieten. Wählt man einen aus dieser Liste aus, wird automatisch Host und Port in die entsprechenden Eigenschaften übernommen.

Die Eigenschaft debug ist besonders während der Entwicklung der SVG-Graphik, aber auch währen der Erstellung des Workspace nützlich: ist sie aktiv, sorgt ein Klick auf ein Element im Visualisierungs-Client für ein Aufblinken aller semantischer Marker (Slots) am Modul, die damit in Verbindung stehen. Simultan bewirkt ein Überfahren eines Slots die Hervorhebung des Elementes, zu dem der entsprechende semantische Marker gehört.

# Actions

### Tabelle 5.2. Actions für das Meta-Modul SVG-Dokument

Öffnet die Verbindung.

Damit wird die Verbindung zu dem gewünschten, im Parameterdialog über die Eigenschaften host und port spezifizierten Visualisierungs-Client aufgebaut. Ab dem Zeitpunkt, zu dem dieser Verbindungsaufbau erfolgreich abgeschlossen wurde, werden die einzelnen Elemente

der SVG-Graphik entsprechend der definierten semantischen Marker und der Daten, die an das Modul gesendet werden, modifiziert.

Beim Aufbau der Verbindung wird getestet, ob die Graphik, die der Visualisierungs-Client anzeigt und die, die diesem Modul zugrunde liegt, identisch sind. Dieser Test wird anhand der semantischen Marker nach Anzahl, Name und Typ durchgeführt. Unterscheiden sich beide in mindestens einem dieser Kriterien, wird die Verbindung zwar trotzdem aufgebaut, der Anwender aber darüber informiert.

Schließt die Verbindung und gibt alle damit zusammenhängenden Ressourcen wieder frei.

Damit wird die Graphik zwar weiterhin auf dem Visualisierungs-Client sichtbar bleiben, jedoch keine der Daten mehr widerspiegeln, die an das Modul gesendet werden.

Lädt das Dokument neu, passt die Inputs und Outputs des Moduls entsprechend an und aktualisiert die Darstellung der Graphik

Startet die DynamicSVG-Komponente zur Anzeige der Graphik in einem eigenen Fenster.

Ein Visualisierungs-Client kann natürlich auch auf demselben Rechner gestartet werden, auf dem dWb+ läuft. In der Entwicklungsphase möchte man unter Umständen noch nicht die Graphik auf den Zielsystemen anzeigen - daher ist es möglich, einen Visualisierungs-Client mittels dieser Action aus dWb+ heraus zu starten - natürlich nur, wenn als host `localhost` eingetragen ist. Nach erfolgreichem Start dieses lokalen Visualisierungs-Client wird die Verbindung zur Kommunikation nicht automatisch hergestellt - dazu muss die entsprechende Action noch ausgeführt werden.

Verfügbare Dienste
: Öffnet einen Dialog zur Auswahl aus einer Liste potentieller Visualisierungs-Clients.

  Diese Clients wurden über den Zeroconf- oder Bonjour-Dienst gefunden. Die Suche wird über die Eigenschaft servicename gesteuert, die im Parameterdialog spezifiziert werden kann. Bei Klick auf einen Eintrag in der Liste werden Hostname und Port automatisch in die entsprechenden Eigenschaften des Parameterdialogs übernommen.

  Speichert ein Bild des aktuellen Status dieses Aviator-Moduls im Format PNG oder SVG.

  Das Format wird dabei über die gewählte Dateinamensendung festgelegt.

# Gruppe

## Instantiierung

Ein Gruppenmodul wird über den entsprechenden Eintrag des Kontextmenüs des Workspaces instantiiert. Weitere Möglichkeiten der Erstellung sind der Import eines Workspaces als Gruppe. In diesem Fall wird ein bestehender Workspace geladen und sein Inhalt direkt in eine Gruppe im aktuellen Workspace umgewandelt. Weiterhin besteht die Möglichkeit, ein Workspacefragment zu definieren und dieses per Refactoring in eine Gruppe umzuwandeln.

Wird eine Gruppe über das Workspace-Kontextmenü angelegt, entsteht ein neues Modul, das zunächst über keine Ein- und Ausgänge verfügt. Zur besseren Unterscheidung verfügt es über ein Icon auf dem Modul, das darauf hinweist, dass es sich dabei um ein Gruppenmodul handelt.

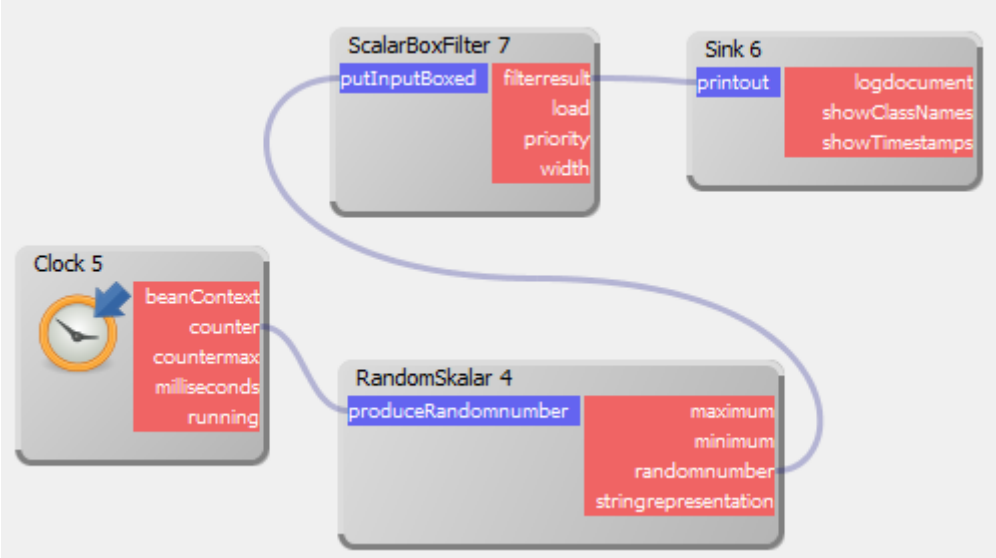
## Parameterdialog

Das Gruppenmodul verfügt wie alle anderen Module auch über einen Parameterdialog. Dieser kann durch Doppelklick geöffnet werden. Dieser Parameterdialog enthält als einzige Komponente einen Workspace wie den in dem sich das Modul selbst befindet. Ein Beispiel für einen solchen Workspace ist in

Abbildung 5.6, „Beispiel für einen hierarchisch aufgebauten Workspace“ dargestellt. Dieser Workspace ist funktional identisch zu dem in Abbildung 5.5, „Beispiel für einen nicht hierarchisch aufgebauten Workspace“ dargestellten, der nicht hierarchisch gegliedert ist. Man kann zu diesem Workspace wiederum Gruppen hinzufügen, so dass man sehr schnell eine beliebig tief gegliederte Hierarchie von Workspaces aufbauen kann. Im Workspace eines Gruppenmoduls stehen dieselben Funktionalitäten zur Verfügung wie im Hauptworkspace: Module werden per Drag'n'Drop hinzugefügt, Verbindungen werden per Drag'n'Drop definiert. Auch alle anderen Möglichkeiten zur Definition von Verarbeitungsketten sind unverändert vorhanden: Import von Workspaces, Definition von Verbindungen durch die Menüs der Output-Slots, Hinzufügen von Metamopdulen,...

Die Definition von Workspaces innerhalb einer Gruppe ist aber für sich allein genommen nicht sinnvoll. Es muss eine Möglichkeit geben, den innerhalb einer Gruppe definierten Funktionalitäten Datenaustausch mit dem Workspace zu gestatten, in den das Gruppenmodul eingebettet wurde. Dafür existieren zwei besondere Meta-Module, die im folgenden Abschnitt näher erläutert werden.

**Abbildung 5.5. Beispiel für einen nicht hierarchisch aufgebauten Workspace**




# Spezifische MetaModule

## Eingang

Input-Module werden über das Kontextmenü der Gruppen-Workspaces hinzugefügt. Wird die entsprechende Action ausgeführt, öffnet sich zunächst ein Dialog, der es gestattet, dem neuen Modul einen Namen zu geben. Diese Namen müssen innerhalb eines Workspace einzigartig sein. Wird ein entsprechender Name vergeben und der Dialog geschlossen, passieren mehrere Dinge:

Zunächst wird ein neues Modul im Gruppen-Workspace angelegt, das einen Output-Slot hat. Das mag im ersten Moment verwirren, da man doch im Kontextmenü ausgewählt hat, dass ein neuer Eingang angelegt werden sollte. Diese Bezeichnung bezieht sich auf die zweite Auswirkung der Action: im Gruppenmodul entsteht ein Eingang. Der Eingang des Gruppenmoduls ist direkt mit dem Ausgang des neuen Moduls im Gruppen-Workspace verbunden - sie stellen quasi die beiden Seiten einer Durchleitung aus dem übergeordneten Workspace in den Gruppen-Workspace dar.

Der Typ des Input-Slots am Gruppenmodul ist der gleiche wie der des Output-Slots am neuen Modul im Gruppen-Workspace - zunächst direkt nach der Erstellung ist das `java.lang.Object`. Erst

nachdem die erste Verbindung an den Input-Slot des Gruppenmoduls ansgeschlossen wird, ändert sich der Typ - diese Änderung vollzieht sich auf beiden Seiten, also auch innerhalb des Gruppen-Workspace. Diese Änderung kann nur einmal geschehen - wenn man so will, funktioniert das ähnlich Stammzellen: direkt nach ihrer Entstehung kann aus den Slots noch alles werden, sobald man aber das erste Mal eine Verbindung an das Gruppenmodul angeschlossen hat, ist der Typ festgelegt und kann sich nicht mehr ändern.

### Generische Module

Besondere Beachtung sollte dabei generischen Modulen zukommen - auch diese haben direkt nach ihrer Platzierung Ein- und Ausgänge, die den Typ `java.lang.Object` aufweisen. Sie verhalten sich ähnlich: Der Typ wird durch die Verbindung mit einem Ausgang festgelegt. (Siehe dazu auch Pogrammierhandbuch dWb+ im Kapitel 9, *Generics*.) Daraus ergibt sich, dass Ein- und Ausgänge generischer Module erst dann mit Input-Modulen verbunden werden sollten, wenn der Typ für den zu verbindenden Slot am generischen Modul spezifiziert wurde. Geschieht das nicht, ändert sich beim Anschließen an das generische Modul zwar der Typ des verbundenen Slots, nicht aber der Typ des damit verbundenen Input-Modules.

## Ausgang

Output-Module werden über das Kontextmenü der Gruppen-Workspaces hinzugefügt. Wird die entsprechende Action ausgeführt, öffnet sich zunächst ein Dialog, der es gestattet, dem neuen Modul einen Namen zu geben. Diese Namen müssen innerhalb eines Workspace einzigartig sein. Wird ein entsprechender Name vergeben und der Dialog geschlossen, passieren mehrere Dinge:

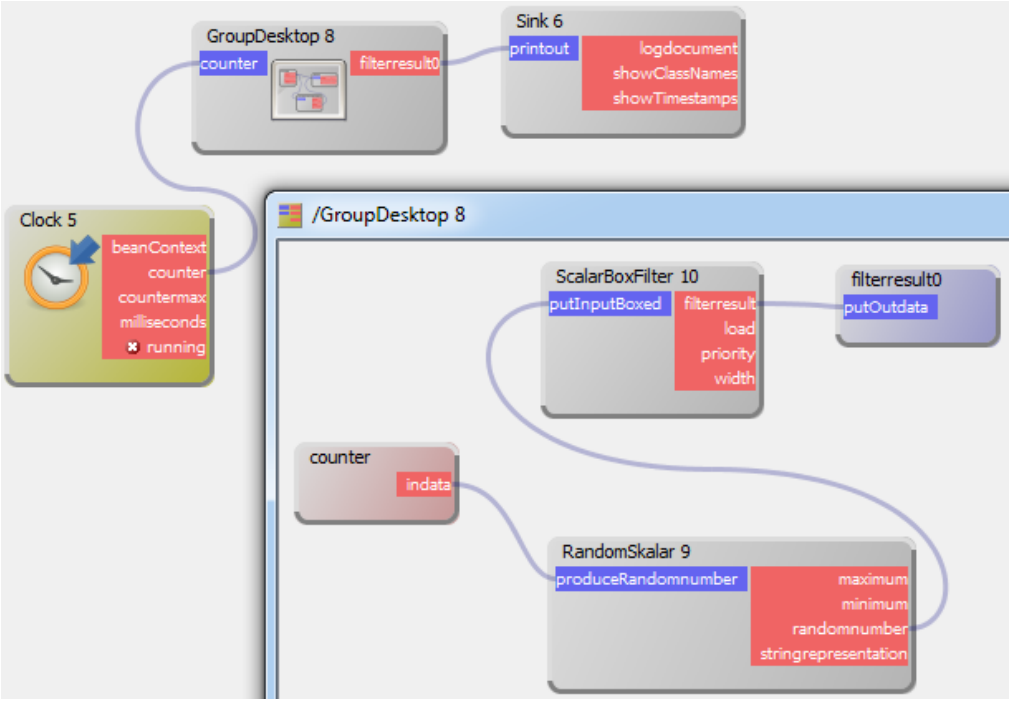
Zunächst wird ein neues Modul im Gruppen-Workspace angelegt, das einen Input-Slot hat. Das mag im ersten Moment verwirren, da man doch im Kontextmenü ausgewählt hat, dass ein neuer Ausgang angelegt werden sollte. Diese Bezeichnung bezieht sich auf die zweite Auswirkung der Action: im Gruppenmodul entsteht ein Ausgang. Der Ausgang des Gruppenmoduls ist direkt mit dem Eingang des neuen Moduls im Gruppen-Workspace verbunden - sie stellen quasi die beiden Seiten einer Durchleitung aus dem Gruppen-Workspace in den übergeordneten Workspace dar.

Der Typ des Output-Slots am Gruppenmodul ist der gleiche wie der des Input-Slots am neuen Modul im Gruppen-Workspace - zunächst direkt nach der Erstellung ist das java.lang.Object. Erst nachdem die erste Verbindung an den Input-Slot des neuen Moduls ansgeschlossen wird, ändert sich der Typ - diese Änderung vollzieht sich auf beiden Seiten, also auch am Gruppenmodul im übergeordneten Workspace. Diese Änderung kann nur einmal geschehen - wenn man so will, funktioniert das ähnlich Stammzellen: direkt nach ihrer Entstehung kann aus den Slots noch alles werden, sobald man aber das erste mal eine Verbindung an das neue Modul angeschlossen hat, ist der Typ festgelegt und kann sich nicht mehr ändern.

### Generische Module

Besondere Beachtung sollte dabei generischen Modulen zukommen - auch diese haben direkt nach ihrer Platzierung Ein- und Ausgänge, die den Typ `java.lang.Object` aufweisen. Sie verhalten sich ähnlich: Der Typ wird durch die Verbindung mit einem Ausgang festgelegt. (Siehe dazu auch Pogrammierhandbuch dWb+ im Kapitel 9, *Generics*.) Daraus ergibt sich, dass Ein- und Ausgänge generischer Module erst dann mit Output-Modulen verbunden werden sollten, wenn der Typ für den zu verbindenden Slot am generischen Modul spezifiziert wurde. Geschieht das nicht, ändert sich beim Anschließen an das generische Modul zwar der Typ des verbundenen Slots, nicht aber der Typ des damit verbundenen Output-Modules.

**Abbildung 5.6. Beispiel für einen hierarchisch aufgebauten Workspace**




## BeanContext Segmentierung

Die Gruppen-Workspaces dienen einer weiteren organisatorischen Aufgabe: Jeder Gruppen-Workspace ist ein eigener BeanContext. Damit ist es möglich, innerhalb eines Workspaces unterschiedliche Implementierungen ein und desselben Service-Interface zu nutzen: Werden Service-Implementierungen als Module realisiert (vergleiche Pogrammierhandbuch dWb+ im Kapitel 7, *BeanContext und BeanContextServices konsumieren*), werden diese durch die Instantiierung des jeweiligen Moduls im Workspace als Service registriert. Dies geschieht nur direkt in dem Workspace, in dem das Modul platziert wurde. Die Workspaces oberhalb oder neben dessen in der Hierarchie werden darüber nicht informiert. So kann man auf einfache Art und Weise mehrere unterschiedliche Implementierungen eines Service Interface parallel nutzen: Man legt einfach mehrere Gruppen-Workspaces nebeneinander an und registriert in jedem eine andere Implementierung des Service-Interfaces. Die Ergebnisse, die durch die Nutzung der einzelnen Implementierungen erzielt werden, kann man dann aus den Gruppen-Workspaces in den die Gruppen-Workspaces beinhaltenden zur weiteren Verarbeitung ausleiten.

# Persistenzunterstützung

## Überblick

Die Persistenzunterstützung ist ein Dienst , der es erlaubt, im System erzeugte und zwischen Modulen versendete Daten mit Zeitstempel zu protokollieren. Dieser Dienst umfasst zum einen die Protokollierung der Daten. Die zweite Seite fasst die Berichts- und Exportfunktionalitäten zusammen: Man kann die protokollierten Daten in dWb+ anschauen und nach verschiedensten Kriterien filtern. Es ist darüber hinaus auch möglich, Berichte aus den Protokollen zu erzeugen und in verschiedensten Formaten zu exportieren.

Die Speicherung des Protokolls geschieht in einer relationalen Datenbank. Der Anwender muss dazu keinen dedizierten Server zur Verfügung stellen - dWb+ bringt eine eigene leichtgewichtige relationale Datenbank mit, die eingesetzt wird, wenn kein Server zur Verfügung steht.

Die Protokollierung wird an die ID eines Moduls gebunden. Die ID zusammen mit dem abstrakten Namen des jeweiligen Output-Slots ergibt den Spaltennamen in der Tabelle der Datenbank, in die die Werte gespeichert werden. Die ID wird für jedes Modul bei der Instantiierung vergeben. Dies geschieht transparent im Hintergrund - der Anwender sieht diese ID normalerweise nie. Die ID wird beim Speichern eines Workspace ebenfalls gespeichert, so dass die Module nach dem Einladen eventuell einen anderen Titel bekommen, aber über ihre ID ihre Identität zweifelsfrei festgestellt werden kann.

Die Protokollierung funktioniert auch über Sitzungen hinweg: Da die ID eines Modules nur bei der erstmaligen Instantiierung festgelegt wird und nicht beim Einladen, können Workspaces nach Neustart von dWb+ wieder geladen und ausgeführt werden. Die Protokollierung ordnet die jeweiligen Module und Slots automatisch wieder richtig zu, so dass im Protokoll der Neustart lediglich als Lücke in den Werten der Zeitstempel sichtbar wird.

# Instantiierung

Ein Modul der Persistenzunterstützung wird auf jeden Fall zur Protokollierung benötigt. Dieses Modul wird automatisch der Ebene persistence zugeordnet und kann so auf einfache Art und Weise über das Kontextmenü des Workspaces ausgeblendet werden.

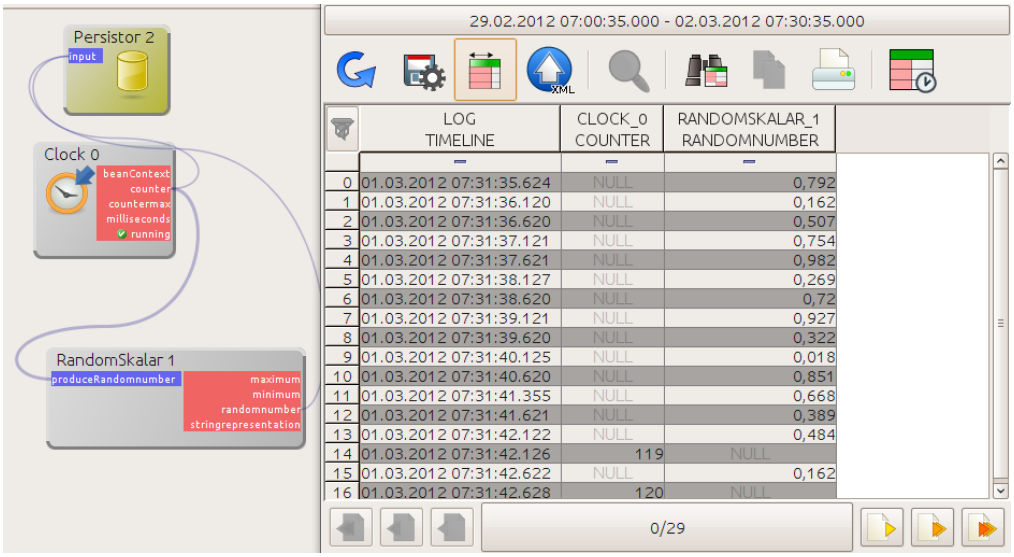
Ein solches Modul hat genau einen Eingang - alle Daten, die an diesen Eingang verschaltet werden, werden in die Protokollierung aufgenommen. Mit anderen Worten bestimmen die Verbindungen zum Persistenz-Modul, welche Daten welcher Module protokolliert werden. Es existiert kein Automatismus, nach dem alle Daten aller Module protokolliert werden, sobald ein Modul der Persistenzunterstützung platziert wurde. Das spart Ressourcen, da nur die tatsächlich benötigten Daten protokolliert werden. Andererseits sollte man Sorgfalt bei der Auswahl der zu protokollierenden Daten walten lassen, um nicht feststellen zu müssen, dass alle Arbeit umsonst war, weil man bei der Protokollierung ein oder mehrere Ausgänge vergessen hat. Abbildung 5.7, „Parameterdialog der Persitenzunterstützung mit Beispielworkspace“ zeigt einen Beispielworkspace mit Inhalten der Datenbank.

Prinzipiell reicht ein Modul bereits aus, es ist aber durchaus möglich, mehrere Module von dieser Sorte auf dem Workspace zu platzieren. Alle an die verschiedenen Persistenzunterstützungsmodule angeschlossenen Daten werden in ein und dieselbe Datenbank geschrieben. Man kann damit also auch in den Berichten Daten gegenüberstellen, die in verschiedene Module der Persistenzunterstützung geflossen sind. Die Module sind also lediglich verschiedene Öffnungen in einen großen Behälter.

# Parameterdialog

Zur Auswertung der Protokolle wird der Parameterdialog benutzt. Dieser Dialog zeigt, wenn er geöffnet wird, zunächst im oberen Bereich einen Knopf, auf dem das Zeitintervall eingeblendet ist, für das der aktuell gerade angezeigte Bericht erzeugt wurde. Darunter folgt die Werkzeugleiste zur Konfiguration des Berichtes und daran schließt sich dann die tabellarische Übersicht der Daten an.In Abbildung 5.7, „Parameterdialog der Persitenzunterstützung mit Beispielworkspace“ ist ein Parameterdialog dargestellt.

### Abbildung 5.7. Parameterdialog der Persitenzunterstützung mit Beispielworkspace




Darunter sind Knöpfe angeordnet, die die Navigation in den Datensätzen erleichtern: Mit den drei Knöpfen links und rechts bewegt man sich wahlweise seitenweise oder um jeweils um 10 Prozent vorwärts oder rückwärts oder gleich bis zur ersten beziehungsweise letzten Seite. Weiterhin ist es möglich, durch Anklicken des Knopfes in der Mitte eine Zeile zu spezifizieren, ab der man die Daten aufgelistet haben möchte. Dieser Knopf reagiert auch darauf, dass das Mausrad bewegt wird: geschieht das, während sich der Mauszeiger über diesem Knopf befindet, wird zeilenweise durch die Resultate gescrollt.

Wenn man die Persistenzunterstützung zum ersten Mal benutzt, wird es wahrscheinlich so sein, dass die tabellarische Übersicht leer ist. Wurden noch keine Verbindungen mit dem Persistenzunterstützungsmodul hergestellt, existieren natürlich auch noch keine Daten, die protokolliert wurden. In diesem Fall muss man eine solche Verbindung herstellen, um die Funktionalitäten der Persistenzunterstützung - was Auswertung, Filterung und Berichtswesen angeht - testen zu können.

Wichtig ist auch das ausgewählte Zeitintervall: Wurde der Parameterdialog geöffnet, wird das dargestellte Zeitintervall automatisch so eingestellt, dass es eine halbe Stunde in die Vergangenheit reicht. Hat man also die Verbindungen zu anderen Modulen erst nach der Öffnung des Parameterdialoges hergestellt, wird die tabellarische Übersicht auch weiterhin nichts anzeigen, bis man das Intervall - genauer: dessen Endzeitpunkt - entsprechend ändert.

Die tabellarische Übersicht verfügt - wie bereits oben erwähnt - über einige in der Werkzeugleiste zusammengefasste Actions, deren Wirkungsweise im Folgenden erklärt wird:

### Tabelle 5.3. Werkzeugleiste für das Meta-Modul Persistenzunterstützung

| | |
|---|---|
| | Lädt die Daten neu |
| | Öffnet ein Fenster zum Ausblenden von Spalten |
| | Verbreitert die Spalten so, dass der Inhalt vollständig sichtbar wird |
| | Speichert den Inhalt dieses Views in einer XML-Datei.<br><br>Diese Aktion erlaubt das Speichern eines Views als XML-Dokument. Dieses wird in die vom Nutzer gewählte Datei geschrieben. Wenn der Dateiname nicht auf .xml endet, wird diese Endung angehängt. Weiterhin wird ein XML-Schema unter dem vom Nutzer gewählten Namen mit der Endung .xsd gespeichert. Dieses kann man zum Beispiel dazu benutzen, die Gültigkeit des Dokuments zu verifizieren. |

Zeigt eine übersichtlichere Darstellung einer selektierten Zeile an

Diese Aktion dient dazu, Details eines Datensatzes besser erfassen zu können. Speziell bei Tabellen oder Ergebnissen von Abfragen mit sehr vielen Spalten oder Spalten mit großen (langen) Inhalten ist es so, dass die Übersichtlichkeit bei tabellarischer Übersicht zwar gut ist, die Details einer Ergebniszeile sich aber nur schwer oder überhaupt nicht zufriedenstellend darstellen lassen. Selektiert man in einer Tabelle eine Zeile, kann diese Aktion angewählt werden. Sie ändert die Darstellung wie unten gezeigt dahingehend ab, dass nun nur noch die Daten jeweils einer einzelnen Zeile dargestellt werden. Mit den Knöpfen am unteren Teil der GUI kann man zwischen den Datensätzen wechseln.

Regelbasierte Hervorhebung bestimmter Zellen

Kopieren markierter Zellen ins die Zwischenablage

Reporting ...

Diese Aktion erlaubt den Zugriff auf sämtliche Aktionen zum Thema Reporting.

Öffnet einen Dialog zur Verwaltung der dynamischen Spalten für diesen View.

Diese Aktion öffnet den Dialog zur Definition einer neuen dynamischen Spalte. Dynamische Spalten berechnen ihre Inhalte auf vom Anwender vorgegebene Weise aus den anderen Werten der jeweiligen Zeile. Diese Spalte sind in allen Aspekten realen Spalten völlig gleichgestellt. Auch in Berichten könne sie natürlich genauso eingesetzt werden. Der Manager verwaltet die dynamischen Spalten in einer Liste. Dazu bietet er zwei Aktionen an:

**Tabelle 5.4. Actions für die Verwaltung dynamischer Spalten**

Hinzufügen eines Elements zu der Liste

Entfernt die ausgewählten Elemente aus der Liste

# Aviator

## Überblick

Der Aviator gestattet es ähnlich dem Meta-Modul SVG-Dokument, Informationen zur Überwachung von Prozeßzuständen intuitiv, klar und eindeutig zu visualisieren. Die Visualisierung kann lokal erfolgen oder mittels des Aviators auch im Internet verteilt werden. Anders als beim SVG-Dokument ist es hier so, Daß der Anwender nicht gezwungen ist, die SVG-Graphik zu erstellen. Seine Aufgabe ist es vielmehr, aus einer vorhandenen Palette vorgefertigter Instrumente ein virtuelles Cockpit zusammenzustellen und den einzelnen Instrumenten dann die Signale aus dem Workspace zuzuordnen, die die Instrumente darstellen sollen.

Dabei kann die Zusammenstellung der Instrumente im Nachhinein ebenso geändert werden, wie ihre Zuordnung zu den darzustellenden Signalen. Es ist problemlos möglich, bereits bestehende Aviator-Module zu löschen oder weitere hinzuzufügen.

## Instantiierung

Nach der Instantiierung zeigt sich das Modul zunächst mit je einem Aus- und Eingang. Öffnet man den Parameterdialog eines neu instantiierten Aviatormoduls, ist dieser abgesehen von einem Knopf leer.

# Parameterdialog

### Abbildung 5.8. Parameterdialog Aviator mit zwei Instrumenten

Außer der Action zum Konfigurieren des Aviatormoduls zeigt der Parameterdialog die Instrumente als Teil des Virtuellen Cockpits an. Es existiert ein spezieller semantischer Marker namens Label. Ist innerhalb eines Instrumentes ein Element damit markiert, bedeutet das, dass dieses Element eine Skalenbeschriftung darstellt. Solche semantischen Marker, beziehungsweise ihren Inhalt kann man auf der zweiten Karteikarte des Parameterdialoges editieren (diese zweite Karteikarte existiert nur, wenn mindestens ein Label im virtuellen Cockpit enthalten ist). Die letzte Karteikarte schließlich bietet Eingabefelder für die Parameter Host und Port. Diese Parameter werden benötigt, wenn der Aviator einen externen Visualisierungs-Client steuern soll. Näheres dazu findet sich in „Actions“.

### Tabelle 5.5. Actions im Parameterdialog der Aviatormodule

| | |
|---|---|
| | Dieser Knopf öffnet den Dialog zur Zusammenstellung des virtuellen Cockpits. Zunächst sind hier alle Schaltflächen ausgegraut, da die Anwendung zu Beginn das Datenverzeichnis nach eventuell neu hinzugekommenen Instrumenten durchsucht. Sobald diese Suche abgeschlossen ist, öffnet sich der Dialog zum Hinzufügen eines Instrumentes. Hier kann man zunächst nach visuellen Kriterien die Instrumente auswählen, die zum Cockpit gehören sollen. |

## Konfiguration

**Abbildung 5.9. Ausschnitt aus der Palette zur Auswahl der Instrumente**

**Tabelle 5.6. Actions zur Konfiguration der Instrumente**

| | |
|---|---|
| | Öffnet einen Dialog zur Auswahl des hinzuzufügenden Instruments. Abbildung 5.9, „Ausschnitt aus der Palette zur Auswahl der Instrumente“ zeigt ein Beispiel für die Vielfalt zur Verfügung stehender Instrumente zur Visualisierung numerischer Werte. |
| | Details aller Instrumente einblenden |
| | Details aller Instrumente ausblenden |
| | Öffnet einen Dialog zur Anpassung von Größen und Positionen der Instrumente im Cockpit<br><br>Der sich daraufhin öffnende Dialog wird weiter unten näher beschrieben |

Soll mehr als ein Instrument im virtuellen Cockpit erscheinen, sind diese mittels der entsprechenden Action in der Werkzeugleiste hinzuzufügen. Nachdem alle Instrumente hinzugefügt wurden, kann man nun die Details eines jeden Instrumentes festlegen.

Eine Besonderheit existiert bei Ampeln: Hier wird nicht nur das numerische Signal bestimmt, dessen Wertveränderungen das Instrument anzeigen soll, sondern auch die Umschaltschwellen bei denen eine andere der Ampelfarben aktiv wird. Es ist maximal möglich, fünf Ampelfarben zu definieren. Ein Beispiel dafür findet sich in Abbildung 5.10, „Konfiguration der Instrumente“

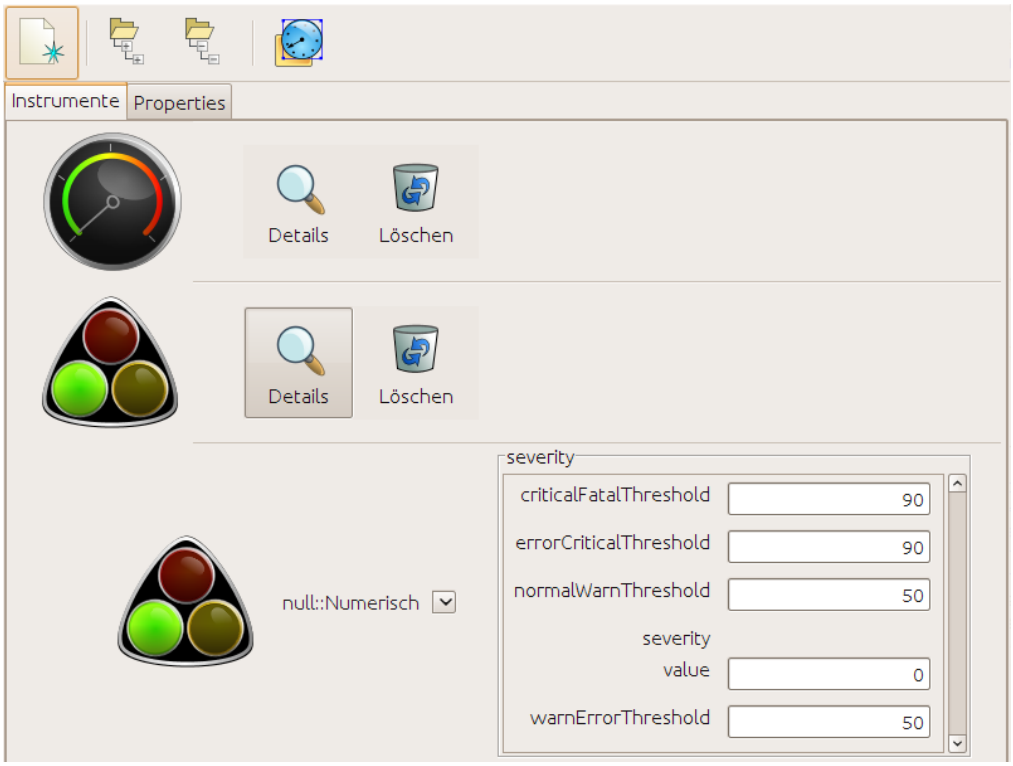
Diese Konfiguration der Instrumente und ihrer Zuordnung befindet sich auf der Karteikarte namens Instrumente. Auf der Karteikarte Properties befinden sich einige weitere Eigenschaften, die man modifizieren kann:

imgFileName — Name der Graphik. Dieser Name ist abgeleitet aus dem Namen des Moduls. Man kann ihn anpassen. Dieser Name findet Verwendung als Teil der URL wenn die Aviatorgraphik im Web publiziert wird.

sizeFactor — Dieser Faktor bestimmt die letztendliche Größe der Darstellung der SVG-Graphik. Ein Wert von 1 bewirkt keine Größenänderung - die Designgröße wird so verwendet, wie in der ursprünglichen SVG-Datei festgelegt. Werte kleiner als 1 lassen die Graphik schrumpfen, Werte größer als 1 lassen sie wachsen. Diese Eigenschaft ist nun überflüssig, da Anordung und Größe der Instrumente nun über den visuellen Editor spezifiziert werden.

templateFile — Eine Datei, die benutzt wird, um die Aviatorgraphik im Web zu publizieren. Damit ist es möglich, das virtuelle Cockpit in beliebige Webseiten einzubauen. Der Aufbau der Datei wird im Programmierhandbuch beschrieben.

**Abbildung 5.10. Konfiguration der Instrumente**




## Layout

Dieser Dialog enthält alle Instrumente, die bisher in dieses Aviatormodul aufgenommen wurden. Die Instrumente lassen sich innerhalb dieses Dialoges verschieben und in ihrer Größe ändern. Überlappen sich zwei Instrumente kann man vorgeben, welches jeweils über dem anderen liegt - welches also teilweise von dem jeweils anderen verdeckt wird. Die hier getroffene Anordnung wird auf dem Parameterdialog ebenso benutzt, wie bei der Publikation im WWW.

Selektierte Instrumente erkennt man am blauen Rahmen um die Selektion. Ist bereits ein Instrument selektiert, kann man ein weiteres zur Auswahl hinzufügen, indem man das hinzuzufügende Instrument bei gedrückter Strg-Taste mit der linken Maustaste anklickt. Wird ein selektiertes Element verschoben, werden alle in der Auswahl befindlichen Instrumente ebenfalls verschoben. Gleiches trifft auf die Größenänderung zu.

Die Größe der Auswahl (eines oder mehrerer Instrumente) wird geändert, indem die Anfasser des Rahmens um die Selektion bei gedrückter und festgehaltener linker Maustaste verschoben werden. Größenänderungen sind nur mittels der Anfasser möglich - befindet sich die Maus über einem solchen, wird der Rahmen um die Auswahl türkis dargestellt.

Ist genau ein Instrument selektiert, werden die Actions in der Werkzeugleiste verfügbar, mit denen man bestimmen kann, welches Instrument welche anderen Instrumente überdecken - also "im Stapel oben liegen" soll:

### Tabelle 5.7. Actions zur Bestimmung der Reihenfolge im Stapel

| | |
|---|---|
| | Bewegt das ausgewählte Instrument eine Ebene nach unten |
| | Bewegt das ausgewählte Instrument eine Ebene nach oben |

Sind zwei oder mehr Instrumente selektiert, wird ein Kontextmenü verfügbar, mit dem man die selektierten Instrumente in verschiedener Weise anordnen kann:

### Tabelle 5.8. Kontextmenü fürs Layout der Instrumente im Aviator

### Tabelle 5.9. Ausrichtung

| | |
|---|---|
| | Richtet die Instrumente so aus, dass die linken Kanten alle auf einer gedachten Linie liegen |
| | Richtet die Instrumente so aus, dass die rechten Kanten alle auf einer gedachten Linie liegen |
| | Richtet die Instrumente so aus, dass die oberen Kanten alle auf einer gedachten Linie liegen |
| | Richtet die Instrumente so aus, dass die unteren Kanten alle auf einer gedachten Linie liegen |
| | Richtet die Instrumente so aus, dass die horizontalen Mitten alle auf einer gedachten Linie liegen |
| | Richtet die Instrumente so aus, dass die vertikalen Mitten alle auf einer gedachten Linie liegen |
| | Verteilt die Instrumente so, dass der horizontale Zwischenraum zwischen je zwei benachbarten überall gleich ist |
| | Verteilt die Instrumente so, dass der vertikale Zwischenraum zwischen je zwei benachbarten überall gleich ist |

## Auswirkungen der Konfigurationsänderungen

Nach dem Schließen des Dialogs werden die vorgenommenen Konfigurationsänderungen in das Modul übernommen: Die Instrumente werden entsprechend ihrer Konfiguration im Parameterdialog angezeigt. Abbildung 5.8, „Parameterdialog Aviator mit zwei Instrumenten“ zeigt ein Beispiel dafür nachdem zwei Instrumente hinzugefügt wurden. Entsprechend der semantischen Marker vom Typ Label werden die Informationen auf dem zweiten Karteireiter aktualisiert. Alle semantischen Marker, die nicht vom Typ Label sind, werden als Eingänge zum Aviatormodul hinzugefügt. Die Verknüpfung der zugehörigen Graphikelemente mit Signalen aus dem Desktop erfolgt wie auch beim Meta-Modul SVG-Dokument: ein einfaches Drag'n'Drop stellt die Verbindung her, sofern der Datentyp des Signals mit dem Typ des semantischen Markers kompatibel ist. Die Tooltips dieser aus semantischen Markern erzeugten Slots stellen eine Ansicht des jeweiligen Instruments dar, damit beim Verbinden der Signale keine Mißverständnisse aufkommen können.

Instrumente, die zwei oder mehr semantische Marker aufweisen, haben auch für jeden der semantischen Marker einen separaten Slot. In der Miniaturansicht im Tooltip ist das Element mit dem zugehörigen semantischen Marker hervorgehoben, um die Zuordnung zu erleichtern. Bei einem Instrument mit zwei Zeigern würden also an den zwei Slots jeweils eine Miniaturansichten des Instrumentes angezeigt, wobei in jeder der beiden je ein Zeiger rot hervorgehoben sein würde.

# Actions

### Tabelle 5.10. Actions für das Meta-Modul Aviator

Speichert ein Bild des aktuellen Status dieses Aviator-Moduls im Format PNG oder SVG.

Maßgeblich für das Format der gespeicherten Datei ist die Dateinamensendung.

Die angezeigte Graphik kann über das Internet mit einer Vielzahl verschiedener Web-Browser betrachtet werden.

Diese Aktion startet die Publikation der Aviatorgraphik über den in dWb+ eingebauten Webserver. Falls sehr viele Zugriffe auf diese Darstellung erwartet werden, sollte man überlegen, ob man den Weg der Publikation über den Proxy wählt (siehe unten). Die URL zum Zugriff erhält man über den entsprechenden Menüpunkt dieses Menüs.

Die angezeigte Graphik kann über das Internet mit einer Vielzahl verschiedener Web-Browser betrachtet werden (über Proxy).

Diese Variante setzt einen entsprechend konfigurierten und zugreifbaren DynamicSVG-Proxy voraus.

Die Anwendung muss über die Adresse, an der der Proxy auf Verbindungen wartet informiert werden, damit die Action verfügbar wird. Das geschieht mittels Übergabe einer System-Property beim Programmstart wie in Kapitel 1, *Überblick* in „Programmstart“ beschrieben.

In dieser Betriebsart werden Änderungen an der Graphik auf den Proxy gespiegelt, von wo sich Benutzer dann die Visualisierung holen können. Dieser Betriebsmodus hat den Vorteil, dass die Last auf die Anwendung dWb+ beherrschbar bleibt, da sie nicht als Web-Server arbeiten muss. Die URL zum Zugriff erhält man über den entsprechenden Menüpunkt dieses Menüs.

Die angezeigte Graphik kann nicht länger mit Web-Browser betrachtet werden

Diese Action erlaubt es, ein konfiguriertes Cockpit oder Dashboard ähnlich wie das Metamodul SVG-Dokument auf beliebige Visualisierungs-Clients zu publizieren.

Aktuelle Konfiguration wird an den Visualisierungs-Client gesendet - anschließend wird er ständig mit aktualisierten Daten versorgt

Diese Action ist nur verfügbar, wenn vorher bereits die Web-Publikation gestartet wurde. Diese Aktion führt mehrere Schritte hintereinander aus: zunächst wird versucht, eine Verbindung zu einem Visualisierungs-Client aufzubauen, der auf dem Rechner unter dem angegebenen Hostnamen läuft und an dem angegebenen Port auf Verbindungen wartet. Ist das erfolgreich, wird im zweiten Schritt dieser Client angewiesen, die Datei zu laden, die im Web publiziert wurde. Anschließend werden alle Datenaktualisierungen an den Visualisierungs-Client gesendet, die durch die Inputslots vom Modul empfangen wurden.

Damit funktioniert der Aviator dann ähnlich dem in Kapitel 5, *Meta-Module* in „SVG-Dokument“ beschriebenen Meta-Modul.

Verbindung zum Visualisierungs-Client wird geschlossen

Kopiert die URL, unter der die Graphik im Web publiziert ist, in die Systemzwischenablage

Öffnet den Browser zur Darstellung der im Web publizierten Graphik

## Alternative Visualisierung

Das Aviatormodul hat je einen Aus- und Eingang, die miteinander in Beziehung stehen: Ein Eingang wird zum Modul hinzugefügt, der - sobald irgendein Modul Daten an diesen Eingang sendet - folgende Aufgabe hat: Die Darstellung der Visualisierungen des Cockpits und der Instrumente wird im Parameterdialog eingestellt. Jedes Datum, das das Modul an diesem Eingang erreicht, sorgt dafür, dass der Status der Instrumente zu diesem Zeitpunkt in eine Datenstruktur im Arbeitsspeicher geschrieben wird. Aus dieser Datenstruktur wird ein Image gemacht und dieses Image über den entsprechenden Eingang an andere Module versendet.

# Interaktive Formulare

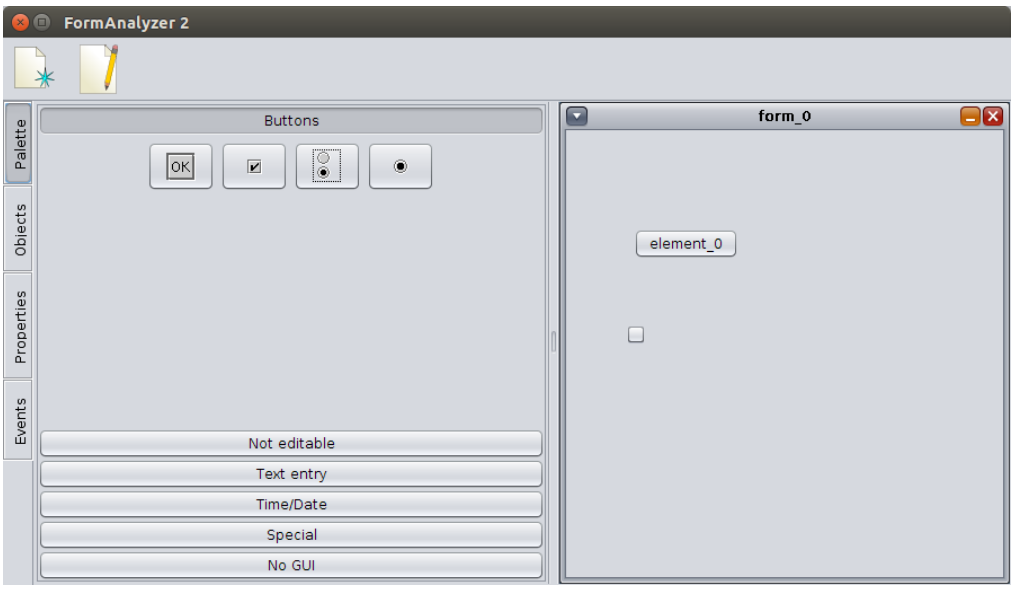
## Überblick

Dieses Metamodul bietet die Möglichkeit, innerhalb der dWb+ komplexe Anwendungen zu erstellen. Dabei werden diese Anwendungen in Formulare strukturiert. Jedes Formular enthält eine beliebige Anzahl an Formularelementen. Diese Formularelemente können visuelle Entsprechungen haben. Dazu könnten zum Beispiel traditionelle Elemente zur Gestaltung von Benutzeroberflächen fallen. Außerdem sind Elemente ohne visuelle Entsprechung möglich. Ein Beispiel für solch ein unsichtbares Element wäre etwa eines, über das auf Inhalte einer Datenbank zugegriffen werden könnte.

Dieses Metamodul öffnet einen Dateiauswahldialog. Dadurch hat der Anwender die Möglichkeit, eine bereits fertig konfigurierte Anwendung als Modul in einen Workspace zu integrieren. Wird dieser Dialog abgebrochen, wird dennoch ein Metamodul Dynamische Formulare erzeugt - dieses enthält jedoch noch keine Formulare oder Formularelemente.

Das Parameter-Fenster dieses Metamoduls beinhaltet den Formulardesigner:

### Abbildung 5.11. Formulardesigner



# Der Designer

## Die GUI

Die Benutzeroberfläche gliedert sich in die folgenden Bestandteile: Oben sieht man in Abbildung 5.11, „Formulardesigner“ dargestellt eine Werkzeugleiste mit einigen Aktionen.

Der restliche Bereich der GUI ist horizontal in zwei Bereiche aufgeteilt: rechts befindet sich der Platz, der der Darstellung der Formulare vorbehalten ist.

Links befindet sich das Dock, in dem verschiedene Hilfsmittel zur Definition und Manipulation der Formulare und Formularelemente gruppiert sind.

## Werkzeugleiste

Die Werkzeugleiste des Formulardesigners

### Tabelle 5.11. Actions in der Werkzeugleiste des Meta-Moduls Dynamische Formulare

Diese Action erzeugt ein neues Formular. Sie zeigt einen Dialog zur Eingabe des Namens an. Dieser ist mit einem Vorschlag vorbelegt. Die Anwendung lässt die mehrfache Vergabe des gleichen Namens für unterschiedliche Formulare innerhalb einer Anwendung nicht zu.

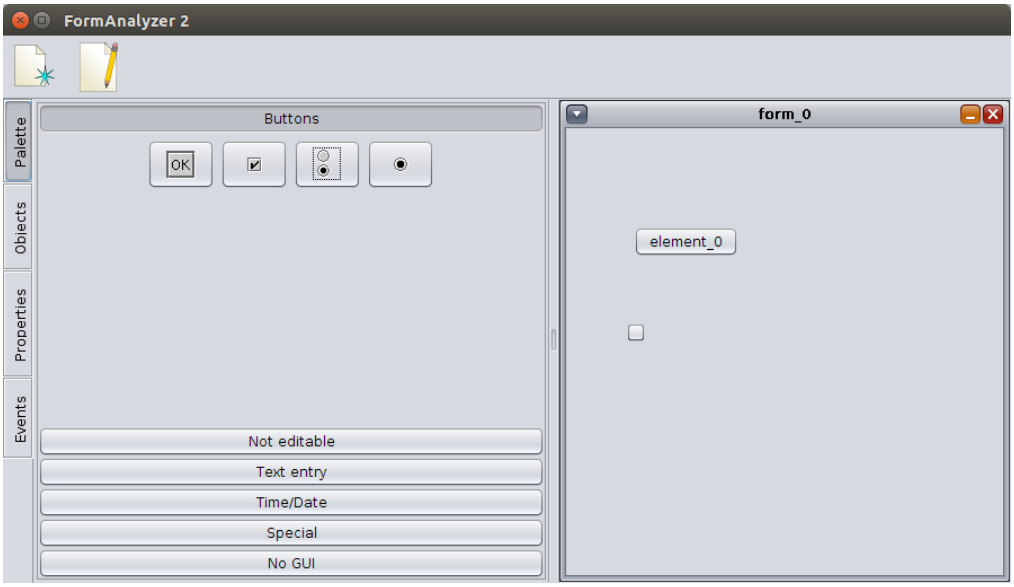
Diese Action schaltet zwischen Laufzeitmodus und Designmodus um: Im Laufzeitmodus agiert die Anwendung normal. Im Designmodus kann man die Eigenschaften und Positionen der einzelnen Formularelemente anpasen. Weiterhin erlaubt es dieser Modus, Events der einzelnen Formularelemente mit Code zu hinterlegen.

## Das Dock

### Die Palette

Ganz oben im Dock findet man die Palette: Darin sind alle Formularelemente in Gruppen gegliedert aufgeführt. Ein Klick auf eines der Elemente und ein anschließender Klick in ein Formular platziert ein solches Formularelement an der entsprechenden Stelle im Formular, nachdem der Anwender entweder den vorgeblendeten Namen akzeptiert oder einen eigenen spezifiziert hat. Formularelemente innerhalb eines Formulars müssen eindeutige Namen haben. Versucht der Anwender, einen Namen anzugeben, der im fraglichen Formular bereits existiert, akzeptiert das System diesen nicht. Im Editiermodus kann man anschließend die Position und Größe des Formularelements anpassen.

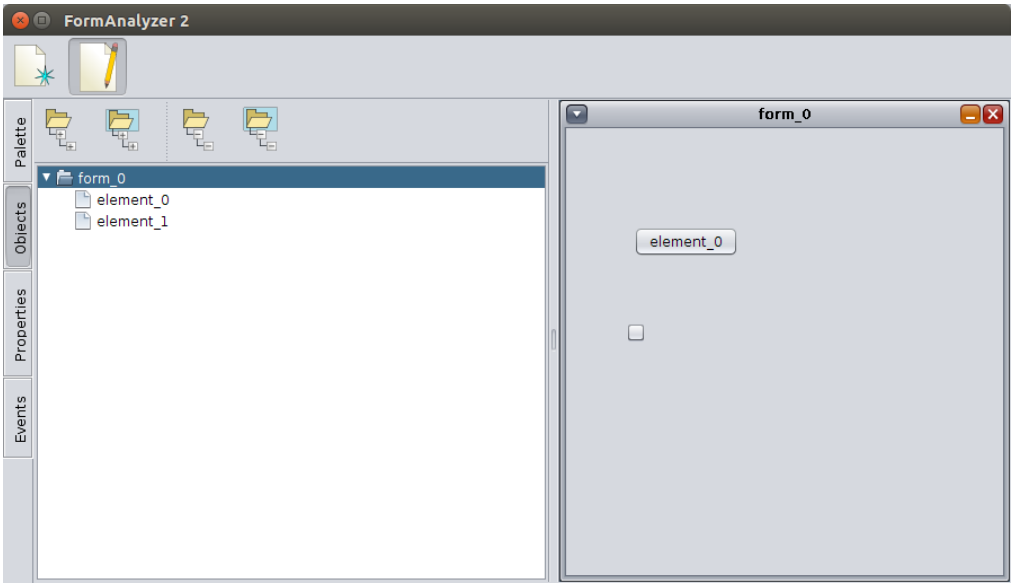
### Abbildung 5.12. Palette




## Der Objektbaum

Im Dock schließt sich direkt unterhalb der Palette der Objektbaum an: ein Klick auf eines der Formulare bringt dieses Formular nach vorn. Ein klick auf ein Formularelement selektiert es.

### Abbildung 5.13.




### Tabelle 5.12. Actions in der Werkzeugleiste des Objektbaumes im Metamodul Dynamische Formulare

| | |
|---|---|
| Alle Knoten anzeigen | Zeigt alle Knoten unter dem selektierten |

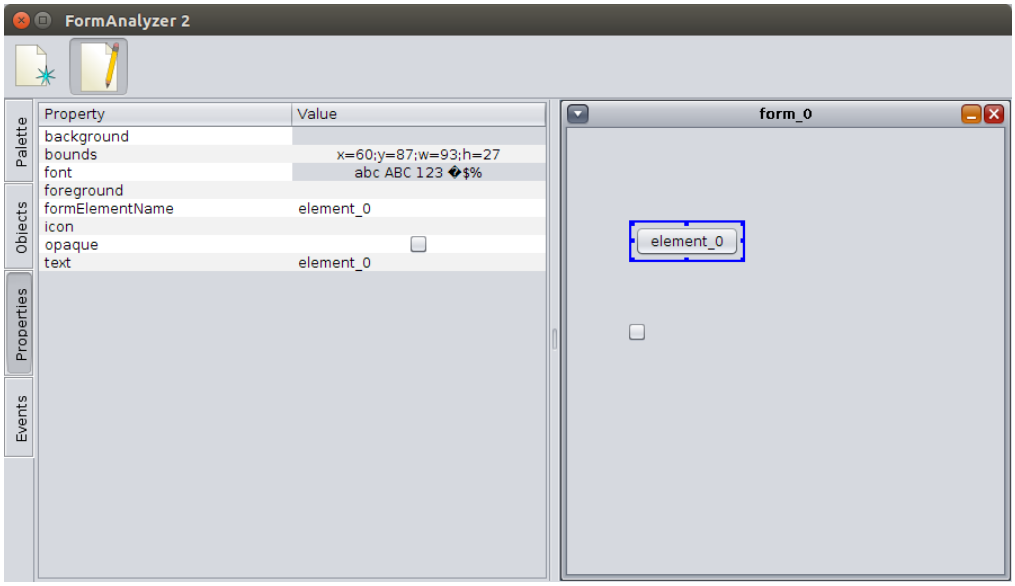
Ausblenden aller Knoten

Ausblenden aller Knoten unterhalb des ausgewählten

## Die Properties

Unterhalb des Objektbaumes wiederum schließt sich der Editor für die Eigenschaften eines Formulars oder Formularelements im Dock an. Die Eigenschaften eines Formulars editiert man, indem man im Editiermodus in das Formular en eine beliebige Stelle klickt, an der sich kein Formularelement befindet. Die Eigenschaften eines Formulaerelements kann man modifizieren, wenn man das fragliche Formularelement im Editiermodus anklickt.

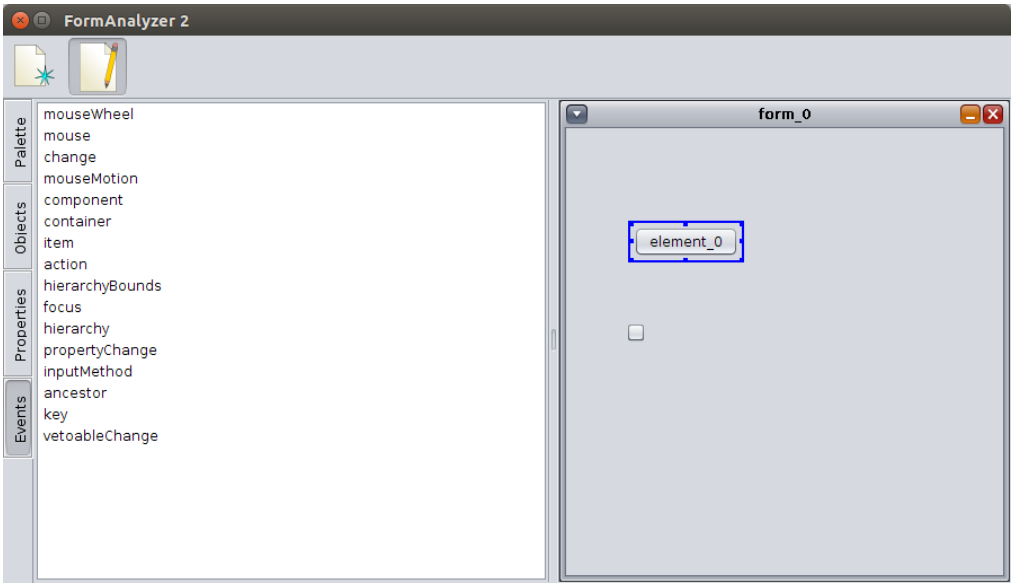
### Abbildung 5.14. Die Properties




## Die Events

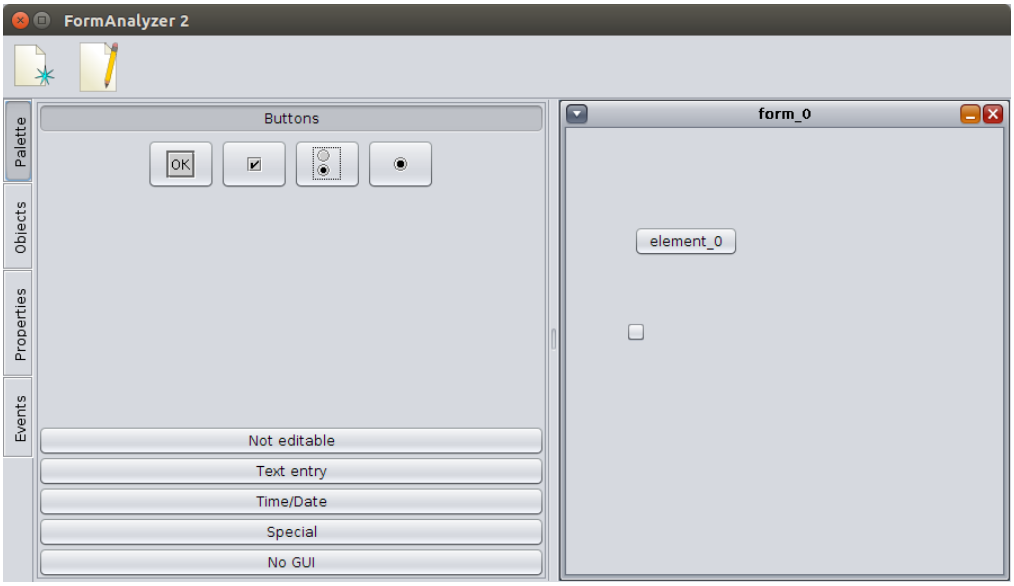
Den Abschluss der Elemente im Dock bildet schließlich die Liste aller Events eines Formulars oder Formularelements. Den Code, der bei Eintreten eines Events ausgeführt wird, editiert man, indem man in der Liste auf den Namen des entsprechenden Events klickt. Daraufhin öffnet sich ein Fenster mit einem Code-Editor.

### Abbildung 5.15. Die Events




## Die Formulare

### Abbildung 5.16. Die Formulare




### Tabelle 5.13. Actions im Kontextmenü der Formulare im Metamodul Dynamische Formulare

Diese Action löscht alle selektierten Formularelements nach nochmaliger Sicherheitsabfrage.

Dieses Untermenü erlaubt es, mehrere Formularelemente relativ zueinander auszurichten. Folgende Möglichkeiten stehen dazu zur Verfügung:

**Tabelle 5.14. Actions zur relativen Anordnung von Formularelementen in Formularen im Metamodul Dynamische Formulare**

| | |
|---|---|
| | Richtet die Module so aus, daß die linken Kanten alle auf einer gedachten Linie liegen |
| | Richtet die Module so aus, daß die rechten Kanten alle auf einer gedachten Linie liegen |
| | Richtet die Module so aus, daß die oberen Kanten alle auf einer gedachten Linie liegen |
| | Richtet die Module so aus, daß die unteren Kanten alle auf einer gedachten Linie liegen |
| | Richtet die Module so aus, daß die horizontalen Mitten alle auf einer gedachten Linie liegen |
| | Richtet die Module so aus, daß die vertikalen Mitten alle auf einer gedachten Linie liegen |
| | Verteilt die Module so, daß der horizontale Zwischenraum zwischen je zwei benachbarten überall gleich ist |
| | Verteilt die Module so, daß der vertikale Zwischenraum zwischen je zwei benachbarten überall gleich ist |

# Kapitel 6. Rollen und Rechte

## Rechte

Ein Anwender, der mit dWb+ arbeitet, hat zunächst einmal keinerlei Rechte. Diese Formulierung bezieht sich darauf, dass zum Beispiel seine Möglichkeiten bei der Gestaltung des Workspace (Hinzufügen/ Entfernen von Modulen) und bei der Analyse der verarbeiteten Daten eingeschränkt sind.

Die einzelnen Rechte sind im Folgenden aufgelistet:

**de.netsysit.ui.moduleworkspace.security.permissions.InsertModulePermission.** Dieses Recht bezieht sich auf das Hinzufügen neuer Module zu Workspaces. Dabei ist es egal, ob das Modul per Drag'n'Drop aus dem Modulbaum kommt, ob ein oder mehrere Module aus der Zwischenablage eingefügt oder ein Workspace geladen werden soll: Jede Handlung, die dazu führen würde, dass ein neues Modul auf einem Workspace auftaucht, benötigt zur erfolgreichen Durchführung dieses Recht.

**de.netsysit.ui.moduleworkspace.security.permissions.EstablishLinkPermission.** Dieses Recht bezieht sich auf das Hinzufügen neuer Verbindungen zu Workspaces. Dabei ist es egal, ob die Verbindung per Drag'n'Drop von Slots erstellt wird, ob ein Workspacefragment mit Verbindungen aus der Zwischenablage eingefügt oder ein Workspace geladen werden soll: Jede Handlung, die dazu führen würde, dass eine neue Verbindung auf einem Workspace auftaucht, benötigt zur erfolgreichen Durchführung dieses Recht.

**de.netsysit.dataflowframework.prg.security.permissions.InspectParamPermission.** Dieses Recht bezieht sich auf die Inspektion von Daten durch dynamische Tooltips an den Output-Slots mittels StateUpdater. Besitzt man dieses Recht nicht, erscheinen nur die normalen Tooltips, die entweder eine Beschreibung der entsprechenden Property oder ihren Typ enthalten.

**de.netsysit.dataflowframework.prg.security.permissions.ShowParamPanelPermission.** Dieses Recht bezieht sich auf die Möglichkeit, die Konfiguration einzelner Module zu ändern. Besitzt man dieses Recht nicht, lassen sich die Parameterdialoge nicht öffnen und man kann die Konfiguration der Module nicht anpassen. Auch wenn ein Workspace geladen wird, der geöffnete Parameterdialoge enthält, werden diese nicht wieder hergestellt, wenn der Anwender nicht über dieses Recht verfügt.

**de.netsysit.ui.moduleworkspace.security.permissions.LoadWorkspacePermission.** Dieses Recht wird benötigt, um Workspaces laden zu können. Dies betrifft nicht nur das Öffnen eines neuen sondern auch das Importieren und das Importieren als Gruppe ebenso wie das Nutzen eines im Workspace-Manager gespeicherten Workspace.

**de.netsysit.ui.moduleworkspace.security.permissions.SaveWorkspacePermission.** Dieses Recht wird benötigt, um Workspaces speichern zu können. Das betrifft auch das Speichern im Workspace-Manager.

**de.netsysit.ui.moduleworkspace.security.permissions.RemoveModulePermission.** Dieses Recht wird benötigt, um Module entfernen zu können. Das betrifft sowohl das Entfernen einzelner Module, ganzer Modulgruppen sowie das Ausschneiden einzelner Module und ganzer Modulgruppen. Besitzt der Anwender dieses Recht nicht, wird die Operation Workspace öffnen zu Workspace importieren, da die bereits vorhandenen Module nicht entfernt werden können.

**de.netsysit.ui.moduleworkspace.security.permissions.RemoveLinkPermission.** Dieses Recht bezieht sich auf das Entfernen von Verbindungen zwischen zwei Modulen. Wenn ein Modul vom Workspace entfernt wird, werden trotzdem alle Verbindungen entfernt, die an diesem Modul enden und von diesem Modul ausgehen.

**de.netsysit.dataflowframework.prg.security.permissions.ToggleLinkActivityPermission.** Dieses Recht bezieht sich auf das Umschalten der Aktivität von Verbindungen. Besitzt der Anwender dieses Recht nicht, ist er nicht in der Lage, die Datenübertragung über einzelne Verbindungen temporär abzuschalten, beziehungsweise diese Abschaltung wieder rückgängig zu machen. Alle Verbindungen erhalten nach Einladen eines Workspace ungeachtet dieses Rechtes ihren zuvor gespeicherten Status.

**de.netsysit.dataflowframework.prg.security.permissions.DynamicSVGPermission.** Dieses Recht bezieht sich auf die Benutzung vom Meta-Modulen vom Typ SVG-Dokument. Nur Anwender, die über dieses Recht verfügen, können solche Meta-Module nutzen.

**de.netsysit.dataflowframework.prg.security.permissions.AviatorPermission.** Dieses Recht bezieht sich auf die Benutzung vom Meta-Modulen vom Typ Aviator. Nur Anwender, die über dieses Recht verfügen, können solche Meta-Module nutzen.

Über diese speziellen Permissions hinaus existieren natürlich noch die normalen Permissions in Java. Diese können ebenfalls Anwendern abhängig von ihrer durch das Login festgelegten Identität zu- oder aberkannt werden.

# Konfiguration

Die Konfiguration des Rollen- und Rechte-Systems geschieht wie bei allen anderen Java-Anwendungen: Die Permissions für die einzelnen Identitäten oder Rollen werden in einer Policy-Datei festgelegt. Die Java-Laufzeitumgebung wird beim Start so konfiguriert, dass sie einen Security Manager benutzt. Außerdem muss der Laufzeitumgebung darüber hinaus noch der Ort und Name der Policy-Datei und der Name der Datei zur Konfiguration der Login Configuration mitgeteilt werden. Jeweils ein Beispiel für beide Dateien findet sich im Anhang B, *Beispieldateien zum Rechtemanagement* in „Policy“ und „Login Configuration“

# Anhang A. Verzeichnis-Layout

Die Anwendung dWb+ benutzt ein Datenverzeichnis, um verschiedene benötigte Daten zu speichern. Das Layout des Dateisystems innerhalb dieses Verzeichnisses wird im Folgenden erklärt, wobei die Gliederung der Dokumentation der Gliederung des Verzeichnisses in Unterverzeichnisse entspricht.

Dieses Verzeichnis befindet sich standardmäßig im Home-Verzeichnis des jeweiligen Anwenders. Der Ort kann aber beliebig gewählt werden, indem man eine System-Property setzt, wie in Kapitel 1, *Überblick* in „Programmstart“ beschrieben.

## dWb.accessory.dirs

Diese Datei enthält die Definitionen der Lesezeichen aus dem Dateiauswahldialog.

## favmod.xml

Diese Datei enthält Definitionen der bevorzugten Module für die Komponente, die in „Favoriten“ in Kapitel 3, *Dock* beschrieben wird.

## scratchpad.xml

Diese Datei enthält die Definitionen der Workspace-Fragmente für die Komponente, die in „Scratch Manager“ in Kapitel 3, *Dock* beschrieben wird.

## macros.template

Diese Datei Fragmente von Java-Quelltexten, die als Macros in selbsterstellten Skripten benutzt werden können.

Diese Funktionalitäten befindet sich zur Zeit noch in der Entwicklung - es existiert noch keine Möglichkeit, Macros zu editieren.

## aviator

Dieses Verzeichnis enthält die Daten, die das in „Aviator“ in Kapitel 5, *Meta-Module* beschriebene Meta-Modul zum Funktionieren benötigt.

### medium

Dieses Verzeichnis enthält größere Bitmap-Versionen der Vektorgraphiken, die vom Meta-Modul Aviator benutzt werden. Diese Graphiken kommen zum Beispiel im Konfigurationsdialog zum Einsatz. Der Anwender muss diese Graphiken nicht selbst erzeugen - fehlen sie, generiert dWb+ sie automatisch und speichert sie in diesem Verzeichnis zur späteren Verwendung ab.

### small

Dieses Verzeichnis enthält kleinere Bitmap-Versionen der Vektorgraphiken, die vom Meta-Modul Aviator benutzt werden. Diese Graphiken kommen zum Beispiel im Konfigurationsdialog zum Einsatz. Der

Anwender muss diese Graphiken nicht selbst erzeugen - fehlen sie, generiert dWb+ sie automatisch und speichert sie in diesem Verzeichnis zur späteren Verwendung ab.

## svg

Dieses Verzeichnis enthält die Vektorgraphiken, die vom Meta-Modul Aviator benutzt werden.

# lib

Diese Komponente enthält Libraries (.dll oder .so), die eventuell von Modulen benötigt werden.

# moduleEmblems

Dieses Verzeichnis nimmt alle Graphiken auf, die als Embleme für Module benutzt werden sollen - vergleiche dazu „Modulmenü“ [46].

# modules

Dieses Verzeichnis nimmt alle Jar-Dateien, die Module enthalten auf.

# remoting

Dieses Verzeichnis nimmt alle Jar-Dateien auf, die für die Verlagerung von Funktionalitäten auf andere Rechner benötigt werden - siehe auch Kapitel 21, *Verteiltes Arbeiten (Remoting)*.

# scripted

Dieses Verzeichnis nimmt alle Skripted Module auf.

# services

Dieses Verzeichnis enthält Jar-Dateien, die Implementierungen für BeanContext-Services darstellen. Dabei enthalten die Jar-Dateien alle Klassen, die mittelbar oder unmittelbar für die Implementierung eines BeanContext-Services benötigt werden. Die Anwendung bindet diese Klassen selbsttätig in den Klassenpfad ein. Die Interfaceklassen müssen im zentralen Klassenpfad zu finden sein, damit die Module darauf zugreifen können.

Die Dienste, die bereits standardmäßig vom dWb+ zur Verfügung gestellt werden, sind im Programmierhandbuch in Anhang B, *BeanContext Services* aufgeführt. Zwei dieser Dienste werden über den hier beschriebenen Mechanismus eingebunden: Nach der Installation findet sich die Datei core_services.jar in diesem Verzeichnis: Dort kann man an praktischen Beispiele die Gestaltung des Inhaltes des Verzeichnisses META-Inf/services nachvollziehen.

Die Jar-Dateien werden beim Start der Anwendung analysiert und alle gefundenen Services in der Anwendung publiziert. Dafür muss im Verzeichnis META-INF der Jar-Datei ein Verzeichnis services vorhanden sein. Die darin enthaltenen Dateien legen fest, welche Services durch welchen ServiceProvider angeboten werden. Der Name einer solchen Datei ist dabei der Klassenname (voll qualifiziert) des Interfaces und der Inhalt der Datei ist der voll qualifizierte Klassenname des ServiceProviders.

Die Interfaceklasse wird mit dem normalen ClassLoader der Anwendung geladen, die ServiceProvider-Klasse mit einem eigens dafür geschaffenen, der die Jar-Datei zum Suchpfad für Klassen hinzufügt.

# stateupdaters

Dieses Verzeichnis enthält Jar-Dateien, die die zur in „Tooltips für Outputs“ in Kapitel 4, *Module* beschriebenen Anpassung der Tooltips an den Modulausgängen benötigten Komponenten enthalten.

# workspaces

Dieses Verzeichnis enthält die Workspaces, die über die in „Workspaces“ in Kapitel 3, *Dock* beschriebene Komponente verwaltet werden.

# conversionPresets.xml

Diese Datei enthält BeanShellFragmente zur Konversion nicht unmittelbar miteinander kompatibler Datentypen durch BeanSehllConverterModule wie zum Beispiel in „Überblick“ in Kapitel 4, *Module* oder in „Datenflussprogrammierung“ in „Matching“ beschrieben. Diese Datei folgt der Syntax für Java-Properties im XML-Format. Ein Beispiel ist hier angegeben:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd">
<properties>
<comment>Default conversion rules for automatically
inserted BeanShellConversionModules</comment>
<entry key="java.lang.Boolean::java.lang.Number">
_output_=_input_!=null?(_input_.booleanValue()?1:0):null;
</entry>
<entry
key="de.netsysit.dataflowframework.modules.common.ClockTypes::java.lang.Number">
_output_=null;
if(_input_!=null)
{
switch(_input_)
{
case CLOCK: _output_=1;break;
case POSITIVE_SPKIE: _output_=2;break;
case NEGATIVE_SPKIE: _output_=3;break;
default: _output_=0;
}
}
</entry>
<entry key="java.lang.Number::java.util.Date">
_output_=_input_!=null?new java.util.Date(_input_.longValue()):null;
</entry>
<entry key="java.lang.Integer::java.util.Calendar">_output_=null;
if(_input_!=null)
{
_output_ = Calendar.getInstance();
_output_.setTimeInMillis(_input_.longValue());
}</entry>
</properties>
```

Wichtig ist hierbei noch zu erwähnen, dass die Schlüssel aus dem voll qualifizierten Namen der Klassentypen des Aus- und des Eingangs verbunden durch "::" gebildet werden. Der Schlüssel muss nicht genau passen - beim Typen des Ausgangs besteht Spielraum - es wäre etwa möglich, eine Konversion für den Ausgangstypen `java.lang.Number` zu spezifizieren - diese würde (sofern der Typ des Eingangs passt) auch dann benutzt, wenn der Ausgangstyp `java.lang.Integer` ist, es aber für diesen und den jeweiligen Ausgangstyp keine explizite Konversion gibt. In der oben angegebenen Konfiguration wäre die Konversion unter dem Schlüssel `java.lang.Number::java.util.Date` ein Beispiel dafür.

# Anhang B. Beispieldateien zum Rechtemanagement

## Policy

```
/** Java 2 Access Control Policy for the dWb+ Application **/

/**
 ** Diese Rechte werden auf jeden Fall für den Betrieb der Anwendung
 ** benötigt - sie sind daher nicht an eine bestimmte Identität gebunden
 **/

grant  {
   permission java.awt.AWTPermission
    "accessClipboard";
   permission java.awt.AWTPermission
    "showWindowWithoutWarningBanner";
   permission java.awt.AWTPermission
    "accessEventQueue";
   permission java.awt.AWTPermission
    "listenToAllAWTEvents";
   permission java.awt.AWTPermission
    "accessSystemTray";
   permission java.io.FilePermission
    "<<ALL FILES>>",  "read";
   permission java.io.FilePermission
    "<<ALL FILES>>",  "write";
   permission java.io.FilePermission
    "<<ALL FILES>>",  "delete";
   permission java.lang.RuntimePermission
    "createClassLoader";
   permission java.lang.RuntimePermission
    "modifyThread";
   permission java.lang.RuntimePermission
    "setContextClassLoader";
   permission java.lang.RuntimePermission
    "accessDeclaredMembers";
   permission java.lang.RuntimePermission
    "createClassLoader";
   permission java.lang.RuntimePermission
    "modifyThread";
   permission java.lang.RuntimePermission
    "setContextClassLoader";
   permission java.lang.RuntimePermission
    "accessClassInPackage.sun.swing";
   permission java.lang.RuntimePermission
    "loadLibrary.rxtxSerial";
   permission java.lang.RuntimePermission
    "getenv.XDG_CONFIG_HOME";
```

```
 permission java.lang.RuntimePermission
  "getClassLoader";
 permission java.lang.RuntimePermission
  "preferences";
 permission java.lang.reflect.ReflectPermission
  "suppressAccessChecks";
 permission javax.security.auth.AuthPermission
  "modifyPrincipals";
 permission javax.security.auth.AuthPermission
  "createLoginContext";
 permission javax.security.auth.AuthPermission
  "doAsPrivileged";
 permission java.util.PropertyPermission
  "os.name", "read";
 permission java.util.PropertyPermission
  "java.specification.version", "read";
 permission java.util.PropertyPermission
  "user.home", "read";
 permission java.util.PropertyPermission
  "java.home", "read";
 permission java.util.PropertyPermission
  "user.dir", "read";
 permission java.util.PropertyPermission
  "javabeansframework.java.ui.File.updateOnFocusLost", "read";
 permission java.util.PropertyPermission
  "javabeansframework.java.io.File.Filechooserbuttonimg", "read";
 permission java.util.PropertyPermission
  "javabeansframework.java.net.URL.updateOnFocusLost", "read";
 permission java.util.PropertyPermission
"de.netsysit.model.table.CachingJDBCReadOnlyTableModel.MaxBinaryArraySize",
  "read";
 permission java.util.PropertyPermission
"de.netsysit.model.table.CachingJDBCReadOnlyTableModel.MaxBinaryArraySize",
  "write";
 permission java.util.PropertyPermission
"de.netsysit.ui.filechooser.TextPreview.MaxPreviewCharacters",
  "read";
 permission java.util.PropertyPermission
"de.netsysit.ui.filechooser.TextPreview.MaxPreviewCharacters",
  "write";
 permission java.util.PropertyPermission
  "jasper.reports.compiler.class", "write";
 permission java.util.PropertyPermission
  "debug", "read";
 permission java.util.PropertyPermission
  "sQLshell.crippled", "read";
 permission java.util.PropertyPermission
  "trace", "read";
 permission java.util.PropertyPermission
  "localscoping", "read";
 permission java.util.PropertyPermission
  "outfile", "read";
 permission java.util.PropertyPermission
  "org.apache.xerces.xni.parser.XMLParserConfiguration", "read";
```

```
permission java.util.PropertyPermission
 "sun.awt.erasebackgroundonresize", "read";
permission java.util.PropertyPermission
 "sun.awt.noerasebackground", "read";
permission java.util.PropertyPermission
 "javax.xml.parsers.SAXParserFactory", "read";
permission java.util.PropertyPermission
 "org.apache.commons.logging.LogFactory", "read";
permission java.util.PropertyPermission
 "org.apache.commons.logging.log", "read";
permission java.util.PropertyPermission
 "java.class.path", "read";
permission java.util.PropertyPermission
 "net.sf.jasperreports.properties", "read";
permission java.util.PropertyPermission
 "jasper.reports.compiler.class", "read";
permission java.util.PropertyPermission
 "jasper.reports.compile.xml.validation", "read";
permission java.util.PropertyPermission
 "jasper.reports.export.xml.validation", "read";
permission java.util.PropertyPermission
 "jasper.reports.compile.keep.java.file", "read";
permission java.util.PropertyPermission
 "jasper.reports.compile.temp", "read";
permission java.util.PropertyPermission
 "jasper.reports.compile.class.path", "read";
permission java.util.PropertyPermission
 "nonBatchMode", "read";
permission java.util.PropertyPermission
 "useJavaUtilZip", "read";
permission java.util.PropertyPermission
 "checkZipIndexTimestamp", "read";
permission java.util.PropertyPermission
 "java.endorsed.dirs", "read";
permission java.util.PropertyPermission
 "sun.boot.class.path", "read";
permission java.util.PropertyPermission
 "java.ext.dirs", "read";
permission java.util.PropertyPermission
 "file.encoding", "read";
permission java.util.PropertyPermission
 "sun.jnu.encoding", "read";
permission java.util.PropertyPermission
 "user.language", "read";
permission java.util.PropertyPermission
 "user.country", "read";
permission java.util.PropertyPermission
 "user.variant", "read";
permission java.util.PropertyPermission
 "user.name", "read";
permission java.util.PropertyPermission
 "javax.xml.parsers.DocumentBuilderFactory", "read";
permission java.util.PropertyPermission
 "plugins.dir", "read";
```

```
    permission java.util.PropertyPermission
     "gnu.io.*", "read";
    permission java.util.PropertyPermission
     "dwb.config.*", "read";
    permission java.util.PropertyPermission
     "org.mortbay.*", "read";
    permission java.util.PropertyPermission
     "DynamicSVG.*", "read";
    permission java.util.PropertyPermission
     "DynamicSVG.*", "write";
    permission java.util.PropertyPermission
     "VERBOSE", "read";
    permission java.util.PropertyPermission
     "IGNORED", "read";
    permission java.util.PropertyPermission
     "DEBUG", "read";
    permission java.util.PropertyPermission
     "ISO_8859_1", "read";
    permission java.util.PropertyPermission
     "SVGChooserPanel.updateBitmaps", "read";
    permission java.util.PropertyPermission
     "dWb.sources.basedir", "read";
    permission java.util.PropertyPermission
     "dWb.sources.i18nbasename", "read";
    permission java.util.PropertyPermission
     "dWb.service.ApplicationServer.port", "read";
    permission java.net.SocketPermission
     "*", "accept";
    permission java.net.SocketPermission
     "*", "resolve";
    permission java.net.SocketPermission
     "*", "connect";
    permission java.lang.RuntimePermission
     "accessClassInPackage.sun.misc";
    permission java.lang.RuntimePermission
     "modifyThreadGroup";
    permission java.lang.RuntimePermission
     "getProtectionDomain";
    permission java.lang.RuntimePermission
     "accessClassInPackage.sun.util.resources";
    permission java.lang.RuntimePermission
     "accessClassInPackage.sun.text.resources";
    permission java.util.PropertyPermission
     "javax.xml.parsers.DocumentBuilderFactory", "read";
    permission java.lang.RuntimePermission
     "accessClassInPackage.sun.util.resources";
    permission java.util.logging.LoggingPermission
     "control";
    permission java.lang.RuntimePermission
     "loadLibrary.jmvfw";
  /*ActiveMQ JMS MQ*/
    permission java.util.PropertyPermission
     "dWb.MQ.Context.INITIAL_CONTEXT_FACTORY", "read";
    permission java.net.NetPermission
```

```
     "getCookieHandler";
    permission java.net.NetPermission
     "getProxySelector";
    permission java.net.NetPermission
     "getResponseCache";
    permission java.util.PropertyPermission
     "activemq.idgenerator.port", "read";
    permission java.util.PropertyPermission
     "AMQ_HOST", "read";
    permission java.util.PropertyPermission
     "AMQ_PORT", "read";
    permission java.util.PropertyPermission
     "BROKER_BIND_URL", "read";
    permission java.util.PropertyPermission
     "http.proxyHost", "read";
    permission java.util.PropertyPermission
     "org.apache.activemq.AMQ_HOST", "read";
    permission java.util.PropertyPermission
     "org.apache.activemq.AMQ_PORT", "read";
    permission java.util.PropertyPermission
     "org.apache.activemq.BROKER_BIND_URL", "read";
    permission java.util.PropertyPermission
     "org.apache.activemq.transport.AbstractInactivityMonitor.keepAliveTime",
     "read";
    permission java.util.PropertyPermission
     "proxyHost", "read";
    permission java.util.PropertyPermission
     "socksProxyHost", "read";
/*JMX*/
    permission javax.management.MBeanServerPermission
     "createMBeanServer";
    permission javax.management.MBeanTrustPermission
     "register";
    permission javax.management.MBeanPermission
     "*", "registerMBean";
 };

/**
 ** Identität mit allen Rechten
 ** Für den Druck wurden die voll qualifizierten Namen abgekürzt.
 ** Beim Einsatz müssen natürlich die voll qualifizierten Namen
 ** verwendet werden.
 **/

grant
 Principal de.netsysit.dataflowframework.prg.security.Principal "t"
{
    permission InsertModulePermission;
    permission EstablishLinkPermission;
    permission InspectParamPermission;
    permission ShowParamPanelPermission;
    permission LoadWorkspacePermission;
    permission SaveWorkspacePermission;
    permission RemoveModulePermission;
```

```
   permission RemoveLinkPermission;
   permission ToggleLinkActivityPermission;
   permission DynamicSVGPermission;
   permission AviatorPermission;
   permission java.io.FilePermission "<<ALL FILES>>",  "execute";
/*permission java.security.AllPermission;
*/};

/**
 ** Identität mit eingeschränkten Rechten
 ** Für den Druck wurden die voll qualifizierten Namen abgekürzt.
 ** Beim Einsatz müssen natürlich die voll qualifizierten Namen
 ** verwendet werden.
 **/

grant
 Principal de.netsysit.dataflowframework.prg.security.Principal "c"
{
   permission InsertModulePermission;
   permission EstablishLinkPermission;
   permission ShowParamPanelPermission;
   permission LoadWorkspacePermission;
   permission RemoveModulePermission;
   permission RemoveLinkPermission;
};
```

# Login Configuration

Diese Konfiguration ist unsicher! Hierbei wird kein Passwort getestet! Bitte im operativen Einsatz unbedingt ändern!

```
/** Login Configuration for the dWb+ Application **/

dWb {
   de.netsysit.dataflowframework.prg.security.LoginModule required debug=true;
};
```

# Anwenderhandbuch Visualisierungs-Client

**Jürgen Key**

# Anwenderhandbuch Visualisierungs-Client

Jürgen Key

# Inhaltsverzeichnis

1. Überblick ..... 1
Anwendung ..... 1
Programmstart ..... 1
Weiterführende Literatur ..... 2

# Kapitel 1. Überblick

## Anwendung

Diese Anwendung erlaubt es, die Fähigkeiten des dWb+ insbesondere mit Blick auf die Visualisierung zu überwachender Key Parameter netzwerkfähig und skalierbar zu nutzen.

Die Anwendung selbst ist in der Lage, verschiedenste SVG-Dokumente zu laden und anzuzeigen. Sie implementiert ein Protokoll, das es gestattet, verschiedene Elemente innerhalb des angezeigten SVG-Dokuments zu modifizieren oder die geladene Datei durch eine andere zu ersetzen. Die modifizierbaren Eigenschaften beinhalten unter anderem:

- Farbe
- Deckkraft
- Sichtbarkeit
- Drehwinkel
- Skalierungsfaktor

Damit ist es möglich, ein SVG-Dokument zu einer interaktiven Datenvisualisierungskomponente aufzuwerten.

Das Protokoll kann über verschiedenste Transportmechanismen bedient werden. Die Anwendung DynamicSVG stellt eine Socketschnittstelle (TCP) zur Verfügung, über die andere Anwendungen mit ihr kommunizieren können. Das ermöglicht es, eine integrierte Visualisierungslösung (Digital Signage) zu installieren, auf der DynamicSVG eine festgelegte SVG-Datei anzeigt und diese Visualisierung dann per Fernsteuerung zu aktualisieren.

Die beiden Meta-Module SVG-Dokument und Aviator - beschrieben im Anwenderhandbuch dWb+ in Kapitel 5, *Meta-Module* in „Aviator“ und „SVG-Dokument“ benutzen diese Schnittstelle. Damit ist es möglich, direkt aus dem visuellen Workflow-Design die Key-Parameter des Prozesses zu überwachen.

Diese Form der verteilten Visualisierung von Prozeßzuständen steht in Konkurrenz zu der Möglichkeit, solche Visualisierung im Browser anzuzeigen. Es ist im jeweiligen Anwendungsfall abzuwägen, ob der Fakt, dass keine zusätzlichen Software-Komponenten installiert werden müssen (Browser-Visualisierung) wichtiger ist, als die höhere Update-Rate, die mit dem dedizierten Visualisierungs-Client möglich wäre.

## Programmstart

Der Start der Anwendung wird wie der jeder anderen Java-Anwendung auch vollzogen. Es existieren einige Parameter, die beim Programmstart spezifiziert werden können/müssen. Diese werden als System-Parameter oder auch System-Properties genannt, spezifiziert. Das bedeutet, dass sie mittels vorangestelltem -D in der Kommandozeile beim Programmstart als Schlüssel=Wert eingefügt werden.

| | |
|---|---|
| DynamicSVG.fullscreen | Diese System-Property entscheidet darüber, ob die Anzeige im Vollbildmodus erfolgen soll (-DDynamicSVG.fullscreen=true) oder nicht (-DDynamicSVG.fullscreen=false). Im Vollbildmodus nimmt das Fenster, in dem die Visualisierung stattfindet, die gesamte zur Verfügung stehende Bildschirmfläche ein. Das Fenster hat in diesem Fall auch keine Dekorationen - damit ist es nicht |

möglich, es zu verkleinern oder in den Hintergrund zu schieben. Wird dieser Parameter beim Start nicht angegeben, wirkt das genauso wie die Angabe -DDynamicSVG.fullscreen=true.

DynamicSVG.repaint.interval — Diese System-Property wird als Zahl interpretiert, die die Zeit (in Millisekunden) darstellt, die zwischen zwei aufeinanderfolgenden Aktualisierungen des Bildschirminhaltes vergeht. Diese Aktualisierungen zeichnen den Inhalt des gesamten Fensters neu. Normalerweise wird die Darstellung der geladenen SVG-Datei nur dann aktualisiert (neu gezeichnet), wenn ein Kommando durch eine Gegenstelle mittels eines korrekten Protokollaufrufs eingeht. Dieser Parameter kann dafür sorgen, dass die Aktualisierung zyklisch ohne externen Reiz ausgeführt wird. Wird dieser Parameter nicht angegeben, wird keine zyklische Aktualisierung durchgeführt.

Der Programmstart kann mit einem oder zwei Parametern erfolgen. Der letzte Parameter ist immer der Port, an dem die Anwendung auf Kommandos aus dem definierten Kommunikationsprotokoll reagieren soll. Werden zwei Parameter angegeben, wird der erste als Dateiname interpretiert. In diesem Fall wird versucht, den Inhalt der Datei als SVG-Graphik zu interpretieren und darzustellen. Wird nur ein Parameter angegeben, wird ein leeres SVG-Dokument erzeugt und angezeigt.

# Weiterführende Literatur

Weiterführende Informationen zum genutzten Protokoll und den verfügbaren Aktionen, die über die Kommandoschnittstelle zur Transformation des geladenen SVG-Dokumentes zur Verfügung stehen entnehmen Sie bitte dem "Technischen Handbuch DMCC" und der Beschreibung der API "Application Programming Interface (API) DMCC".

# Anwenderhandbuch DynamicSVG-Proxy

**Jürgen Key**

# Anwenderhandbuch DynamicSVG-Proxy

Jürgen Key

# Inhaltsverzeichnis

1. Überblick ..... 1
Einsatz ..... 1
Programmstart ..... 1
Servlets ..... 1
XHTMLSnapshotServlet ..... 1
JavaScriptServlet ..... 2
ProxyRotationSpecServlet ..... 2
ProxyOnOffSpecServlet ..... 2
ProxyScaleSpecServlet ..... 2
ProxyTranslationSpecServlet ..... 3
ProxyGraphSpecServlet ..... 3
FSImageProviderServlet ..... 3
ProxyCurrentValuesUpdaterServlet ..... 3

# Kapitel 1. Überblick

## Einsatz

Diese Anwendung ist eine Webanwendung zur Benutzung als Erweiterung für Standard-Servletcontainer wie etwa Jetty oder Tomcat.

Diese Anwendung erweitert dWb+. Die Anwendung dWb+ ist in der Lage, Servlets und andere Resourcen in der Art eines Servletcontainers oder Webservers im Internet über das Protokoll HTTP zur Verfügung zu stellen. Diese Inhalte können dann zum Beispiel mit einem Internet-Browser betrachtet werden.

dWb+ verfügt neben den normalen Funktionsmodulen über sogenannte Meta-Module. Eines dieser Meta-Module, das im Anwenderhandbuch dWb+ in Kapitel 5, *Meta-Module* im „Aviator“ beschrieben wird, nutzt diese Funktionalität, um vom Anwender konfigurierte Cockpits zur Informationsdarstellung dem Anwender im Webbrowser zu präsentieren.

Diese Vorgehensweise trifft aber eventuell nicht den Geschmack des Konstrukteurs des jeweiligen Workspace. Ein Grund dafür könnte sein, dass die Visualisierung von vielen Anwendern per Browser angesehen wird. Das würde dazu führen, dass dWb+ neben seiner eigentlichen Aufgabe - der Berechnung des konfigurierten Workspaces - viel Rechenleistung für die Befriedigung der Anforderungen der Browser aufwenden muss.

Das ist der Punkt, an dem der Proxy für DynamicSVG die Last der Bearbeitung der Anfragen von dWb+ abhalten kann. Grundsätzlich funktioniert der Einsatz des Proxy wie folgt: DynamicSVG versendet beim Start der Publikation des virtuellen Cockpits den Inhalt desselben als SVG-Graphik an den Proxy. Damit können sich alle Interessierten den Status der Graphik und damit des Cockpits und der zu überwachenden Parameter vom Proxy holen. Damit der Status der Graphik und der Parameter immer aktuell ist, sendet dWb+ alle Änderungen der zu überwachenden Signale ebenfalls an den Proxy, der sie auf Anfrage an die Browser weitergibt. Damit muss die Rechenleistung für die Kommunikation mit den Browsern von dem Rechner aufgebracht werden, auf dem der DynamicSVG-Proxy ausgeführt wird und der Rechner, auf dem die Anwendung dWb+ ausgeführt wird, ist davon entlastet.

## Programmstart

Der Programmstart gestaltet sich einfach - die Anwendung kommt als WebARchiv (Endung .war). Dieses Archiv wird einfach an die korrekte Stelle (abhängig vom eingesetzten Servlet-Container) kopiert und dieser dann neu gestartet - fertig!

Gegebenenfalls müssen weitere Einstellungen vorgenommen werden - zum Beispiel dann, wenn der Servlet-Container als nachgeordnete Instanz eines Web-Servers konfiguriert ist - Stichworte hier zum Beispiel apache httpd und mod_jk. Die in einem solchen Szenario notwendigen zusätzlichen Schritte würden den Rahmen dieses Dokumentes sprengen.

## Servlets

Die Anwendung besteht aus mehreren Servlets, deren Anwendungszweck im folgenden kurz umrissen wird:

### XHTMLSnapshotServlet

Dieses Servlet dient der Auslieferung der XHTML-Seite, in die die SVG-Graphik eingebettet ist. Es benutzt ein Velocity-Template, das nach Ersetzung der Platzhalter eine gültige XHTML-Seite ergibt, die

dann an die Browser ausgegeben wird. Nähere Informationen zum Format dieses Templates finden sich im Pogrammierhandbuch dWb+ im Kapitel 28, *Aviator Templates*.

## Methoden

POST Benutzt, um die Templates auf den Proxy zu laden, so dass sie für den Abruf mittels Browser bereitstehen. Nach dem Upload werden die Platzhalter ersetzt, so dass die fertiggestellte XHTML-Seite anschließend nur noch an die Browser ausgegeben werden muss.

GET Die Methode gibt die XHTML-Seite an die Browser aus.

# JavaScriptServlet

## Methoden

GET Dieses Servlet liefert die für die Funktionsweise von DMCC im Browser nötigen JavaScript-Fragmente an die Browser aus.

# ProxyRotationSpecServlet

Dieses Servlet empfängt von der Steuereinheit die Angaben zu Parametern für die semantischen Marker zum Drehen eines Elements. Im Browser erfolgt keine Analyse der SVG-Datei auf semantische Marker und deren Parameter - Diese Informationen kommen von der zentralen Steuerungsinstanz und werden bei Bedarf an den Browser gesendet.

## Methoden

POST Empfang und Speichern der Parameter mit der XML-ID des betroffenen Elements

GET Senden der Parameter für ein Element an den anfragenden Browser

# ProxyOnOffSpecServlet

Dieses Servlet empfängt von der Steuereinheit die Angaben zu Parametern für die semantischen Marker zum Umschalten der Sichtbarkeit. Im Browser erfolgt keine Analyse der SVG-Datei auf semantische Marker und deren Parameter - Diese Informationen kommen von der zentralen Steuerungsinstanz und werden bei Bedarf an den Browser gesendet.

## Methoden

POST Empfang und Speichern der Parameter mit der XML-ID des betroffenen Elements

GET Senden der Parameter für ein Element an den anfragenden Browser

# ProxyScaleSpecServlet

Dieses Servlet empfängt von der Steuereinheit die Angaben zu Parametern für die semantischen Marker zum Umschalten der Sichtbarkeit. Im Browser erfolgt keine Analyse der SVG-Datei auf semantische Marker und deren Parameter - Diese Informationen kommen von der zentralen Steuerungsinstanz und werden bei Bedarf an den Browser gesendet.

## Methoden

POST Empfang und Speichern der Parameter mit der XML-ID des betroffenen Elements

GET Senden der Parameter für ein Element an den anfragenden Browser

# ProxyTranslationSpecServlet

Dieses Servlet empfängt von der Steuereinheit die Angaben zu Parametern für die semantischen Marker zum Verschieben der Position eines Elementes. Im Browser erfolgt keine Analyse der SVG-Datei auf semantische Marker und deren Parameter - Diese Informationen kommen von der zentralen Steuerungsinstanz und werden bei Bedarf an den Browser gesendet.

## Methoden

POST Empfang und Speichern der Parameter mit der XML-ID des betroffenen Elements

GET Senden der Parameter für ein Element an den anfragenden Browser

# ProxyGraphSpecServlet

Dieses Servlet empfängt von der Steuereinheit die Angaben zu Parametern für die semantischen Marker zum Darstellen von numerischen Werten als Zeitreihe oder Diagramm. Im Browser erfolgt keine Analyse der SVG-Datei auf semantische Marker und deren Parameter - Diese Informationen kommen von der zentralen Steuerungsinstanz und werden bei Bedarf an den Browser gesendet.

## Methoden

POST Empfang und Speichern der Parameter mit der XML-ID des betroffenen Elements

GET Senden der Parameter für ein Element an den anfragenden Browser

# FSImageProviderServlet

Dieses Servlet speichert die SVG-Graphik in ihrem Urzustand zwischen und gibt sie an anfragende Browser weiter. Der Ablauf bei der Nutzung der Web-Visualisierung mittels DMCC ist:

- Laden der beinhaltenden XHMTL-Seite
- Im onload-Event Laden der SVG-Daten und Einfügen in den DOM-Baum der XHTML-Seite
- Zyklisch abfragen, ob sich die Daten für die Element mit semantischen Markern geändert haben und falls ja - die betroffenen Elemente im DOM-Baum entsprechend des zugehörigen semantischen Markers manipulieren

## Methoden

POST Hochladen und Speichern der SVG-Graphik im Urzustand

GET Ausliefern der Graphik im Urzustand

# ProxyCurrentValuesUpdaterServlet

Dieses Servlet ist verantwortlich dafür, die Daten, die die Grundlage der Manipulation der Elemente mit semantischen Markern im SVG-Dokument bilden, entgegenzunehmen und für die Auslieferung an anfragende Browser zwischenzuspeichern.

## Methoden

POST Empfängt Daten für alle Elemente, die mit semantischen Markern ausgestattet sind und für die die zugeordneten Datenproduzenten bereits Daten geliefert haben

POST Liefert den aktuellen Datenstand an anfragende Browser.

# Architektur dWb+

**Jürgen Key**

# Architektur dWb+

Jürgen Key

# Inhaltsverzeichnis


1. Überblick ..... 1
    Datenflussprogrammierung ..... 1
    Gliederung ..... 1
    Programmiermodell ..... 2
        Quelltextbearbeitung ..... 2
        Änderungen werden nach Neustart wirksam ..... 2
        Änderungen werden zur Laufzeit wirksam ..... 2
2. Runtime ..... 3
    Workspace ..... 3
        Logik ..... 3
        Benutzerschnittstelle ..... 4
    Modules ..... 4
        Beans ..... 4
        Benutzerschnittstelle ..... 4
        Metamodule ..... 4
        Slots ..... 4
    Connections ..... 4
        Logik ..... 4
        Benutzerschnittstelle ..... 4
3. Connections ..... 5
    Overview ..... 5
    Implementierung ..... 5
    Erweiterbarkeit ..... 6
        ..... 6
4. Tools ..... 9
    Docking-Panel ..... 9
    Menüs ..... 9
    Navigationshilfen ..... 9
    Plugins ..... 9
5. De-/Serialisierung ..... 10
    Basisformat ..... 10
    Alternative Formate ..... 10
6. Verteiltes Rechnen ..... 11
    Einordung ..... 11
    Remoting ..... 11
        Infrastruktur ..... 11
        Realisierung ..... 11
7. Class-Loading ..... 13
    Einleitung ..... 13
    Umsetzung ..... 13
    Hierarchische Workspaces ..... 14
    Integration neuer Quellen ..... 14
        ClassLoader ..... 14
        ClassProvider ..... 14
8. Security ..... 15
    Überblick ..... 15
    Realisierung ..... 15
    Spezielle Permissions ..... 16
        mw.security.permissions.InsertModulePermission ..... 16
        mw.security.permissions.EstablishLinkPermission ..... 16
        dwb.prg.security.permissions.InspectParamPermission ..... 17

dwb.prg.security.permissions.ShowParamPanelPermission ..... 17
mw.security.permissions.LoadWorkspacePermission ..... 17
mw.security.permissions.SaveWorkspacePermission ..... 17
mw.security.permissions.RemoveModulePermission ..... 18
mw.security.permissions.RemoveLinkPermission ..... 18
dwb.prg.security.permissions.ToggleLinkActivityPermission ..... 18
dwb.prg.security.permissions.DynamicSVGPermission ..... 18
dwb.prg.security.permissions.AviatorPermission ..... 18
Schaffung neuer spezifischer Permissions ..... 18
Check spezieller Permissions ..... 19
Boolesche Entscheidung ..... 19
PrivilegedAction ..... 20

# Liste der Beispiele

3.1. ..... 5
8.1. Spezieller Principal mit Permission InsertModulePermission ..... 15
8.2. Allgemeiner Principal mit Permission InsertModulePermission ..... 15
8.3. Spezielle Permission ..... 19
8.4. Einfacher Check, ob Permission erteilt ..... 19
8.5. Check über PrivilegedAction ob Permission erteilt ..... 20

# Kapitel 1. Überblick

## Datenflussprogrammierung

Datenflussprogrammierung ist eine Technologie, die nunmehr bereits seit 30 bis 40 Jahren angewendet wird, um Probleme in der Informationsverarbeitung zu lösen. Abstrakt gesprochen handelt es sich bei der Datenflussprogrammierung um die Definition und Konfiguration von Datenquellen und Datensenken und Datentransformationen und um die Definition der Pfade, über die Informationen zwischen diesen Akteuren transportiert werden.

Es existieren sowohl traditionelle Programmiersprachen, bei denen diese Definitionen und Konfigurationen in einer Textdatei hinterlegt wird, wie auch graphische Systeme, in denen die Senken, Quellen und Transformationen graphisch repräsentiert werden und die Datenpfade diese graphischen Repräsentationen verbinden, dadurch entsteht ein Netzwerk oder Graph, das visuell die Abläufe repräsentiert - man kann den Daten sozusagen bei ihrem Weg durch die Verarbeitungseinheiten zusehen.

dWb+ ist ein solches graphisches Framework zur Datenflussprogrammierung. Es geht aber noch über die bisher beschriebenen Leistungen hinaus: Außerdem ist es ein Framework, um schnell und einfach neue Transformationen, Quellen und Senken programmieren zu können. Dazu können außer der Haupt-Programmiersprache Java auch Skriptsprachen wie etwa BeanShell (ein interpretierter Java-Dialekt) oder Groovy benutzt werden. Die Menge von Sprachen, mit denen neue Module implementiert werden können, ist erweiterbar. Weiterhin kann man dWb+ sehr einfach dazu benutzen, ganz generell Datenflüsse zu modellieren. Dadurch wird es für Fälle interessant, in denen Sprachen ohne eigene visuelle Designwerkzeuge eingesetzt werden sollen. In diesem Fall kann dWb+ die Modellierung übernehmen und die eigentliche Zielsprache dann die Ausführung.

dWb+ bleibt den prinzipiellen Vorteilen von Systemen zur Datenflussprogrammierung treu: eine inherente Parallelisierbarkeit ist ebenso gegeben, wie die intuitive Bedienbarkeit, die auch Nicht-Programmierern auf einfache Art und Weise gestattet, komplexe Datenflüsse zu konfigurieren, um bestimmte Ziele zu erreichen.

Eine ausführliche Beschreibung einiger Aspekte der Datenflussprogrammierung und ihre Umsetzungin dWb+ sind im Anwenderhandbuch dWb+ im „Datenflussprogrammierung“ zu finden.

## Gliederung

Dieses Architekturhandbuch gibt einen Überblick über das Zusammenspiel der einzelnen Komponenten in der Anwendung dWb+. Dabei werden mehrere Komponenten jeweils zu einem thematisch zusammenhängenden Komplex zusammengefasst, dem ein eigenes Kapitel gewidmet wird.

Zunächst wird es um die Laufzeitumgebung für Module gehen - das ist die Komponente, die die einzelnen Module enthält und sie zum Leben erweckt: Sie stellt verschiedene Dienste für die Module zur Verfügung und verwaltet ihren Lebenszyklus.

In einem weiteren Kapitel werden die Werkzeuge genauer vorgestellt, die mit der Laufzeitumgebung interagieren um sie mit Modulen zu bevölkern.

Darauf folgt ein Kapitel, in dem die verschiedenen architektonischen Aspekte der Verbindungen zwischen Modulen erörtert werden.

Im Anschluss daran werden die verschiedenen Varianten der De-/Serialisierung von Workspaces vorgestellt und verglichen. Vor- und Nachteile werden erörtert und gezeigt, welche Variante für welchen Einsatzzweck vorgesehen ist.

Am Schluss schließlich steht ein Kapitel, in dem dem Thema verteiltes Rechnen/Remoting Raum gegeben wird: Es ist möglich, Module mit einem Mausklick auf einen entfernten Rechner zu verschieben, so daß die eigentliche Berechnung ausgelagert wird. Damit sind auch Workspaces möglich, die auf nur einem Rechner aus Performancegründen nicht ausgeführt werden könnten.

# Programmiermodell

Prinzipiell ist die Anwendung verschieden erweiterbar. Die verschiedenen Wege unterscheiden sich dabei in ihrer Agilität. Im Einzelnen existieren folgende Stufen der Agilität:

# Quelltextbearbeitung

Änderungen am Quelltext beeinflussen das Verhalten der Anwendung. Die Ändeurngen werden erst dann wirksam, wenn die Anwendung neu übersetzt und ausgerollt wurde.

# Änderungen werden nach Neustart wirksam

Ein Beispiel dafür ist die Benutzung der SPI-Schnittstelle für Java: Beim Start werden die JAR-Dateien im Klassenpfad nach bestimmten Dateien durchsucht, die Informationen zu konkreten Implementierungen von verschiedensten Interfaces beinhalten. Diese Informationen benutzt die Anwendung, wenn auf die Interfaces zugegriffen werden soll.

Solche Änderungen können durchgeführt werden, ohne die gesamte Anwendung neu übersetzen oder ausrollen zu müssen. Man muss dazu lediglich eine JAR-Datei mit den korrekten Metadaten und allen für die Implementierung benötigten kompilierten Klassen erstellen und diese in ein Verzeichnis zu legen, das zum Klassenpfad der Anwendung gehört.

# Änderungen werden zur Laufzeit wirksam

## BeanContext Services

Die Anwendung benutzt das Konzept der BeanContextServices. Solche Services stellen letztlich Interfaces und konkrete Implementierungen der Interfaces dar, Allerdings existiert hier - anders als zum Beispiel bei SPI - die Infrastruktur, um die Implementierung von Interfaces zur Laufzeit auszutauschen und Konsumenten der Interfaces von der neuen Implementierung zu informieren.

Die Anwendung bietet die Möglichkeit, solche Services in Workspaces zu deployen, indem Module zum Workspace hinzugefügt werden, die BeanContextServices zur Verfügung stellen. Mit einem einfachen Löschen des jeweiligen Moduls ist der Service verschwunden und mit dem Fallenlassen eines anderen Moduls, das denselben Service anbietet ist die Implementierung geändert. Die Tatsache, dass es die Anwendung erlaubt, Modulsammlungen zur Laufzeit hinzuzufügen kann man erkennen, dass diese Methode die agilste der drei vorgestellten ist.

## Scripting

Ausgewählte Teilaspekte der Anwendung dWb+ sind durch Skripte anpassbar. Dies bezieht sich auf reine Verhaltensmodifikationen der Anwendung wie zum Beispiel die Möglichkeit, Skripte für einzelne Verbindungen zwischen Modulen zu definieren, die die übertragenen Daten modifizieren. Dazu wird als Skriptsprache BeanShell benutzt. Vergleiche dazu auch „Skripts für Verbindungen“ und „Actions im Editor von Skripts für Verbindungen“ im Anwenderhandbuch dWb+ Darüber hinaus ist es auch möglich, Module selbst als Skripte zu implementieren. Dazu stehen zur Zeit als Sprachen BeanShell und Groovy zur Verfügung. Weitere Informationen dazu sind im „Skripted Module für noch schnelleres Rapid Prototyping“ im Anwenderhandbuch dWb+ zu finden.

# Kapitel 2. Runtime

## Workspace

### Logik

#### Überblick

Der Workspace dient als Behälter für Module und Verbindungen. In der Anwendung dWb+ sind Module Funktionseinheiten, die Daten von anderen Modulen entgegennehmen und Daten an andere Module versenden können. Verbindungen sind Spezifikationen, welche Module Daten von welchen anderen Modulen entgegennehmen können.

Die Kommunikation geschieht Event-basiert. Da alle Module in der Anwendung dWb+ JavaBeans darstellen, werden dazu PropertyChangeEvents benutzt. Allerdings werden diese Events nur zum Senden benutzt: Technisch gesehen sitzt zwischen den beiden Modulen, die an jeder Kommunikation beteiligt sind, noch ein Proxy, der das Interface PropertyChangeListener implementiert und die korrekte Methode am Empfänger-Module aufruft. Daher muss das empfangende Modul überhaupt nicht auf PropertyChangeEvents lauschen; das erledigt der in jeder Verbindung enthaltene, dem Empfänger vorgeschaltete Proxy.

Der Grund für diese Abweichung vom JavaBeans-Paradigma war zweigeteilt: Zum einen sind PropertyChangeEvents typunabhängig: Der Listener nimmt alle Events entgegen - egal, welchen Typ seine Payload hat. Da aber von vornherein feststand, dass der Anwender nur solche Aus-und Eingänge miteinander verbinden können sollte, die vom Typ her aufeinander passen, musste der PropertyChangeEvent unterstützt werden, da er nicht allein in der Lage ist, diese Anforderung zu erfüllen. Deswegen wurde festgelegt, dass jede öffentliche Methode mit einem Parameter und ohne Rückgabewert einen Eingang eines Moduls darstellt. Der Proxy macht dann nichts anderes, als diese Methode mit der im Event eingepackten Payload aufzurufen.

Der zweite Grund für die Einführung des Proxy-Mechanismus war das Nachdenken über das Dilemma, für einfachste Operationen Module einsetzen zu müssen: Das Skalieren eines Zahlenwertes zum Beispiel. Bei sehr vielen solch einfachen Operationen wäre der Workspace sehr schnell mit Modulen überfüllt, ohne dass die eigentliche Aufgabe gelöst ist. Daher wurde die Idee umgesetzt, Daten während der Übertragung zu transformieren: Jede Verbindung bietet die Möglichkeit, ein Skript zu definieren, das mit jedem übertragenen Datum aufgerufen wird. Der Empfänger bekommt dann nicht die Ausgabe des Vorgängermoduls als Eingabe präsentiert, sondern das Ergebnis des Skripts, in das die Eingabe des Vorgängermoduls als Parameter eingespeist wird.

Damit ergibt sich die Kommunikation zwischen Sender A und Empfänger B wie folgt: Eine Verbindung zwischen einem PropertyChangeEvent für das Property C in A und einem Methode D in B wird definiert. Das richtet einen entsprechenden Proxy ein, der als PropertyChangeListener an A auf Events mit dem PropertyName C lauscht. Sendet A einen entsprechenden PropertyChangeEvent, empfängt ihn der Proxy und extrahiert die Payload als zu transportierendes Datum E . Anschließend ermittelt der Proxy, ob ein Skript definiert wurde. Ist das der Fall, ersetzt er E durch das Ergebnis der Skriptauswertung. Danach ruft er D an B auf und übergibt als Parameter E.

BeanContext
SelectionAnalyzer
Variablen
Benutzerschnittstelle
Interaktionsmetaphern
Drag'n'Drop
Menüs
Modules
Beans
Konsumenten von BeanContextServices
API und Basisklassen
Benutzerschnittstelle
(Semi-)automatische Erstellung von Konfigurationsoberflächen
Embleme zur Informationsvermittlung
Metamodule
Slots
Connections
Logik
PropertyChangeProxy
Script
Visitor
PropagationManager
Benutzerschnittstelle
Graphische Darstellung
Hervorhebung
Embleme

# Kapitel 3. Connections

## Overview

Verbindungen werden zwischen Modulen definiert. Dabei verbindet eine Verbindung immer genau einen Ausgang mit genau einem Eingang. Verbindungen funktionieren gerichtet: Sie leiten Daten von einem Ausgang an einen Eingang weiter. Die Verbindungen in dWb+ sind in der Lage, Daten zu manipulieren: Sie bieten die Möglichkeit an, Daten, die aus dem Ausgang des verbundenen Moduls empfangen wurden, vor der Weiterleitung an den verbundenen Eingang per BeanShell-Skripting zu ändern.

Verbindungen können nur zwischen passenden Ein- und Ausgängen installiert werden. Die Entscheidung, ob Ein- und Ausgang zueinander passen wird anhand ihres Datentyps getroffen. Dabei gilt, dass der Eingang einen Untertyp des Ausgangs aufweisen muss, damit eine Verbindung zwischen ihnen etabliert werden kann. Das Konzept von Ober- und Untertypen ist am einfachsten an einem Beispiel zu erklären:

**Beispiel 3.1.**

Seien zwei Typen als Zahl und Ganzzahl gegeben. Ganzzahl ist ein Untertyp von Zahl. Zahl ist ein Obertyp von Ganzzahl. Das liegt daran, dass alle Ganzzahlen Zahlen sind, jedoch nicht alle Zahlen Ganzzahlen.

Um Verbindungen zwischen Modulen nicht unnötig zu erschweren, ist es ebenfalls möglich, Eingänge und Ausgänge miteinander zu verbinden, wenn einer von beiden ein Array ist, aber der Fundamentaldatentyp des Arrays mit dem Datentyp des anderen so kompatibel ist, wie oben bereits ausgeführt. In einem solchen Fall hängt was passiert davon ab, welcher von beiden der Arraydatentyp ist: ist der Eingang derjenige mit einem Arradatentyp, wird jeder skalare Wert, der vom verbundenen Ausgang empfangen wird vor der Weiterleitug an den Eingang in ein einelementiges Array verpackt. Werden vom Ausgang Arrays empfangen und der verbundene Eingang erwartet einzelne skalare Werte, so entnimmt die Verbindung die einzelnen Werte aus dem Array und leitet sie in der korrekten Reihenfolge als skalare Werte an den Eingang weiter.

# Implementierung

Zur Zeit ist die eigentliche Kommunikation -das heißt die Übertragung der Daten zwischen dem Ausgang des einen und dem Eingang des anderen Moduls - noch direkt implementiert: Ein versendetes Datenpaket aus dem Ausgang löst einen Event für den mit der Verbindung entstandenen Observer aus. Dieser führt nun die benötigten Anpassungen durch: Das Skript wird ausgeführt und gegebenenfalls die Anpassung zwischen skalarem und Array-Datentyp. Der Observer weiß über das mit dem Eingang verbundene Modul Bescheid und ruft direkt die entsprechende Methode auf, die diesem Eingang entspricht. Daher können derzeit zwei mit einer Verbindung verbundene Module nur in derselben Virtuellen Maschine existieren.

Es existiert für jede Verbindung ein eigener Observer: Selbst wenn von einem Ausgang zwei Verbindungen zu unterschiedlichen Eingängen eingerichtet werden, erhält jede dieser Verbindungen doch ihren eigenen Observer. Dies liegt darin begründet, dass durch die Möglichkeit der Skriptverarbeitung unterschiedliche Daten an die einzelnen Eingänge geliefert werden müssen, dass die Verbindungen einzeln (in-)aktiv geschaltet werden können und dass irgendwann die Beschränkung auf die zwingende Anwesenheit aller verbundenen Module in derselben VM aufgehoben werden soll.

Es existiert noch eine weitere Indirection in diesem Prozess: Sie wird durch die Klasse `PropagationManager` im Namensraum `de.elbosso.dataflowframework.logic` realisiert. Diese Indirection kann nicht zur Laufzeit der Anwendung de/aktiviert werden: Vielmehr wird sie über den Schlüssel `dwb.PropagationManager.use` beim Start der Anwendung de/aktiviert.

Ist sie aktiv, werden alle Daten, die normalerweise sofort nach Empfang und Aufbereitung über den Observer direkt über den besprochenen Methodenaufruf an den Empfänger weitergeleitet werden, an den PropagationManager übergeben. Dieser sorgt dann erst für den Methodenaufruf. Der Sinn dieser weiteren Indirection ist, dass dadurch eine determinierte Reihenfolge der Kommunikationsvorgänge erreicht wird. Dies wiederum ist die Grundlage für das Debugging in dWb+: Der Anwender kann - wenn der oben genannte Schlüssel in der Konfiguration aktiviert wurde - die Abarbeitung des Workspace anhalten und dann im Einzelschritt abarbeiten. Einzelschritt bedeutet hierbei, dass eine Verbindung nach der anderen ihre Daten überträgt und der Anwender nach jeder solchen Übertragung das Gesamtsystem analysieren kann.

Da die Zwangssequentialisierung wie sie oben geschildert wurde natürlich mit einer Einschränkung der Parallelisierung einhergeht, ist dieser Modus per Default zunächst inaktiv und muss vom Anwender explizit aktiviert werden.

Während der Übertragung der Daten vom Ausgang des Vorgängermoduls zum Eingang des Empfängers wird gleichzeitig auch der Context (so er gesetzt ist) weitergeleitet: Der Context wird von einem Modul an ein anderes weitergereicht, wann immer Daten zwischen beiden übertragen werden. Damit kann man zum Beispiel einen HTTP-Server implementieren: Ein Modul nimmt den Request entgegen und speichert die Response als Context. Die Daten des Requests - zum Beispiel der Body - wird an die nächsten Module zur Verarbeitung versendet. Bei jeder Übertragung wird der Context ebenfalls propagiert. Irgendwann kommt die Verarbeitung zu einem Ergebnis, das an den HTTP-Client gesendet werden muss. Das kann geschehen, weil im Context die Informationen zur Response weitergeleitet wurden und ebenfalls zur Verfügung stehen. Ein weiteres Beispiel ist eine Verarbeitungskette mehrerer Module, wobei das erste Bilder aus Dateien lädt und die Dateinamen in den Context schreibt. Am Ende der Kette steht ein Ergebnis der Bildverarbeitung, das durch die Informationen im Context mit dem jeweiligen Bildnamen korrelliert werden kann.

Jedes Modul kann nur ein Objekt im Kontext haben. Möchte man mehr darin abspeichern, muss man das Objekt im Kontext als Container - zum Beispiel als Map - ablegen. Die Zuordnung geschieht mittels einer 1:1 Beziehung zum Modul oder (bei Threaded-Modulen, vergleiche Kapitel 5, *Arbeiten im Hintergrund (Threads)* und Kapitel 25, *Module mit geänderter Kommunikationsmetapher* im Pogrammierhandbuch dWb+) zum Event, der die Datenweitergabe aktiviert. Dazu existieren in den Moduilbasisklassen entsprechende Methoden, die man aufrufen muss, möchte man neue Informationen in den Kontext schreiben oder welche daraus auslesen. Um sicherzustellen, dass Informationen, die bereits im Context enthalten sind, weotergeleitet werden, ist keine Änderung an Modulen nötig - die Weiterleitung geschieht für Entwickler von Modulen vollkommen transparent.

# Erweiterbarkeit

Die Erweiterbarkeit auf Quelltextebene um neue Funktionalitäten ist relativ einfach zu erreichen:

### Liste der Punkte, an denen Erweiterungen für Verbindungen im Quelltext eingefügt werden können

| | |
|---|---|
| Wichtige Klassen (zur Serialisierung/Deserialisierung) | Die Klasse `Link` im Namensraum `de.netsysit.dataflowframework.ui` ist die zentrale Klasse für die Zusammenfassung aller Aspekte einer Verbindung in dWb+. Sie ist abgeleitet von der Klasse gleichen Namens im Namensraum `de.netsysit.ui.moduleworkspace`.<br><br>Die Klasse `PropertyChangeProxy` im Namensraum `de.netsysit.dataflowframework.logic` kümmertsich um die Mechanismen eigentlichen Datenkommunikation zwischen den verbundenen Modulen. |

Wichtig wäre noch die Klasse `LinkDescription` im Namensraum `de.netsysit.dataflowframework.ui`, die die Grundlage dafür bildet, dass der Zustand jeder Verbindung serialisiert und nach dem erneuten Einladen wiederhergestellt werden kann. Die Erstellung einer Instanz dieser Klasse erfolgt durch Aufruf des Kostruktors mit Aufru des Konstruktors, der eine Instanz vom Typ Link übergeben bekommt.

Für die erfolgreiche Deserialisierung ist die Klasse `GraSim` im Namensraum `de.netsysit.dataflowframework.ui` verantwortlich - genauer die Methode `decode`: Die liest die verschiedenen Beschreibungen und erstellt daraus wieder Workspace-Objekte, zu denen auch die Link-Instanzen gehören. Fügt man also neue Attribute zu dieser Klasse hinzu, die in die De-/Serialisierung mit einbezogen werden sollen, so ist diese Methode entsprechend zu erweitern.

Actions im Zusammenhang mit Verbindungen

Actions, die mit Verbindungen im Zusammenhang stehen, existieren im Workspace-Menü (siehe auch Tabelle 2.5, „Untermenü Verbindungen“)

Alle Actions, die auf der aktuell selektierten Verbindung arbeiten sollen, müssen entsprechend der Tatsache, ob es derzeit eine selektierte Verbindung gibt, en/disabled werden. Das geschieht global in der Methode `preparePopupShow`. Der Zugriff auf die aktuell selektierte Verbindung erfolgt über die Methode `getSelectedLink()` derselben Klasse.

Graphische Darstellung der Verbindung im Workspace

Die graphische Darstellung der Verbindung im Workspace setzt sich aus verschiedenen Komponenten zusammen. Folgende Teilaspekte der Darstellung liefern Informationen über die Verbindung: Die Strichstärke der Verbindung zeigt an, ob die Verbindung aktuell selektiert wurde (diese ist dicker gezeichnet als alle anderen). Der Stil veranschaulicht den Status: Verbindungen, die durchgezogen dargestellt werden sind aktiv, solche, die gestrichelt dargestellt werden, sind inaktiv. Der Anwender kann die Farben der Verbindungslinien anpassen, um sie visuell hervorzuheben. Diese Aspekte werden in der Methode `drawYourself` der Klasse `Link` im Namensraum `de.netsysit.dataflowframework.ui`, beziehungsweise der Methode gleichen Namens in der Klasse `Link` im Namensraum `de.netsysit.ui.moduleworkspace` implementiert.

Die Benennung der Verbindung und ein eventuell einzublendendes Emblem wird über die Methode `decorate` realisiert, die in der Klasse `Link` im Namensraum `de.netsysit.ui.moduleworkspace` zu finden ist. Die gleichnamig Klasse im Namensraum `de.netsysit.dataflowframework.ui` überschreibt diese Methode zur Zeit noch nicht.

Visitors

Aktuell können an jeder Verbindung mehrere sogenannte Visitors registriert werden. Um nicht in die Irre zu führen: Diese Registrierung erfolgt für jede Verbindung.

Alle zur Verfügung stehenden Visitors werden an alle Verbindungen registriert, sobald eine Verbindung erstellt wird. Diese Visitors dienen dazu, informiert zu werden, sobald Daten über die Verbindung übertragen werden. Dazu umfasst die Signatur der einzigen Methode `dataCameThrough` des Interface `PropertyChangeProxy.Visitor` im Namensraum `de.netsysit.dataflowframework.logic` Informationen über die Unique ID und den Namen des Aus-/ Eingangs des sendenden und empfangenden Moduls, wie auch die Daten, die vor und nach der Bearbeitung durch ein (eventuell für die Verbindung definiertes) Skript über die Verbindung weitergeleitet werden.

Damit wäre es prinzipiell möglich, neue Kommunikationsmetaphern zu realisieren. Module können nämlich dieses Interface ebenfalls implementieren. Tun sie das, werden sie als Observer, bzw. Visitor, für neu eingerichtete Verbindungen zu einem ihrer Eingänge registriert und erhalten nicht nur die Daten des sendenden Moduls, sondern ebenso seine Unique ID und den Namen des sendenden Ausgangs.

Wie bereits gesagt wird ein Modul, das den Empfänger einer neuen Verbindung beisteuert und dieses Interface implementiert, automatisch an der neuen Verbindung als Visitor registriert. Darüber hinaus wird bei Einrichtung einer neuen Verbindung über das SPI-Framework nach Implementierungen dieses Interface gesucht, die über die Klasse `ServiceLoader` im Namensraum `java.util` gefunden werden können. Alle gefundenen Implementierungen bzw. Instanzen werden ebenfalls als Visitors registriert.

BeanContextChild

Für die Arbeit mit BeanContext Services ist es nötig, die Klasse `Link` aus dem Namensraum `de.netsysit.dataflowframework.ui` in den entsprechenden Methoden `setBeanContext`, `getBeanContext`, `serviceAvailable` und `serviceRevoked` entsprechend zu erweitern wann immer die Klasse neue Services benutzen soll.

Sollen neue Services für `Link` zur Verfügung gestellt werden, sind diese in den Workspaces zur Verfügung zu stellen. Das geschieht entweder durch die Registrierung in der Klasse `ServiceManager` im Namensraum `de.netsysit.dataflowframework.logic.services` oder durch die Schaffung von Modulen, die als ServiceProvider arbeiten können und dynamisch zu den Workspaces hinzugefügt werden (Vergleiche dazu auch das im Kapitel 8, *BeanContextServices bereitstellen* im Pogrammierhandbuch dWb + Gesagte). Das System erkennt solche Module und stellt die von ihnen angebotenen Services allen anderen Modulen und Verbindungen im Workspace zur Verfügung. Der entsprechende Code dafür findet sichin der Klasse `ModuleWorkspace` im Namensraum `de.netsysit.ui.moduleworkspace` in der Methode `addToServiceManager`.

# Kapitel 4. Tools

## Docking-Panel

## Menüs

## Navigationshilfen

## Plugins

# Kapitel 5. De-/Serialisierung

## Basisformat

Workspaces als Sammlung von Modulen und Verbindungen werden als XML-Dateien gespeichert. Dabei wird das Format benutzt, das der in Java selbst eingebaute XML De-/Serialisierungsmechanismus nutzt.

Im Detail werden folgende Dinge in den Dateien gespeichert: Angaben zum Workspace - also zum Beispiel, welche Layer sichtbar und welche unsichtbar geschaltet sind. Für jede Verbindung wird zunächst der zugehörige Sender und Empfänger gespeichert, weiterhin: ob die Verbindung aktiv oder inaktiv ist und gegebenenfalls das Skript zur Modifikation der übertragenen Daten. Für die Module wird die Logging-Konfiguration gespeichert, ferner Position und Status (geöffnet/geschlossen) des Parameterdialogs und alls Werte der schreibbaren Properties.

Bei dieser Form ist zu beachten, dass diese nur dann erfolgreich sein kann, wenn es für die Klassen aller zu speichernden Instanzen möglich ist, diese aus dem XML-Fragment wieder herzustellen. Das funktioniert nur, wenn es sich durchgängig um JavaBeans handelt (öffentlicher parameterloser Konstruktor!) oder für die Klassen, auf die das nicht zutrifft, PersistenceDelegates registriert wurden. Das können Entwickler von Modulen am einfachsten dadurch erreichen, wenn sie mindestens einen öffentlichen Konstruktor mittels Annotationen auszeichnen und so festlegen, welches Property welchem Konstruktorparameter entspricht.

## Alternative Formate

# Kapitel 6. Verteiltes Rechnen

## Einordung

Traditionell wurden Anwendungen für das graphische Programmieren oder Datenfluss-Programmierung im Bereich des Rapid Prototyping eingesetzt. Das bedeutete traditionell, dass die Anwendung auf einem Rechner ausgeführt wurde und verteiltes Rechnen wenn überhaupt eine untergeordnete Rolle spielte. Auch in anderen technologischen Ansätzen - wie zum Beispiel dem in früheren Inkarnationen auf Silverlight und aktuell auf HTML5 und Javascript aufbauende Webbles - wurde die Möglichkeit des verteilten Rechnens nicht in die Konzeption der Architektur einbezogen.

Das merkt man unter anderem daran, dass die Kommunikationsbeziehungen in den einzelnen Ansätzen eine sehr enge Kopplung der beiden Kommunikationspartner bedingen - die Silverlight-Version von Webbles ist eines der extremeren Beispiele: Hier wird die Übertragung von Informationen über einen direkten Methodenaufruf durch den Sender am Empfänger abgebildet

dWb+ sollte von Beginn an anders sein: Die Architektur sollte bereits bei der Konzeption die Möglichkeit vorsehen, dass unterschiedliche Module auf verschiedenen Rechnern angesiedelt sein sollten - auch wenn sie miteinander im selben Workspace arbeiten. Der Anwender sollte dadurch nur minimale Einschränkungen beim Aufbau der Workspaces erfahren und im Betrieb sollte die Tatsache für ihn völlig transparent sein.

Darüber hinaus sollte die Fähigkeit eines Moduls, sich auf einen Compute-Server auslagern zu lassen nicht bei der Entwicklung des Moduls beachtet werden müssen: Möglichst jede Klasse, die in dWb+ als Modul benutzt werden kann, sollte auch ohne Änderungen am Quelltext auf Compute-Server platziert werden können. Dazu musste die Anwendung eine Architektur aufweisen, dies zu erlauben. Dieses Kapitel geht auf die Möglichkeiten ein, die dWb+ zur Umsetzung dieser Anforderungen bietet.

## Remoting

### Infrastruktur

Aktuell ist es so, dass dWb+ nur eine begrenzte Menge von Compute-Servern erlaubt. Diese werden beim Start der Anwendung aus der System-Property `de.elbosso.util.beans.context.service.provider.RemotingServiceProvider.hosts` gelesen. Module, die auf Compute-Server ausgelagert werden können sollen verfügen über eine Auswahlmöglichkeit, auf welchen der definierten Hosts sie geschickt werden sollen.

Die Lösung funktioniert wie folgt: Als Teil der Tool-Suite rund um dWb+ existiert eine Komponente, die alles enthält, was einen Compute-Server für dWb+ ausmacht. Dieses self-contained JAR muss man lediglich auf einen entsprechenden Rechner ausrollen und starten. Steht dem Betrieb keine Firewall-Regel oder sonstige Betriebssystem oder Netzwerkkonfiguration entgegen, kann man den Namen dieses Rechners in die oben bereits erwähnte Property eintragen und dWb neu startem. Danach kann man bereits Remoting-fähige Module auf den neuen Compute-Server verschieben.

### Realisierung

Die Realisierung baut auf RMI auf: Eine der schönen Eigenschaften dieser Technologie ist, dass die Klassen, die auf dem Rechner benötigt werden, an den RMI-Aufrufe ergehen, dort nicht vorliegen müssen: RMI kann mit einem Remote-ClassLoader konfiguriert werden, von dem aus versucht wird, die fehlenden Klassen zu laden.

Für dWb+ heißt das, dass die Anwendung einen ClassLoader-Server startet, wenn die System-Property `de.elbosso.dataflowframework.ClassLoaderServer.port` einen gültigen Netzwerk-Port darstellt. Dieser Port muss natürlich auch allen Remote-Server-Komponenten beim Start bekannt gemacht werden. Wird jetzt ein Modul auf den Compute-Server verschoben, geschieht das so, dass eine Methode am Remote-Server aufgerufen wird und die eigentliche Modul-Klasse dieser Methode als Argument übergeben wird. Da der Remote-Server diese Klasse nicht kennt, fragt er seinen Remot-Class-Server (die dWb+-Anwendung selbst), die ihm wiederum die benötigte Klasse zur Verfügung stellt. Anschließend kann der Remote-Server das übergebene Argument instantiieren. Der Vorgang bis hierher wird Binding genannt. Nach Abschluss des Binding gilt das Modul aus Anwendersicht als auf den Compoute-Server verschoben. Das Modul im Workspace arbeitet danach lediglich als Proxy.

Nachdem das Argument instantiiert wurde, können daran Methoden aufgerufen werden. Dies entspricht der normalen Tätigkeit von Modulen: Das Empfangen von Daten ist ja auf technischer Ebene nichts anderes als der Aufruf einer Methode.

Wie kommen aber nun die Resultate wieder vom Compute-Server in den Workspace? Der Aufruf der Methode am Compute-Server erfolgtt immer synchron: Das Ergebnis des Aufrufes wird an den Proxy im Workspace zurückgegeben, der das Versenden der entsprechenden Events zur Information der nachfolgenden Empfänger übernimmt.

Das genaue Vorgehen und Beispiele bei der Implementierung von Modulen für das Remoting kann man im Pogrammierhandbuch dWb+ im Kapitel 21, *Verteiltes Arbeiten (Remoting)* finden.

# Kapitel 7. Class-Loading

## Einleitung

Für die Anwendung musste ein spezieller Modus zum Finden benötigter Klassen und Ressourcen geschaffen werden. Im Folgenden bezieht sich alles zum Thema Class-Loading gesagte immer sowohl auf Ressourcen, als auch auf Klassen.

Es wurde festgelegt, dass Klassen nur in JAR-Dateien vorliegen können. Es wird durch die Anwendung nicht unterstützt, Klassen oder Ressourcen einzeln im Dateisystem zu organisieren.

Der traditionelle ClassLoader, der in Java zum Einsatz kommt, kann in der Anwendung nicht benutzt werden: Normalerweise stellt ein ClassLoader in Java eine strenge Hierarchie dar, in der es eine Wurzel gibt. Klassen werden prinzipiell erst in den im Baum weiter oben liegenden Implementierungen gesucht. Erst wenn sie dort nicht gefunden werden, werden sie in den weiter unten liegenden Ebenen gesucht. Es ist unmöglich, In der Hierarchie zur Laufzeit neue Elemente hinzuzufügen ist nur dann möglich, wenn die neuen Elemente unterhalb von Blättern des Baumes eingefügt werden.

Im vorliegenden Fall müssen neue ClassLoader zur Laufzeit hinzugefügt werden, die als gleichberechtigte Zweige parallel zu bestehenden zu organisieren sind.

## Umsetzung

Es wurde eine Möglichkeit geschaffen, einen eigenen ClassLoader überall da zu benutzen, wo Klassen geladen werden sollen. Das betrifft zum Beispiel die Instantiierung von Modulen beim Drag'n'Drop oder die De-/Serialisierung von Worksapces nach und von XML. Dieser ClassLoader wird auch als CurrentClassLoader an allen Threads gesetzt, die Objekte instantiieren müssen.

Dieser ClassLoader wurde mit der Möglichkeit ausgestattet, Klassen zunächst in den parallel vorhandenen Zweigen von ClassLoadern nach Klassen oder Ressourcen zu suchen, bevor diese mit dem Parent-ClassLoader aufgelöst werden.

Zu diesem Zweck wurde eine Klasse namens `VariableClassLoaderProxy` im Namensraum `de.elbosso.util` benutzt. Diese Implementierung stellt zunächst eine Fassade für einen anderen ClassLoader dar, der bei der Konstruktion einer Instanz angegeben werden muss. Er stellt jeweils zwei Methoden zur Verfügung, mit denen man neue ClassLoader registrieren oder bestehende deregistrieren kann:

Damit ist es möglich, direkt Instanzen als alternative Quellen für Klassendefinitionen zu nutzen, die mittelbar von `ClassLoader` im Namensraum `java.lang` ebgeleitet sind. Dazu benutzt man die Methoden, die eine Instanz einer Klasse als Parameter erwarten, die das Interface `ClassLoaderProvider` aus dem Namensraum `de.elbosso.util` implementieren. Es gibt allerdings auch durchaus Anwendungsfälle, in denen eine Instanz einer Klasse als alternative Quelle dient, die nicht mittelbar von `ClassLoader` ClassLoader abgeleitet wurde. Für solche Fälle existieren die anderen beiden Methoden, die eine Instanz einer Klasse als Parameter erwarten, die das Interface `ClassProvider` aus dem Namensraum `de.elbosso.util` implementieren.

Wird eine Instanz der Klasse namens `VariableClassLoaderProxy` nach einer Ressource befragt, wird diese Anfrage zunächst an den bei der Instantiierung übergebenen ClassLoader übergeben. Findet dieser nichts, werden der Reihe nach alle registrierten ClassLoaderProvider befragt. Falls auch das fehlschlägt, werden alle ClassProvider befragt. Wurde auch dort keine auf die Anfrage passende Klasse oder Ressource gefunden, wird dieses Ergebnis an den aufrufenden Code gemeldet - *Auf keinen Fall wird der Parent-ClassLoader befragt!*

Mittels dieser Mechanik ist es möglich, dynamisch ClassLoader zur Laufzeit zum Suchpfad hinzuzufügen und auch wieder daraus zu entfernen. Darauf basiert zum Beispiel auch das Laden von Klassen aus den Modul-JARs oder aus den Bibliotheken (vergleiche Anhang A, *Verzeichnis-Layout* im Anwenderhandbuch dWb+).

# Hierarchische Workspaces

Eine entsprechende Instanz der Klasse `VariableClassLoaderProxy` wird beim Anlegen des Top-Level-Workspace erzeugt und dem Konstruktor der Klasse `GraSim` im Namensraum `de.netsysit.dataflowframework.ui` übergeben. Hierarchisch darunter liegende Unter-Workspaces (vergleiche „Gruppe“ im Anwenderhandbuch dWb+) werden nicht mit eigenen Instanzen dieser Klasse erzeugt: Dem Konstruktor der Klasse `GroupDesktop` im Namensraum `de.netsysit.dataflowframework.ui.groups` wird eine Instanz des jeweils übergeordneten Workspace übergeben. Auf diese Weise wird der ClassLoader,beziehungsweise die Instanz der Klasse `VariableClassLoaderProxy` auch an alle Unter-Workspaces propagiert. Damit haben alle Workspaces in der Hierarchie Zugriff auf den speziellen ClassLoader

# Integration neuer Quellen

## ClassLoader

Schafft man neue Plugins, die ebenfalls zum Beispiel Module zur Verfügung stellen sollen, muss man diesen vorgestellten Mechanismus nutzen, damit die Anwendung über die neuen Quellen informiert ist und diese auch nutzen kann. Dazu existiert in der Klasse `GraSim` im Namensraum `de.netsysit.dataflowframework.ui` die zwei Methoden `setPluginClassLoaderProvider` und `unsetPluginClassLoaderProvider`. Diese können jedoch nur von Klassen aus dem Namensraum `de.netsysit.dataflowframework.ui` genutzt werden. Die Skripting-Schnittstelle für die Klasse `GraSim` (`BSHGraSimAPI` im selben Namensraum exportiert diese Funktionalität aber auch für Klassen anderer Namensräume mittels der beiden Methoden `setPluginClassLoaderProvider` und `unsetPluginClassLoaderProvider`.

Diese Methoden erwarten Objekte als Parameter, deren Klassen das Interface `ClassLoaderProvider` aus dem Namensraum `de.elbosso.util` implementieren.

## ClassProvider

Schafft man neue Plugins, die ebenfalls zum Beispiel Module zur Verfügung stellen sollen, muss man diesen vorgestellten Mechanismus nutzen, damit die Anwendung über die neuen Quellen informiert ist und diese auch nutzen kann. Dazu existiert in der Klasse `GraSim` im Namensraum `de.netsysit.dataflowframework.ui` die zwei Methoden `setPluginClassLoaderProvider` und `unsetPluginClassLoaderProvider`. Diese können jedoch nur von Klassen aus dem Namensraum `de.netsysit.dataflowframework.ui` genutzt werden. Die Skripting-Schnittstelle für die Klasse `GraSim` (`BSHGraSimAPI` im selben Namensraum exportiert diese Funktionalität aber auch für Klassen anderer Namensräume mittels der beiden Methoden `setPluginClassProvider` und `unsetPluginClassProvider`.

Diese Methoden erwarten Objekte als Parameter, deren Klassen das Interface `ClassProvider` aus dem Namensraum `de.elbosso.util` implementieren.

# Kapitel 8. Security

## Überblick

Obwohl die Anwendung selbst hauptsächlich für die Benutzung im Rapid Prototyping gedacht war, war doch von Beginn der Entwicklung an die Sicherheit ein wichtiges Thema - und dabei vor allem in Bazug auf Autorisierung und Authentifizierung.

Genauer gesagt sollte es möglich sein, die Anwendung, bzw. deren Funktionsumfang entsprechend der Rolle, die ein authentifizierter Nutzer zugewiesen bekommt, anpassen zu können. Man könnte sich also zum Beispiel Szenarien vorstellen, in denen ein Nutzer A einen Workspace entwirft, indem er Module auf dem Arbeitstisch anordnet und Verbindungen zwischen ihnen definiert. Nach Fertigstellung übergibt er diesen an andere Nutzer, die mit ihren jeweiligen Auorisierungen aber in eine andere Gruppe eingeordnet werden. Die Mitglieder dieser anderen Gruppe verfügen nicht über die Rechte zur strukturellen Veränderung eines Workspace (Hinzufügen/Ändern/Entfernen von Modulen/Verbindungen), können aber Parameter der einzelnen Module ändern.

## Realisierung

Dieses Thema wurde mit JAAS umgesetzt: eine entsprechende jaas-Konfigurationsdatei steuert die Art und Weise der Authentifizierung und Autorisierung. Danach ist ein Security-Principal für den Anwendungskontext etabliert. Über entsprechende Konfigurationsabschnitte in der Konfigurationsdatei für den SecurityManager werden dem Principal dann seine ihm eingeräumten Privilegien zugeordnet. Hier sieht man die Konfiguration für einen Principal namens examplePrincipal, dem erlaubt wird, Module auf Arbeitstische zu platzieren:

**Beispiel 8.1. Spezieller Principal mit Permission InsertModulePermission**

```
grant
 Principal de.netsysit.dataflowframework.prg.security.Principal "examplePrincipal"
{
   permission de.netsysit.ui.moduleworkspace.security.permissions.InsertModulePerm
}
```

Zu beachten ist hierbei, dass der Principal die Rechte nur eingeräumt bekommt, wenn die Implementierung, mittels derer er registriert wird, der hier angegebenen entspricht: Wird ein anderes JAAS-Login-Module benutzt, daseine andere Implementierung von `java.security.Principal` registriert, greift die oben angegebene Regel nicht. Möchte man agnostisch gegenüber der Implementierung des Login-Module sein, aknn man die Regel auch so schreiben:

**Beispiel 8.2. Allgemeiner Principal mit Permission InsertModulePermission**

```
grant
 Principal java.security.Principal "examplePrincipal"
{
   permission de.netsysit.ui.moduleworkspace.security.permissions.InsertModulePerm
}
```

# Spezielle Permissions

Im Folgenden werden die speziellen Permissions für dWb+ zusammen mit ihren Auswirkungen aufgezählt. Dabvei steht `dwb` für `de.netsysit.dataflowframework` und `mw` für `de.netsysit.ui.moduleworkspace`

## mw.security.permissions.InsertModulePermission

- Schaltet die Ausführung der Methode `mouseClicked` in `de.netsysit.dataflowframework.ui.InputSlot` frei.
- Schaltet die Ausführung der Methode `mouseClicked` in `de.netsysit.dataflowframework.ui.OutputSlot` frei.
- Schaltet die Ausführung der Methode `dragGestureRecognized` in `dWb.plugins.GroovyScriptedModules.ScriptedGroovyDragTree` frei.
- Schaltet die Ausführung der Methode `dragGestureRecognized` in `dWb.plugins.osgimodules.OSGIDragTree` frei.
- Schaltet die Ausführung der Methode `dragGestureRecognized` in `de.elbosso.dataflowframework.ui.JMXDragTree` frei.
- Schaltet die Ausführung der Methode `dragGestureRecognized` in `de.elbosso.dataflowframework.ui.ScriptedDragTree` frei.
- Schaltet die Ausführung der Methode `dragGestureRecognized` in `de.netsysit.dataflowframework.ui.CompiledDragTree` frei.
- Schaltet die Ausführung der Methode `dragGestureRecognized` in `de.netsysit.dataflowframework.ui.FavouritesManager` frei.
- Schaltet die Ausführung der Methode `preparePopupShow` in `de.netsysit.dataflowframework.ui.GraSim` frei.
- Schaltet die Ausführung der Methode `dragGestureRecognized` in `de.netsysit.dataflowframework.ui.ScratchManager` frei.
- Schaltet die Ausführung der Methode `dragGestureRecognized` in `de.netsysit.dataflowframework.ui.WorkspaceList` frei.
- Schaltet die Ausführung der Methode `preparePopupShow` in `de.netsysit.dataflowframework.ui.groups.GroupDesktop` frei.

## mw.security.permissions.EstablishLinkPermission

- Schaltet die Verfügbarkeit der Action `splitlinkaction` in `de.netsysit.dataflowframework.ui.GraSim` um.
- Schaltet die Ausführung der Methode `mouseClicked` in `de.netsysit.dataflowframework.ui.InputSlot` frei.
- Schaltet die Ausführung der Methode `dragGestureRecognized` in `de.netsysit.dataflowframework.ui.OutputPanel` frei.

- Schaltet die Ausführung der Methode `mouseClicked` in `de.netsysit.dataflowframework.ui.OutputSlot` frei.
-

# dwb.prg.security.permissions.InspectParamPermission

- Schaltet die Ausführung der Methode `showPopup` in `de.netsysit.dataflowframework.ui.beans.PopupStateUpdater` frei.

# dwb.prg.security.permissions.ShowParamPanelPermission

- Schaltet die Ausführung der Methode `preparePopupShow` in `de.netsysit.dataflowframework.ui.ModuleWidget` frei.
- Schaltet die Ausführung der Methode `setParamPanelDimensions` in `de.netsysit.dataflowframework.ui.ModuleWidget` frei.
- Schaltet die Ausführung der Methode `openParamPanel` in `de.netsysit.dataflowframework.ui.ModuleWidget` frei.

# mw.security.permissions.LoadWorkspacePermission

- Schaltet die Ausführung der Methode `importWS(final java.net.URL url)` in `de.netsysit.dataflowframework.ui.GraSim` frei.
- Schaltet die Ausführung der Methode `importWS(final de.netsysit.dataflowframework.ui.WorkspaceDescription[] wd)` in `de.netsysit.dataflowframework.ui.GraSim` frei.
- Schaltet die Verfügbarkeit der Action `loadaction` in `de.netsysit.dataflowframework.ui.GraSim` um.
- Schaltet die Verfügbarkeit der Action `importaction` in `de.netsysit.dataflowframework.ui.GraSim` um.
- Schaltet die Verfügbarkeit der Action `importasgroupaction` in `de.netsysit.dataflowframework.ui.GraSim` um.

# mw.security.permissions.SaveWorkspacePermission

- Schaltet die Ausführung der Methode `save` in `de.netsysit.dataflowframework.ui.GraSim` frei.
- Schaltet die Ausführung der Methode `export` in `de.netsysit.dataflowframework.ui.GraSim` frei.
- Schaltet die Verfügbarkeit der Action `saveaction` in `de.netsysit.dataflowframework.ui.GraSim` um.
- Schaltet die Verfügbarkeit der Action `exportaction` in `de.netsysit.dataflowframework.ui.GraSim` um.

## mw.security.permissions.RemoveModulePermission

- Schaltet die Ausführung der Methode `clear` in `de.netsysit.dataflowframework.ui.GraSim` frei.
- Schaltet die Verfügbarkeit der Action `clearaction` in `de.netsysit.dataflowframework.ui.GraSim` um.
- Schaltet die Verfügbarkeit der Action `deleteAction` in `de.netsysit.dataflowframework.ui.ModuleWidget` um.
- Schaltet die Verfügbarkeit der Action `cutAction` in `de.netsysit.dataflowframework.ui.ModuleWidget` um.

## mw.security.permissions.RemoveLinkPermission

- Schaltet die Verfügbarkeit der Action `splitlinkaction` in `de.netsysit.dataflowframework.ui.GraSim` um.
- Schaltet die Verfügbarkeit der Action `deletelinkaction` in `de.netsysit.dataflowframework.ui.GraSim` um.

## dwb.prg.security.permissions.ToggleLinkActivityPermission

- Schaltet die Verfügbarkeit der Action `togglelinkstateaction` in `de.netsysit.dataflowframework.ui.GraSim` um.
- Schaltet die Verfügbarkeit der Action `togglelinkstateaction` in `de.netsysit.dataflowframework.ui.ModuleWidget` um.

## dwb.prg.security.permissions.DynamicSVGPermission

- Schaltet den Verbindungsaufbau in der Methode `manageConnectionImpl` in `de.netsysit.dataflowframework.modules.SVGAnalyzer` um.
- Schaltet die Verfügbarkeit der Action `usedocumentwidgetaction` in `de.netsysit.dataflowframework.ui.GraSim` um.
- Schaltet die Verfügbarkeit der Action `connectAction` in `de.netsysit.dataflowframework.modules.SVGAnalyzer` um.

## dwb.prg.security.permissions.AviatorPermission

- Schaltet die Verfügbarkeit der Action `showPortDescriptionsAction` in `de.netsysit.dataflowframework.ui.GraSim` um.

# Schaffung neuer spezifischer Permissions

Dazu ist es lediglich notwendig, eine neue Klasse zu schaffen, die von `java.security.BasicPermission` abgeleitet wird. Im Kostruktor wird der Oberklassenkonstruktor aufgerufen, dem ein beschreibender String zu übergeben ist.

**Beispiel 8.3. Spezielle Permission**

```
public class CustomPermission extends java.security.BasicPermission
{
    public CustomPermission()
    {
        super("Custom Example Permission");
    }
}
```

# Check spezieller Permissions

Die Permissions werden - egal, ob es sich um neu implementierte Permissions oder bereits vorhandene handelt - immer gleich benutzt: Bestimmte Aktionen, die nur ausgeführt werden dürfen, wenn die Permission für den aktuell gültigen Principal vorliegt, werden mittels zweier verschiedener Patterns umgesetzt:

## Boolesche Entscheidung

**Beispiel 8.4. Einfacher Check, ob Permission erteilt**

```
try
{
    if (System.getSecurityManager() != null)
    {
        System.getSecurityManager().checkPermission(new CustomPermission());
    }
    //Permission granted
}
catch (java.lang.SecurityException exp)
{
    //Permission not granted
}
```

# PrivilegedAction

**Beispiel 8.5. Check über PrivilegedAction ob Permission erteilt**

```
javax.security.auth.Subject subject = LoginManager.
          getSharedInstance().getSubject();
javax.security.auth.Subject.doAsPrivileged(subject,
          new java.security.PrivilegedAction()
{
    public Object run()
    {
        try
        {
            if (System.getSecurityManager() != null)
            {
                    System.getSecurityManager().
                        checkPermission(new CustomPermission());
            }
            //Permission granted
        }
        catch (java.lang.Throwable t)
        {
            EXCEPTION_LOGGER.error(t.getMessage(), t);
        }
        return null;
    }
}, null);
```

# Pogrammierhandbuch dWb+

**Jürgen Key**

# Pogrammierhandbuch dWb+

Jürgen Key

# Inhaltsverzeichnis


1. Einleitung und Überblick ..... 1
    - dWb+ ..... 1
    - Module ..... 1
        - Konfiguration von Modulen ..... 2
        - Visuelle Entsprechungen in dWb+ ..... 2
    - Umsetzung in Java ..... 2
        - Allgemeines ..... 2
        - Inputs ..... 3
        - Algorithmus ..... 3
        - Outputs ..... 3
        - Konfigurationsparameter ..... 4
        - Diagnose ..... 4
        - Zusammenfassung/Schlußfolgerung ..... 5
    - Entwicklungsschritte ..... 5
2. BeanInfo-Verwendung in dWb+ ..... 6
    - Internationalisierung ..... 6
    - Slots ..... 6
        - Beschriftung ..... 6
        - Tooltips ..... 6
        - StateUpdaters ..... 6
        - Automatisches Erstellen von Verbindungen ..... 6
        - Maximale Anzahl von Verbindungen ..... 7
    - Modultitel ..... 7
    - Modulbeschreibung ..... 7
    - Icon ..... 7
    - Layer ..... 7
    - Verbergen von Properties ..... 8
3. Ein einfaches Beispiel ..... 9
    - Funktionsbeschreibung ..... 9
    - Selber machen oder Basisklasse benutzen? ..... 9
    - Input ..... 9
    - Konfiguration ..... 10
    - Output ..... 10
    - Algorithmus ..... 11
4. Eine JMX MBean als Modul ..... 12
    - Funktionsbeschreibung ..... 12
    - Selber machen oder Basisklasse benutzen? ..... 12
    - MXBean ..... 12
    - Konfiguration ..... 13
    - Notification Infrastruktur ..... 13
    - Output ..... 14
    - Input ..... 14
5. Arbeiten im Hintergrund (Threads) ..... 16
    - Warum? ..... 16
    - Ein einfaches Beispiel ..... 16
    - Selber machen oder Basisklasse benutzen? ..... 16
    - Input ..... 18
    - Konfiguration ..... 18
    - Output ..... 19
    - Algorithmus ..... 19
6. Parallele Verarbeitung innerhalb eines Moduls ..... 21

Funktionsbeschreibung ..... 21
Selber machen oder Basisklasse benutzen? ..... 21
BeanContext ..... 22
Workload ..... 23
Input ..... 23
Output ..... 24
7. BeanContext und BeanContextServices konsumieren ..... 25
Warum? ..... 25
Ein einfaches Beispiel ..... 25
Selber machen oder Basisklasse benutzen? ..... 25
Dienst nutzen ..... 26
BeanContext-Mitgliedschaft ..... 27
Modul wird zu Context hinzugefügt ..... 27
Modul wird aus Context entfernt ..... 27
Dienst wird zur Verfügung gestellt ..... 27
Dienst wird zurückgezogen ..... 29
8. BeanContextServices bereitstellen ..... 30
Warum? ..... 30
Ein einfaches Beispiel ..... 30
Selber machen oder Basisklasse benutzen? ..... 30
Dienst definieren ..... 31
BeanContext-Mitgliedschaft ..... 31
Modul wird zu Context hinzugefügt ..... 31
Modul wird aus Context entfernt ..... 31
Dienst wird zur Verfügung gestellt ..... 32
Dienst wird zurückgezogen ..... 32
Dienst bereitstellen ..... 33
Ressourcen freigeben ..... 33
ServiceSelectors definieren ..... 33
9. Generics ..... 35
Warum ..... 35
Ein einfaches Beispiel ..... 36
Selber machen oder Basisklasse benutzen? ..... 36
Input ..... 37
Output ..... 37
Algorithmus ..... 37
BeanInfo ..... 37
10. Variable Anzahl von Inputs ..... 39
Warum ..... 39
Ein einfaches Beispiel ..... 39
Selber machen oder Basisklasse benutzen? ..... 39
Input ..... 40
Output ..... 40
Algorithmus ..... 40
BeanInfo ..... 41
11. Module zur Visualisierung mit einer variablen Anzahl von Inputs ..... 42
Warum ..... 42
Ein einfaches Beispiel ..... 42
Selber machen oder Basisklasse benutzen? ..... 43
Input ..... 43
Hinzufügen neuer Komponenten durch Hinzufügen neuer Input-Slots ..... 43
Schicken neuer Daten an die korrekte Komponente ..... 44
Model ..... 44
BeanInfo ..... 44

Persistenz der Konfiguration für die Komponenten ..... 44
12. Variable Module ..... 46
Warum ..... 46
Ein einfaches Beispiel ..... 46
Selber machen oder Basisklasse benutzen? ..... 46
Input ..... 47
Konfigurationsparameter ..... 47
Output ..... 47
Versenden des PropertyChangeEvents "insAndOuts" ..... 47
Interface VariableBean ..... 48
Ergebnis ..... 48
13. Variable Module für User Eingaben ..... 50
Warum ..... 50
Ein einfaches Beispiel ..... 50
Selber machen oder Basisklasse benutzen? ..... 50
Input ..... 51
Konfigurationsparameter ..... 51
Output ..... 51
Erzeugen der Komponenten zur Konfiguration der Ausgänge ..... 51
Glue ..... 51
14. Module zur Filterung ..... 53
Funktionsbeschreibung ..... 53
Selber machen oder Basisklasse benutzen? ..... 53
Konstruktor ..... 54
15. Eigene Bedienoberflächen für ein Modul ..... 55
Anpassung der Bedienoberfläche für Module ..... 55
Vorarbeit ..... 56
Implementierung ..... 57
16. Actions zur Steuerung eines Moduls ..... 58
Warum ..... 58
Implementierung ..... 58
Actions erzeugen ..... 58
Vorbereitung Anzeige Menü ..... 59
17. Spezielle Properties zur besseren Integration von Modulen ..... 60
Warum ..... 60
18. Module, die mit externen Ressourcen arbeiten müssen ..... 61
Motivation ..... 61
Funktionsbeschreibung ..... 61
Die Basisklasse ..... 61
Input ..... 62
Konfiguration ..... 62
Deklarierte Basisklassenmethoden definieren ..... 63
19. De-/aktivierbare Module ..... 65
Funktionsbeschreibung ..... 65
Selber machen oder Basisklasse benutzen? ..... 65
Output ..... 66
Start des Hintergrundprozesses ..... 66
Workload ..... 66
Erweiterung: Einzelschritt ..... 67
20. Java Api for XML Binding ..... 69
Marshaling ..... 69
Unmarshaling ..... 69
Selber machen oder Basisklasse benutzen? ..... 69
21. Verteiltes Arbeiten (Remoting) ..... 71

Warum? ..... 71
Ein einfaches Beispiel ..... 71
Selber machen oder Basisklasse benutzen? ..... 71
Input ..... 73
Konfiguration ..... 73
Output ..... 73
Algorithmus ..... 73
Deployment ..... 74
22. Getaktete Module ..... 75
Funktionsbeschreibung ..... 75
Selber machen oder Basisklasse benutzen? ..... 75
Input ..... 76
Konfiguration ..... 76
Output ..... 76
Algorithmus ..... 76
23. Mandantenfähigkeit ..... 78
Warum? ..... 78
Selber machen oder Basisklasse benutzen? ..... 78
Daten in den Context einspeisen ..... 78
BeanContextChildModuleBase ..... 78
ThreadingBeanContextChildModuleBase ..... 78
Daten aus dem Context auslesen ..... 79
BeanContextChildModuleBase ..... 79
ThreadingBeanContextChildModuleBase ..... 79
24. Enterprise JavaBeans (EJBs) als Module ..... 80
Funktionsbeschreibung ..... 80
Selber machen oder Basisklasse benutzen? ..... 80
JNDI Lookup ..... 81
Output ..... 82
Input ..... 82
Workload ..... 82
Benutzung als Skripted Module ..... 83
25. Module mit geänderter Kommunikationsmetapher ..... 84
Warum? ..... 84
Ein einfaches Beispiel ..... 84
Selber machen oder Basisklasse benutzen? ..... 84
Input ..... 85
Konfiguration ..... 85
Output ..... 85
Algorithmus ..... 86
26. Packaging ..... 87
27. StateUpdaters ..... 88
Einführung ..... 88
Implementierung ..... 88
Ein einfaches Beispiel ..... 88
Selber machen oder Basisklasse benutzen? ..... 88
Der Konstruktor ..... 89
Das Update ..... 90
Rollout ..... 90
28. Aviator Templates ..... 91
Gestaltung ..... 91
29. Exportieren in eigenen Dateiformaten ..... 93
Warum? ..... 93
Wie? ..... 93

Implementierung ..... 93
Ergebnisse ..... 95
Registrierung/Packaging ..... 95
Weiterführende Informationen ..... 96
30. Graphische Programmierung für beliebige Komponenten ..... 97
Warum? ..... 97
Wie? ..... 97
Formale Spezifikation ..... 97
Metadaten ..... 97
Verbindungsendpunkte ..... 99
VisualComponentSpec ..... 101
Properties ..... 102
Connections ..... 103
Children ..... 103
Technische Spezifikation ..... 104
Simples Beispiel ..... 104
Komplexes Beispiel einschließlich hierarchischer Gliederung ..... 105
Introspector Implementation ..... 111
Metadaten ..... 112
Verbindungsendpunkte ..... 114
Ergebnisse ..... 114
Weiterführende Informationen ..... 115
31. Service Provider Infrastructure ..... 116
Einleitung ..... 116
Module ..... 116
Überblick ..... 116
Interface ..... 116
Beispielimplementierung ..... 116
Verbindungen ..... 117
Überblick ..... 117
Interface ..... 117
Beispielimplementierung ..... 118
32. Modul-Wrapper ..... 120
Einleitung ..... 120
Die Idee ..... 120
Anforderungen an ModuleWidgetWrapper ..... 121
GUI-Integration ..... 121
Link-Management ..... 121
Persistenz ..... 121
Message-Management ..... 121
Die Einbindung ..... 121
Die Basisklasse ..... 122
Abstrakte Methoden, die überschrieben werden müssen ..... 122
Methoden der Basisklasse, die überschrieben werden sollten ..... 123
33. Unter der Haube ..... 127
Kommunikation ..... 127
Grundlegendes ..... 127
Context (Mandantenfähigkeit) ..... 130
Basisklassen ..... 132
ModuleBase ..... 132
ResetableModuleBase ..... 133
BeanContextChildModuleBase ..... 133
CommunicationTemplate ..... 133
VariableNumberofInputsVisualization ..... 133

ThreadingModuleBase ..... 133
ThreadingBeanContextChildModuleBase ..... 134
ThreadingResetableModuleBase ..... 134
ThreadingCommunicationTemplate ..... 134
HostPortCommunicationTemplate ..... 134
StartStopModule ..... 134
StartStopModuleWithDoOnce ..... 135
JaxbBase ..... 135
MapMessageModule ..... 135
RemoteModule ..... 135
A. Quellcode ..... 136
Ein einfaches Modul ..... 136
Eine JMX MBean als Modul ..... 137
Interface ..... 137
Modul ..... 137
Ein Modul mit Algorithmusbearbeitung im eigenen Thread ..... 139
Parallele Verarbeitung innerhalb eines Moduls ..... 140
Benutzung eines Dienstes aus dem BeanContext ..... 142
Bereitstellung eines Dienstes für einen BeanContext ..... 144
Service ..... 144
Implementierung ..... 145
ServiceProvider ..... 145
Generics ..... 147
Variable Anzahl von Inputs ..... 147
Variable Module zur Visualisierung ..... 148
Variable Module ..... 149
Variable Module für User Eingaben ..... 150
Module zur Filterung ..... 152
Implementierung einer alternativen Bedienoberfläche ..... 152
Modul ..... 152
Angepasster JToggleButton ..... 153
Actions für Module ..... 154
SocketOut ..... 155
Start/Stop ..... 157
Start/Stop mit Einzelschritt ..... 158
Enterprise JavaBeans als Module ..... 159
Module mit geänderter Kommunikationsmetapher ..... 160
Verteiltes Arbeiten (Remoting) ..... 161
Modul ..... 161
Interface ..... 162
Implementierung ..... 163
Getaktete Module ..... 163
StateUpdater ..... 164
DemoWorkspaceExporter ..... 165
B. BeanContext Services ..... 167
LoggingConfig ..... 167
Einsatz ..... 167
Methoden ..... 167
DialogParentFrame ..... 167
Einsatz ..... 167
Methoden ..... 167
IDProvider ..... 168
Einsatz ..... 168
Methoden ..... 168

Notification ..... 168
Einsatz ..... 168
Methoden ..... 168
ReportingEngine ..... 169
Einsatz ..... 169
Methoden ..... 169
WorkspaceAPI ..... 170
Einsatz ..... 170
Methoden ..... 170
Bonjour ..... 171
Einsatz ..... 171
Methoden ..... 171
JSMandantManager ..... 171
Einsatz ..... 171
Methoden ..... 172
BackgroundExecutor ..... 172
Einsatz ..... 172
Methoden ..... 172
ApplicationServer ..... 173
Einsatz ..... 173
Methoden ..... 174
Environment ..... 174
Einsatz ..... 174
Methoden ..... 174
BeanContextServices ..... 175
Einsatz ..... 175
..... 175
C. Erstellen von Komponenten für dWb+ mittels Maven ..... 176
Einleitung ..... 176
Module ..... 176
Service Provider Interface ..... 179
OSGI-Bundles ..... 181
D. Fragen und Antworten ..... 183

# Abbildungsverzeichnis

1.1. Modul mit geöffnetem Parameterdialog ..... 2
9.1. Generisches Modul vor und nach Verbindung des Inputs ..... 36
10.1. Beispiel für ein Modul mit variabler Anzahl von Eingängen ..... 39
11.1. Beispiel für den Parameterdialog eines Modules zur Visualisierung mit variabler Anzahl von Eingängen ..... 42
12.1. Variables Modul direkt nach Hinzufügen zum Arbeitsbereich ..... 49
12.2. Variables Modul nach Hinzufügen dreier weiterer Ausgänge ..... 49
29.1. Beispielworkspace zur Demonstration einer einfachen Export-Komponente ..... 95
30.1. Beispiel der Komponente aus dem oben angegebenen simplen Fragment der Umsetzung der technischen Spezifikation ..... 114
30.2. Beispiel der Komponente aus dem oben angegebenen simplen Fragment der Umsetzung der technischen Spezifikation ..... 115
33.1. Beispielworkspace zur Demonstration der Kopplung zweier Module ..... 127
33.2. Innere Abläufe bei der Kopplung zweier Module ..... 128
33.3. Innere Abläufe bei der Kopplung zweier Module als UseCase-Diagramm ..... 129
33.4. Innere Abläufe bei der Kopplung zweier Module als Sequenzdiagramm ..... 129

# Tabellenverzeichnis

18.1. Actions für das Verbindungsmanagement ..... 61
19.1. Actions zum Starten und Stoppen der Verarbeitung ..... 65
19.2. Actions zum Starten und Stoppen der Verarbeitung ..... 67

# Liste der Beispiele

32.1. Beispiel für die Persistierung eines Konfigurationsparameters für einen `ModuleWrapper`... 122
32.2. Beispiel für das Löschen eines Konfigurationsparameters für einen `ModuleWrapper` nach der Serialisierung ..... 123
32.3. Beispiel für einen Konstruktor für `ModuleWrapper` ..... 123
32.4. Beispiel das Management am `ModuleWrapper` eingehender Daten ..... 124
32.5. Beispiel für das Versenden eines Ausgabedatums durch einen `ModuleWrapper` ..... 125
32.6. Beispiel für das Übernehmen eines Konfigurationsparameters für einen `ModuleWrapper` nach der Deserialisierung ..... 126

# Kapitel 1. Einleitung und Überblick

## dWb+

Diese Anwendung und das Framework, auf das diese aufbaut, erlaubt es, schnell und ohne Quelltextkontakt Datenverarbeitungseinheiten aufzubauen, zu testen und umzukonfigurieren. Die Grundbestandteile einer solchen Lösung auf Basis des dWb+ sind Module, die miteinander verschaltet werden. Diese Verschaltungen oder Verbindungen zwischen Modulen bestimmen den Datenfluß in der Anwendung. Module und deren Verbindungen machen zusammen den sogenannten Workspace aus. Der Workspace kann gespeichert und später wieder geladen werden. Er entspricht damit in etwa dem, was auch ein traditionelles Programm ausmacht. Bei einem Programm werden die Anweisungen und ihre Abfolge definiert, in einem Workspace geschieht das auf der Ebene ganzer Funktionsblöcke. Ein weiterer Unterschied zum traditionellen Programm ist, dass dort noch die Reihenfolge der Anweisungen explizit angegeben wird und damit eine Sequenz von Arbeitsschritten vorgegeben wird. In einem Workspace ist das nicht mehr der Fall: es werden lediglich Module und deren Beziehungen untereinander definiert - nicht die Reihenfolge in der die Module aktiv sein sollen. Der Zeitpunkt, zu dem ein Modul aktiv wird und arbeitet wird zur Laufzeit ganz allein durch die Tatsache bestimmt, dass neue Daten an den Eingängen des Moduls vorliegen. Ist das der Fall, nimmt das Modul die Daten entgegen, verarbeitet sie und produziert eventuell neue Daten, die an alle ihm nachgeschalteten Module weitergeleitet werde, wo sie eventuell wieder für den Start der Datenverarbeitung sorgen.

## Module

Module kann man im traditionellen Softwareentwicklungszyklus zum Beispiel mit Prozeduren oder Funktionen vergleichen: Sie haben einen oder mehrere Inputdaten, stellen eine wohldefinierte Verarbeitungsleistung dar, die wieder ein oder mehrere Resultate erzeugt. Damit kann ein solches Modul durch drei Dinge vollständig charakterisiert werden:

### Gliederungseinheiten eines Moduls

Input
: Dieser Funktionsblock fasst alle Verknüpfungspunkte (Slots) zusammen, durch die das Modul Daten von anderen Modulen empfangen kann. Jeder einzelne Slot ist durch einen Datentyp und einen Namen spezifiziert. Der Name existiert einmal als abstrakter Name. Dieser Name bleibt immer gleich und dient dem Herstellen der Verbindungen beim Einladen eines Workspace. Darüber hinaus existiert ein konkreter oder Displayname. Dieser kann sich (zum Beispiel abhängig von der eingestellten Benutzersprache des Betriebssystems) ändern. Er ist in der Programmoberfläche des dWb+ als Beschriftung des Slots sichtbar. Die Aufteilung in abstrakten und konkreten Namen ermöglicht die Weitergabe von Workspaces in einem internationalen Umfeld - jeder Anwender sieht verständliche Beschriftungen an den Slots und doch funktioniert der Workspace immer gleich. Wird kein konkreter Name spezifiziert, wird der abstrakte an seiner Statt verwendet.

Algorithmus
: Der Algorithmus ist die Vorschrift zur Datenverarbeitung, die das Modul umsetzen soll.

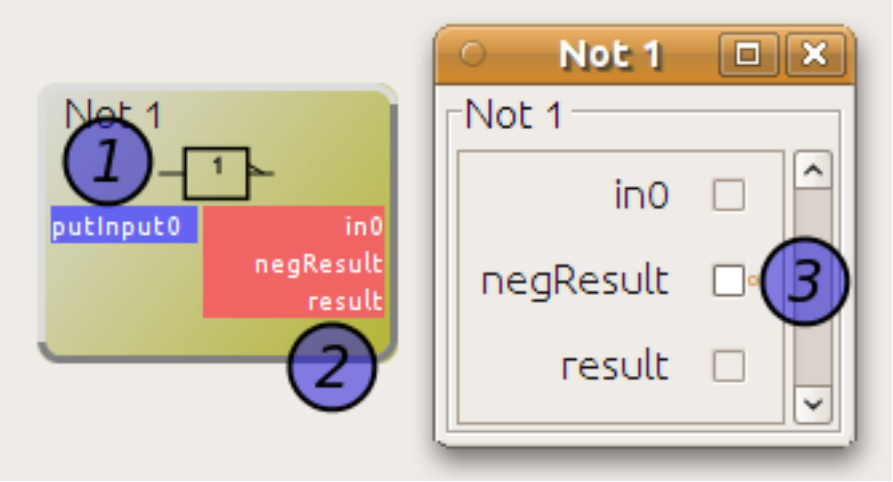
Output
: Dieser Funktionsblock fasst alle Verknüpfungspunkte (Slots) zusammen, durch die das Modul Daten an andere Modulen versenden kann. Jeder einzelne Slot ist durch einen Datentyp und einen Namen spezifiziert. Der Name existiert einmal als abstrakter Name. Dieser Name bleibt immer gleich und dient dem Herstellen der Verbindungen beim Einladen eines Workspace. Darüber hinaus existiert ein konkreter oder Displayname. Dieser kann sich (zum Beispiel abhängig von der eingestellten Benutzersprache des

Betriebssystems) ändern. Er ist in der Programmoberfläche des dWb+ als Beschriftung des Slots sichtbar. Die Aufteilung in abstrakten und konkreten Namen ermöglicht die Weitergabe von Workspaces in einem internationalen Umfeld - jeder Anwender sieht verständliche Beschriftungen an den Slots und doch funktioniert der Workspace immer gleich. Wird kein konkreter Name spezifiziert, wird der abstrakte an seiner Statt verwendet.

# Konfiguration von Modulen

Die Module bieten wie bereits erwähnt bestimmte Algorithmen zur Datenverarbeitung an. Oft ist es so, dass diese Algorithmen über Parameter verfügen, mit denen man ihr Verhalten in gewissen Grenzen steuern kann. Solche Konfigurationsparameter kann man als Inputs sehen, die den zu verarbeitenden Daten völlig gleichberechtigt sind. Im dWb+ wird eine andere Herangehensweise gewählt: Die Werte für die Konfigurationsdaten kommen nicht von anderen Modulen, sondern vom Anwender - daher existieren für diese Daten auch keine Inputs an den Modulen, sondern Bedienpanels oder Formulare, über die diese Daten angepasst werden können. Diese Formulare kann der Anwender öffnen, wenn er auf den Titel des jeweiligen Moduls doppelklickt.

**Abbildung 1.1. Modul mit geöffnetem Parameterdialog**




# Visuelle Entsprechungen in dWb+

Abbildung 1.1, „Modul mit geöffnetem Parameterdialog“ zeigt ein Modul neben seinem geöffneten Parameterdialog. Man sieht die blau dargestellten Eingänge und die roten Ausgänge am Modul ebenso wie das Formular zur Parametrisierung.

# Umsetzung in Java

## Allgemeines

Module folgen stets den JavaBeans-Konventionen. Das bedeutet, dass Module Klassen sind, die

- als public deklariert sind,
- einen parameterlosen public Konstruktor haben und
- nicht abstrakt sind.

# Inputs

Die Inputs werden definiert, indem Methoden mit einer wohldefinierten Signatur geschrieben werden. Die Methoden, die von dWb als Inputs erkannt werden

- sind public,
- haben keinen Rückgabewert,
- sind nicht abstrakt und
- erwarten genau einen Parameter - der legt auch gleichzeitig den Typ des jeweiligen Inputs fest.

Jede dieser Methoden muss darüber hinaus den Algorithmus - die Verarbeitung - des Modules starten, wenn sie aufgerufen wird und also das Modul am entsprechenden Input Daten erhält. Das in Abbildung 1.1, „Modul mit geöffnetem Parameterdialog“ (1) dargestellte Modul besitzt also genau eine Methode, auf die die genannten Kriterien zutreffen.

Dabei ist zu beachten, dass auf diese Weise nur die Methoden in die Produktion der Slots einbezogen werden, die genau in der betreffenden Klasse deklariert wurden. Methoden der direkten und aller weiteren Oberklassen werden nicht berücksichtigt. Um das zu erreichen, ist ein Modul um folgenden Code zu erweitern:

```
import de.netsysit.util.beans.InterfaceFactory;
//...
static
{
 InterfaceFactory.setSuperclassAssociationForEventDispatchThread(
  Module.class,Stop.class);
}
```

Diese Anweisung sorgt dafür, dass alle Methoden der Modulklasse Module und die aller Oberklassen bis zur Klasse Stop in die Generierung der Slots einbezogen werden. Die Methoden der Klasse Stop werden nicht mit einbezogen. Diese Klasse ist also die erste in der Hierarchie, die dann ignoriert wird.

# Algorithmus

Der Algorithmus kann in vom Programmierer des Moduls völlig freier art und Weise implementiert werden. Es gibt keine Einschränkungen hinsichtlich Coding-Policy oder Art der Implementierung. Es sollte lediglich darauf geachtet werden, dass dWb+ bereits einige Klassenbibliotheken mitbringt. Bei der Entwicklung sollte man die Quelltexte neu erstellter Module, die die gleichen Bibliotheken nutzen unbedingt gegen die Bibliotheksversionen, die dWb+ mitbringt, kompilieren um schwer zu findende Classpath-Probleme von vornherein zu vermeiden.

# Outputs

Als Outputs werden von dWb+ alle Properties erkannt, die eine get- (Lese-)Methode haben. Das bedeutet auch, dass, selbst wenn es vom Algorithmus nicht benötigt wird, für jedes Ergebnis des Algorithmus eine get-Methode implementiert werden muss - daran erkennt der Java-Introspector das Vorhandensein eines Property!

Das Versenden der Ergebnisse an nachgeschaltete Module erfolgt mittels PropertyChangeEvents. Daher muss jedes Modul, das Ergebnisse an andere Module weitergeben will, entsprechende Methoden

zum PropertyChangeListener-Management anbieten. Weiterhin ist es wichtig, die entsprechenden PropertyChangeEvents auch wirklich zu erzeugen und zu versenden! Das in Abbildung 1.1, „Modul mit geöffnetem Parameterdialog“ (2) dargestellte Modul besitzt also genau drei Properties, auf die die genannten Kriterien zutreffen.

Dabei ist zu beachten, dass auf diese Weise nur die Properties in die Produktion der Slots einbezogen werden, die genau in der betreffenden Klasse deklariert wurden. Properties der direkten und aller weiteren Oberklassen werden nicht berücksichtigt. Um das zu erreichen, ist ein Modul um folgenden Code zu erweitern:

```
import de.netsysit.util.beans.InterfaceFactory;
...
static
{
 InterfaceFactory.setSuperclassAssociationForEventDispatchThread(
  Module.class,Stop.class);
}
```

Diese Anweisung sorgt dafür, dass alle Properties der Modulklasse Module und die aller Oberklassen bis zur Klasse Stop in die Generierung der Slots einbezogen werden. Die Properties der Klasse Stop werden nicht mit einbezogen. Diese Klasse ist also die erste in der Hierarchie, die dann ignoriert wird.

# Konfigurationsparameter

Die Konfigurationsparameter werden über JavaBeans-Properties realisiert. Alle Parameter sollten als bound Property ausgeführt sein - also bei Änderung entsprechende PropertyChangeEvents versenden. Wegen ihrer Rolle als Konfigurationsparameter müssen diese Properties sowohl les- wie auch schreibbar sein.

Diese Variante hat folgende Vorteile: Der Programmierer muss keinen Code zum Speichern und Wiederherstellen des Zustandes bei Laden und Speichern des Workspace schreiben: dWb+ sorgt dafür, dass der Status eines Moduls (die Menge aller Properties) automatisch korrekt in die Workspacedatei geschrieben wird und beim Einladen der Status des jeweiligen Modules wiederhergestellt wird.

Weiterhin wird dadurch erreicht, dass der Programmierer eines Moduls nicht selbst ein Formular zur Änderung der Konfigurationsparameter erstellen muss: dWb+ bringt ein Framework mit, das aus einer JavaBean, beziehungsweise ihren Properties zur Laufzeit Formulare erzeugen kann. Dieses Framework wird zur Erstellung der Parameterfenster jedes Modules benutzt, das selbst nicht explizit ein eigenes Formular zur Parametrierung anbietet. Das in Abbildung 1.1, „Modul mit geöffnetem Parameterdialog“ (3) dargestellte Modul besitzt also genau eine Property, auf die die genannten Kriterien zutreffen.

# Diagnose

Die hier geschilderten Möglichkeiten sind rein optional, da sie auch über das Konzept einer normalen JavaBean hinausgehen. Die Basisklassen zur Erstellung von Modulen bieten Methoden an, um den Anwender über Probleme und kritische Fehler in einem Modul zu unterrichten. Nutzt man diese, wird der Anwender über die Einblendung von Symbolen über das Auftreten informiert. Diese Symbole zeigen als Tooltip die zugehörige Botschaft. Fehler können von Warnungen nicht überschrieben werden: Trat ein Fehler auf, wird das Symbol dafür mit dem entsprechenden Tooltip auch dann weiterhin angezeigt, wenn danach weitere Warnungen gemeldet wurden.

Der Anwender kann sich dann eine Liste aller derartigen Meldungen anzeigen lassen und diese Liste auch wieder löschen. Um diesen Komfort anzubieten, ist es aber nötig, ein neu entwickeltes Modul eng an dWb+

zu binden, da hier spezifische APIs genutzt werden, die es nur innerhalb des Frameworks gibt. Tatsächlich muß das Modul von einer der BeanContextAware-Basisklassen abgeleitet werden, um die Informationen über das ModuleWidget anzeigen zu können.

Die hierzu definierten Basisklassen-Methoden nehmen als erstes Argument einen Logger aus der Log4J-Bibliothek entgegen - Diese Referenz darf auch null sein. Das zweite Argument ist ein String und die eigentliche Meldung. Optional existiert als drittes Argument ein Array von Object, dessen Inhalte als Argumente in die eigentliche Meldung integriert werden. Dies geschieht über java.text.MessageFormat.

```
import org.apache.log4j.Logger;
...
 public void warn(Logger logger, String message);
 public void warn(Logger logger, String message,Object... args);
 public void error(Logger logger, String message);
 public void error(Logger logger, String message,Object... args);
```

## Zusammenfassung/Schlußfolgerung

Zur Erstellung eines neuen Moduls ist es also lediglich nötig, eine JavaBean zu erstellen, deren Properties auf Änderungen mit einem PropertyChangeEvent eventuelle Listener informieren und entsprechende Methoden zu implementieren, die die Inputs in das Modul modellieren.

# Entwicklungsschritte

Module können iterativ entwickelt werden: Die Erstellung und das Compilieren der einzelnen Klassen erfolgt mit einer beliebigen IDE (oder sogar per Kommandozeile - da sind den persönlichen Vorlieben der Entwickler keine Grenzen gesetzt). Wichtig ist, daß dWv+ die Klassen in einem Jar-Archiv benötigt, um sie nutzen zu können. Dieses Jar-Archiv muss dann in den Ordner kopiert werden, in dem dWb+ nach Modulen sucht wie unter „modules“ in Anhang A, *Verzeichnis-Layout* beschrieben. Diese beiden Schritte - das Erzeugen und das Kopieren des Archivs - können mit entsprechenden Entwicklungswerkzeugen ebenfalls automatisiert werden.

Zum Test werden die Module in einen Workspace in dWb+ gezogen und mit entsprechenden Modulen verbunden, die definierte Testdaten liefern und es gestatten, die Ergebnisse zu analysieren.

Stellt man beim Test Fehler fest, oder möchte man nach erfolgreichen Tests weitere Funktionalität zu den entwickelten Modulen hinzufügen, muß man nach der erneuten Erstellung und Kopie des Jar-Archivs dWb + nicht neu starten: dWb+ bietet die Möglichkeit, neue Versionen von Modulen zur Laufzeit zu laden. Bestehende Modulinstanzen erfahren dadurch kein Upgrade, nur neu angelegte Instanzen zeigen die an der Klasse vorgenommenen Änderungen.

Dazu muss man lediglich den entsprechenden Knopf über dem Modulbaum betätigen wie unter „Actions“ in Kapitel 3, *Dock* beschrieben. Zur Zeit ist es so, daß dabei alle Module neu geladen werden - bei einer großen Modulsammlung kann es also durchaus einige Sekunden dauern, bis alle Module analysiert sind.

# Kapitel 2. BeanInfo-Verwendung in dWb+

## Internationalisierung

Wie bereits weiter vorn beschrieben haben Inputs und Outputs abstrakte und konkrete Namen. Diese Unterteilung kommt besonders der Internationalisierung zugute. Damit ist es möglich, einen Workspace, der auf einem System mit englischer Sprachunterstützung erstellt wurde, auf einem System mit beispielsweise deutscher Sprachunterstützung zu nutzen. Dabei sieht man - wenn die Module entsprechend implementiert wurden - auf dem deutschen System die deutschen Bezeichnungen der Slots und auf dem englischen System entsprechend die Beschriftungen dieser Sprache.

Um diese Mechanismen nutzen zu können, muss derjenige, der ein Modul für dWb zur Verfügung stellen möchte, eine BeanInfo-Klasse für jedes Modul implementieren. Die folgenden Abschnitte gehen genauer darauf ein, welche Felder in der BeanInfo wie von dWb+ benutzt werden.

## Slots

### Beschriftung

Der Titel eines Slots wird über die Eigenschaft displayName des zugehörigen PropertyDescriptors gesteuert. Hier kann man durch Benutzung eines ResourceBundle Internationalisierung unterstützen. Der Titel wird als Beschriftung des Slots in der Darstellung des Moduls im Workspace verwendet. Wird er nicht spezifiziert, wird einfach der Name der Property benutzt.

### Tooltips

Der Tooltip eines Slots wird über die Eigenschaft shortDescription des zugehörigen PropertyDescriptors gesteuert. Hier kann man durch Benutzung eines ResourceBundle Internationalisierung unterstützen. Wird er nicht spezifiziert, wird als Tooltip der Typ der zugehörigen Property angezeigt.

Der Tooltip kann als HTML-formatierter Text spezifiziert werden - entsprechende Formatierungen werden bei der Darstellung im Tooltip berücksichtigt.

### StateUpdaters

Der Tooltip eines OutputSlots kann darüber hinaus dafür benutzt werden, die Daten, die darüber versendet werden, zu visualisieren - vergleiche dazu auch „Tooltips für Outputs“. Dazu werden spezielle Implementierungen, sogenannte StateUpdater eingesetzt. Die Zuordnung der einzelnen Implementierungen zu den Typen der jeweiligen Output-Slots ist flexibel änderbar und wird über eine zentrale Registry geregelt wie in „Einführung“ in Kapitel 27, *StateUpdaters* beschrieben.

Darüber hinaus besteht die Möglichkeit, diese Zuordnung über entsprechende, den einzelnen Properties zugeordneten, Metadaten zu steuern.

### Automatisches Erstellen von Verbindungen

Die Markierung von Ein- oder Ausgängen für die automatische Etablierung von Verbindungen geschieht über das Setzen des entsprechenden Attributes an PropertyDescriptor (Ausgang), beziehungsweise MethodDescriptor (Eingang). Der Name des Attributes lautet `AUTOCONNECTALLOWEDATTRIBUTE`.

## Maximale Anzahl von Verbindungen

Ein- und Ausgänge können mit einer maximalen Anzahl gleichzeitig bestehender Verbindungen beaufschlagt werden - das System verhindert nach Erreichen dieser Anzahl mit dem jeweiligen Slot verbundener Verbindungen die Erstellung neuer Verbindungen an dem jeweiligen Slot. In diesem Fall können neue Verbindungen mit diesem Slot erst wieder hergestellt werden, wenn vorher mindestens eine der bestehenden entfernt wurde. Die Spezifikation der erlaubten Maximalanzahl von Verbindungen für den jeweiligen Slot geschieht über das Setzen des entsprechenden Attributes an PropertyDescriptor (Ausgang), beziehungsweise MethodDescriptor (Eingang). Der Name des Attributes lautet `MAXCONNECTIONSALLOWED`. Der Wert muss ein `java.lang.Integer` sein. Erlaubt sind nur Werte größer als `0`.

# Modultitel

Der Titel des Moduls wird über die Eigenschaft displayName des BeanDescriptors gesteuert. Hier kann man durch Benutzung eines ResourceBundle Internationalisierung unterstützen. Der Titel wird als Text im Modulbaum und als Titel der Module im Workspace verwendet. Wird er nicht spezifiziert, wird einfach der Name der Klasse ohne vorangestelltes Paket benutzt.

# Modulbeschreibung

Die Beschreibung des Moduls wird über die Eigenschaft shortDescription des BeanDescriptors gesteuert. Hier kann man durch Benutzung eines ResourceBundle Internationalisierung unterstützen. Der Titel wird als Tooltip im Modulbaum und im Parameterpanel der Module im Workspace verwendet. Wird er nicht spezifiziert, wird kein Tooltip angezeigt und das Parameterpanel bleibt unverändert.

Die Beschreibung kann als HTML-formatierter Text spezifiziert werden - entsprechende Formatierungen werden bei der Darstellung sowohl im Tooltip wie im Parameterpanel berücksichtigt.

# Icon

Das Icon des Moduls wird über das Resultat der Methode getIcon des BeanDescriptors gesteuert. Der Methode wird als Argument ICON_COLOR_32x32 übergeben. Das Icon wird im Modulbaum verwendet. Ist unter diesem Schlüssel kein Icon definiert, wird kein Icon im Modulbaum angezeigt.

Weiterhin wird ein hier definiertes Icon zwischen den Input- und den Output-Slots des ModulWidgets angezeigt. Dieser Bereich dient auch der Anzeige für das Auftreten von Problemen oder kritischen Fehlern in Modulen. Dafür wird ein entsprechendes Symbol über das Icon gelegt.

Ist unter diesem Schlüssel kein Icon definiert, wird dieser Bereich durch einen unsichtbaren Platzhalter belegt, der dann als Grundlage für die Benachrichtigungssymbole dient.

# Layer

Module werden in den Layer platziert, der beim Hinzufügen des Moduls ausgewählt ist. Ist hier keine explizite Auswahl getroffen ("default" ist ausgewählt), wird im `BeanDescriptor` nach einem Schlüssel namens `de.netsysit.dataflowframework.ui.ModuleWidget.DEFAULTLAYERTITLE` gesucht. Wird dieser gefunden, wird der damit assoziierte Wert als Name eines Layers interpretiert. Existiert ein Layer diesen Namens noch nicht, wird er erzeugt. Anschließend wird das neu hinzugefügrt Modul diesem Layer zugeordnet.

# Verbergen von Properties

Möchte man Properties sowohl im Parameterdialog wie auch als Ausgangsslots verbergen, setzt man die Property `hidden` am zugehörigen PropertyDescriptor auf `true.`

Möchte man hingegen eine Property nur als Output-Slot, nicht jedoch im Parameterdialog auftauchen lassen, setzt man am zugehörigen PropertyDescriptor den Wert der symbolischen Konstante `de.netsysit.util.beans.InterfaceFactory.HIDDEN` auf den String `"true"`. Groß- und Kleinschreibung sind dabei egal.

# Kapitel 3. Ein einfaches Beispiel

## Funktionsbeschreibung

Dieses einfache Modul soll eine Verarbeitung realisieren, die Zeichen in einem String zählt. Das bedeutet, dass wir einen Input vom Typ String haben werden, weiterhin einen Output vom Typ int und - um die Sache interessanter zu machen - eine Konfigurationsoption, die bestimmt, ob Leerzeichen mitgezählt werden sollen. Den vollständigen Code für dieses Modul kann man im Anhang A, *Quellcode* in „Ein einfaches Modul“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

# Selber machen oder Basisklasse benutzen?

Wie bereits im vorhergehenden Kapitel beschrieben, hängt die nahtlose Integration neuer Module in dWb + vor allem vom korrekten Versenden der PropertyChangeEvents zusammen. Der Programmierer kann dazu natürlich die gesamte Infrastruktur selbst implementieren. Eine zweite Möglichkeit wäre, die Klasse PropertyChangeSender im Paket java.beans zu nutzen und die Arbeit im Zusammenhang mit Events an diese zu delegieren.

Die mit Sicherheit einfachste Lösung jedoch besteht darin, eine der bereits vom dWb+ bereitgestellten Basisklassen zu benutzen. Diese Methode scheitert natürlich, wenn das Modul aus fachlichen Gründen von einer anderen Klasse abgeleitet werden muss.

Wir werden für dieses Beispiel jedoch die einfachste Variante wählen und unser neues Modul von ModuleBase im Paket de.netsysit.dataflowframework.modules ableiten. Das hat unter anderem zwei Vorteile: zum einen muss man dann nicht mehr selbst die Methoden zum Listener-Management implementieren (add/remove), zum anderen existieren in dieser Klasse bereits eine Menge überladener send-Methoden zum Versenden der PropertyChangeEvents.

```
import de.netsysit.dataflowframework.modules.ModuleBase;

public class CharacterCounter extends ModuleBase
{
}
```

# Input

Der Input wird über die Implementierung einer entsprechenden Methode definiert - in unserem Fall hat diese Methode die Signatur `public void input(java.lang.String in)`. Im Körper dieser Methode wird nichts anderes getan, als den Algorithmus zu starten - weiter unten dazu mehr. Wir vereinbaren darüber hinaus eine Instanzvariable, in der der letzte String gespeichert wird - das erlaubt es, auch auf Änderungen der Konfigurationsoptionen sofort mit einer erneuten Ausführung des Algorithmus zu reagieren. Möchte man den Algorithmus jedoch nur dann starten, wenn wirklich frische Inputdaten zur Verfügung stehen und nicht bei Änderung der Konfiguration, kann man diese Zwischenspeicherung weglassen.

```
private String lastInput;

public void input(String in)
{
 lastInput=in;
 countCharacters();
}
```

# Konfiguration

Da wir als einzigen Konfigurationsparameter eine Umschaltung für die Berücksichtigung von Leerzeichen beim Zählen vorgesehen haben, legen wir in der Klasse nun noch ein Property vom Typ boolean an. Dieses Property soll lese- und schreibbar sein - daher muss es über eine get- und set-Methode verfügen. Wichtig: die set-Methode muss einen entsprechenden PropertyChangeEvent versenden! Schließlich rufen wir in der set-Methode wie weiter oben schon angedeutet den Algorithmus auf.

```
private boolean ignoreSpaces;

public boolean isIgnoreSpaces()
{
 return ignoreSpaces;
}

public void setIgnoreSpaces(boolean ignoreSpaces)
{
 boolean old=isIgnoreSpaces();
 this.ignoreSpaces = ignoreSpaces;
 send("ignoreSpaces",old,isIgnoreSpaces());
}
```

# Output

Wir definierten einen Output in der Funktionsbeschreibung des Moduls. Die Implementierung geschieht über eine Property des entsprechenden Typs, die lediglich über eine get-Methode verfügt. Weiterhin wird eine Instanzvariable angelegt, die den letzten berechneten Wert aufnimmt, da man zum Versenden eines PropertyChangeEvents ja immer den alten und den neuen Wert benötigt.

```
private int characterCount;

public int getCharacterCount()
{
 return characterCount;
}
```

# Algorithmus

Der Algorithmus für unser Beispiel ist denkbar einfach - der Programmierer muss bei der Implementierung lediglich darauf achten, dass er die Einstellungen aller Konfigurationsparameter korrekt umsetzt und PropertyChangeEvents für alle gewonnenen Resultate versendet.

```
private void countCharacters()
{
 int old=getCharacterCount();
 //Algorithmus-Implementierung hier
 //nicht vergessen, characterCount zuzuweisen!
 send("characterCount",old,getCharacterCount());
}
```

# Kapitel 4. Eine JMX MBean als Modul

## Funktionsbeschreibung

Dieses einfache Modul soll die Einhaltung eines Schwellwertes überwachen. Der Schwellwert soll vom Anwender spezifiziert werden können. Dies soll sowohl im Parameterfenster des Moduls, wie auch über JMX möglich sein. Bei Überschreiten des Schwellwertes soll ein Signal am Ausgang des Moduls für andere Module erzeugt werden. Darüber hinaus soll in diesem Falls eine JMX-Notification ausgelöst werden. Für die zu überwachende Größe wird ein Input vom Typ Number bereitgestellt, die Signalisierung der Schwellwertüberschreitung erfolgt mittels eines Outputs vom Typ boolean. Die Konfigurationsoption, die den zu überwachenden Schwellwert festlegt, wird ebenfalls vom Typ Number sein. Den vollständigen Code für dieses Modul kann man im Anhang A, *Quellcode* in „Eine JMX MBean als Modul“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

## Selber machen oder Basisklasse benutzen?

Wir werden für die Integration des Moduls in dWb+ die neue Klasse wieder von ModuleBase im Paket de.netsysit.dataflowframework.modules ableiten. Damit es eine für die Benutzung mit JMS taugliche MBean darstellt, muss sie ein Interface implementieren, dessen Name auf MXBean endet. Damit das Modul JMX-Notifications versenden kann, muss es das Interface NotificationEmitter aus dem Namensraum javax.management implementieren:

```
import de.netsysit.dataflowframework.modules.ModuleBase;
import javax.management.AttributeChangeNotification;
import javax.management.ListenerNotFoundException;
import javax.management.MBeanNotificationInfo;
import javax.management.Notification;
import javax.management.NotificationBroadcasterSupport;
import javax.management.NotificationEmitter;
import javax.management.NotificationFilter;
import javax.management.NotificationListener;

public class JMXBean extends ModuleBase implements JMXBeanMXBean
,NotificationEmitter
{
}
```

## MXBean

Das Interface, das festlegt, welche Eigenschaften zur Manipulation über JMX verfügbar sein sollen, sieht wie folgt aus:

```
public interface JMXBeanMXBean
{
 public double getThreshold();
 public void setThreshold(double threshold);
```

```
}
```

# Konfiguration

Da wir als einzigen Konfigurationsparameter den Schwellwert vorgesehen haben, legen wir in der Klasse nun noch ein Property vom Typ double an. Dieses Property soll entsprechend des Interface lese- und schreibbar sein - daher muss es über eine get- und set-Methode verfügen.

```
 private double threshold;

 public double getThreshold()
 {
  return threshold;
 }

 public void setThreshold(double threshold)
 {
  double old=getThreshold();
  this.threshold=threshold;
  send("threshold",old,getThreshold());
 }
```

# Notification Infrastruktur

Jede JMX-Notification, die versendet wird, benötigt eine laufende Nummer. Wir legen daher eine entsprechende Instanzvariable an. Wir implementieren das Interface NotificationEmitter nicht selbst, sondern delegieren an eine Instanz vom Typ NotificationBroadcasterSupport aus dem Namensraum javax.management. Diese wird zunächst im Konstruktor angelegt:

```
private long sequenceNumber=1;
private NotificationBroadcasterSupport notificationBroadcasterSupport;

public JMXBean()
{
 super();
 String[] types = new String[]{
  javax.management.AttributeChangeNotification.ATTRIBUTE_CHANGE
 };
 String name = AttributeChangeNotification.class.getName();
 String description = "Threshold violated";
 MBeanNotificationInfo info = new MBeanNotificationInfo(types, name, description);
 notificationBroadcasterSupport=
  new NotificationBroadcasterSupport(new MBeanNotificationInfo[]{info});
}
```

Anschließend werden die Methoden des Interface entsprechend ausformuliert:

```
public void removeNotificationListener(NotificationListener listener,
   NotificationFilter filter, Object handback) throws ListenerNotFoundException
{
 notificationBroadcasterSupport.removeNotificationListener(listener,
   filter, handback);
}
public void addNotificationListener(NotificationListener listener,
   NotificationFilter filter, Object handback) throws IllegalArgumentException
{
 notificationBroadcasterSupport.addNotificationListener(listener,
   filter, handback);
}
public void removeNotificationListener(NotificationListener listener)
   throws ListenerNotFoundException
{
 notificationBroadcasterSupport.removeNotificationListener(listener);
}
public MBeanNotificationInfo[] getNotificationInfo()
{
 return notificationBroadcasterSupport.getNotificationInfo();
}
```

# Output

Wir definierten einen Output in der Funktionsbeschreibung des Moduls. Die Implementierung geschieht über eine Property des entsprechenden Typs, die lediglich über eine get-Methode verfügt. Weiterhin wird eine Instanzvariable angelegt, die den letzten berechneten Wert aufnimmt, da man zum Versenden eines PropertyChangeEvents ja immer den alten und den neuen Wert benötigt.

```
private boolean state;

public boolean isState()
{
 return state;
}
```

# Input

Der Input wird über die Implementierung einer entsprechenden Methode definiert - in unserem Fall hat diese Methode die Signatur `public void input(java.lang.Number in)`. Im Körper dieser Methode wird nichts anderes getan, als den Schwellwert zu prüfen. Wird er verletzt, wird eine JMS-Notification abgesetzt. Außerdem wird das Ausgangssignal von hier aus über das Versenden von PropertyChangeEvents gesteuert.

```
public void input(Number in)
{
 boolean old=isState();
 state=in.doubleValue()<threshold;
 send("state",old,isState());
```

```
 if(state)
 {
  Notification n = new AttributeChangeNotification(this,
   sequenceNumber++,System.currentTimeMillis(), "State changed",
   "State", "boolean",old, isState());
  notificationBroadcasterSupport.sendNotification(n);
 }
}
```

# Kapitel 5. Arbeiten im Hintergrund (Threads)

## Warum?

Die einfachen Module, die dem Muster folgen, das im vorhergehenden Kapitel vorgestellt wurde, arbeiten im selben Thread wie die GUI. Wenn also der implementierte Algorithmus sehr rechenintensiv ist, kann es dazu führen, dass die GUI sich schlecht bedienen lässt und zum Beispiel Visualisierungen nicht mehr oder nur noch sehr langsam und unregelmäßig aktualisiert werden. Daher sollte man rechenaufwendige Algorithmen in Hintergrundthreads berechnen lassen.

## Ein einfaches Beispiel

Dieses Kapitel wird die Möglichkeiten und Fallstricke der Arbeit mit threaded Modulen am gleichen Beispiel beleuchten, das schon im vorhergehenden Kapitel Pate stand. Den vollständigen Code für dieses Modul kann man im Anhang A, *Quellcode* in „Ein Modul mit Algorithmusbearbeitung im eigenen Thread“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

## Selber machen oder Basisklasse benutzen?

Wie bereits im vorhergehenden Kapitel besteht auch beim Einsatz von Threads die Möglichkeit, alles selbst zu implementieren.

Die mit Sicherheit einfachste Lösung jedoch besteht darin, eine der bereits vom dWb+ bereitgestellten Basisklassen zu benutzen. Diese Methode scheitert natürlich, wenn das Modul aus fachlichen Gründen von einer anderen Klasse abgeleitet werden muss.

Wir werden für dieses Beispiel jedoch die einfachste Variante wählen und unser neues Modul von ThreadedModuleBase im Paket de.netsysit.dataflowframework.modules ableiten. Dies hat zu den bereits im vorhergehenden Kapitel genannten Vorteile bei der Benutzung der Basisklassen von dWb+ den Vorteil, dass man sich nicht um das Thread-Management darum kümmern muss. Das schließt zum Beispiel ein, dass der Thread eines Modules natürlich beendet werden muss, wenn das Modul aus dem Workspace entfernt wird. Außerdem benutzen Module, die von dieser Basisklasse abgeleitet wurden, den Threadpool in dWb+.

Wird von dieser Klasse abgeleitet, ist zu beachten, dass sie keinen parameterlosen Konstruktor besitzt - Der Basisklassenkonstruktor erwartet einen Namen, den dann der gestartete Thread erhält. Im Beispiel benutzen wir den Namen der Modulklasse.

```
import de.netsysit.dataflowframework.modules.ModuleBase;
import de.netsysit.util.threads.CubbyHole;
import de.netsysit.util.threads.SimpleBufferingCubbyHole;

public class ThreadedCharacterCounter extends ThreadingModuleBase
{
 public ThreadingCharacterCounter()
```

```
 {
  super(ThreadingCharacterCounter.class.getName());
 }
}
```

Die Initialisierung des Moduls kann im Konstruktor erfolgen. Möchte der Entwickler dies aus irgendwelchen Gründen so frühzeitig im Lebenszyklus der Instanz nicht tun, steht die Möglichkeit offen, die Methode `setup` zu implementieren. Diese wird nicht bereits bei der Konstruktion der Instanz ausgeführt, sondern erst im Rahmen des Starts des Threads. Das Gegenstück dieser Methode ist die Methode `teardown`, die direkt vor Beendigung des Threads ausgeführt wird - hier kann der Entwickler Code unterbringen, der hinter dem Thread aufräumen soll:

```
 protected void setup()
 {
  //hier Code einfügen, der beim
  //Start des Threads ausgeführt werden soll!
 }
 protected void teardown()
 {
  //hier Code einfügen, der vor
  //Beendigung des Threads ausgeführt werden soll!
 }
```

Diese beiden Methoden müssen nicht implementiert werden - ihre Default-Implementierungen in der Basisklasse sind leer.

Die Kommunikation zwischen dem Thread, der für die Kommunikation der Module untereinander verantwortlich ist und in dessen Kontext zum Beispiel auch die Methoden ausgeführt werden, die die Input-Slots modellieren und dem Thread, der den Algorithmus ausführt, geschieht über sogenannte CubbyHoles. Jede Klasse, die von ThreadingModuleBase erbt, muss daher eine Instanz zur Verfügung stellen, die dieses Interface implementiert. Wir benutzen für dieses Beispiel eine sehr einfache Implementierung.

```
 @Override
 protected CubbyHole createCubbyHole()
 {
  return new SimpleBufferingCubbyHole();
 }
```

Die CubbyHoles sind dazu da, die Threads zu starten: Sobald ein Datum in ein solches CubbyHole geschrieben wird, merkt das der Thread und startet die Abarbeitung der in ihm implementierten Funktionalität. Die CubbyHoles können verschiedene Strategien bereitstellen, um mit dem Fall umzugehen, daß bereits ein neues Datum eingeht, bevor der Thread die Bearbeitung des letzten abgeschlossen hat. Die in unserem Beispiel benutzte speichert eingehende Daten in einer Warteschlange, wenn der Thread noch neschäftigt ist. Sobald der Thread mit der Abarbeitung des letzten Auftrages fertig ist, setzt er seine Arbeit mit dem ersten Element in der Warteschlange fort, falls diese nicht leer ist.

Neben CubbyHoles, die eine bestimmte Strategie festlegen, existiert auch die Möglichkeit, ein

```
de.elbosso.util.threads.MimikryCubbyHole
```

zu verwenden: dieses erlaubt es, für das Modul zur Laufzeit über ein Menü die Strategie zu wählen - siehe dazu auch Tabelle Tabelle 4.11, „Pufferstrategie für eingehende Daten“.

# Input

Der Input wird über die Implementierung einer entsprechenden Methode definiert - in unserem Fall hat diese Methode die Signatur `public void input(java.lang.String in)`. Im Körper dieser Methode wird nichts anderes getan, als den Algorithmus zu starten - weiter unten dazu mehr. Wir vereinbaren darüber hinaus eine Instanzvariable, in der der letzte String gespeichert wird. Der Zugriff auf diese Variable wird über einen Monitor gesteuert, da dWb+ und damit der Transport der Daten zwischen den Modulen und der Algorithmus in verschiedenen Threads ablaufen.

Der Start der Bearbeitung im Thread geschieht durch Senden eines Datenobjektes an das CubbyHole dieses Moduls.

```
private String lastInput;

public void input(String in)
{
 setLastInput(in);
 processData(in);Hierdurch Start des Algorithmus
}

private synchronized String getLastInput()
{
 return lastInput;
}

private synchronized void setLastInput(String lastInput)
{
 this.lastInput = lastInput;
}
```

# Konfiguration

Die Konfiguration bleibt ungeändert im Vergleich zu der Variante ohne Threading - lediglich der Zugriff auf die Property wird über einen Monitor geregelt.

```
private boolean ignoreSpaces;

public synchronized boolean isIgnoreSpaces()
{
 return ignoreSpaces;
}

public synchronized void setIgnoreSpaces(boolean ignoreSpaces)
{
```

```
 boolean old=isIgnoreSpaces();
 this.ignoreSpaces = ignoreSpaces;
 send("ignoreSpaces",old,isIgnoreSpaces());
}
```

# Output

Die Implementierung des Output bleibt ebenfalls bis über die Absicherung über einen Monitor gleich.

```
private int characterCount;

public synchronized int getCharacterCount()
{
 return characterCount;
}

private synchronized void setCharacterCount(int characterCount)
{
 this.characterCount = characterCount;
}
```

# Algorithmus

Der Algorithmus für unser Beispiel ist denkbar einfach - der Programmierer muss bei der Implementierung lediglich darauf achten, dass er die Einstellungen aller Konfigurationsparameter korrekt umsetzt und PropertyChangeEvents für alle gewonnenen Resultate versendet.

Die Implementierung geschieht in der Methode `doWork`. Diese Methode hat einen Parameter - hier bekommt die Methode das Objekt übergeben, das in das CubbyHole geschrieben wurde, um die Ausführung zu starten.

Der Zugriff auf alle Variablen, die mit Inputs, der Konfiguration oder Outputs zu tun haben erfolgt ausschließlich über Monitore aus dem Algorithmus-Thread heraus.

```
@Override
protected void doWork(Object ref) throws InterruptedException
{
 int old=getCharacterCount();
 //Datenzugriff lesend über Monitor:
 java.lang.String data=getLastInput();
 int cc=0;
 //Datenzugriff lesend über Monitor:
 if(isIgnoreSpaces())
 {
  //Algorithmus-Implementierung ohne Leerzeichen hier
 }
 else
 {
  cc=data.length();
```

```
 }
 //Datenzugriff schreibend über Monitor:
 setCharacterCount(cc);
 send("characterCount",old,getCharacterCount());
}
```

# Kapitel 6. Parallele Verarbeitung innerhalb eines Moduls

## Funktionsbeschreibung

Während es in dem in Kapitel 5, *Arbeiten im Hintergrund (Threads)* beschriebenen Szenario darum geht, die aufwändige Abarbeitung einer Aufgabe aus dem Event Dispatch Thread in einen eigenen auszulagern, soll hier eine Möglichkeit vorgestellt werden, eine Verarbeitung mehrfach parallel durchzuführen. Das kann unter anderem dann interessant werden, wenn eine Verarbeitung lange dauert, aber während der Abarbeitung größere Wartezeiten auftreten, wie es etwa bei der Netzwerkkommunikation geschieht.

Die hier beschriebene Variante eines Moduls startet daher für jedes eingehende Datum einen Thread, in dem die Bearbeitung dieses Datums geschieht. Es werden jedoch besondere Maßnahmen ergriffen um sicherzustellen, dass die Ergebnisse der parallelen Threads in der Reihenfolge an die nachfolgenden Module weitergegeben werden, wie die einzelnen Threads gestartet wurden.

Dieses einfache Modul gibt auf einem Slot Nachrichten aus, wenn ein Thread gestartet wurde, wenn einer beendet wurde und wenn das Resultat der Bearbeitung vorliegt. Jede dieser Botschaften enthält die Id des jeweiligen Threads, der die Erzeugungsreihenfolge widerspiegelt. Die erste und die dritte Botschaft sollte stets mit aufsteigenden Ids erscheinen, während die zweite die IDs in zufälliger Reihenfolge anzeigt. Als Input nimmt es eine Zahl, die aussagt, wie lange der jeweils gestartete Thread vor seiner Beendigung schlafen soll. Koppelt man das Modul an einen geeigneten Zufallszahlengenerator sieht man, dass die Nachrichten über das Thread-Ende in zufälliger Reihenfolge kommmen, die Thread-Resultate jedoch in der Reihenfolge, in der die Threads gestartet wurden. Den vollständigen Code für dieses Modul kann man im Anhang A, *Quellcode* in „Parallele Verarbeitung innerhalb eines Moduls“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

## Selber machen oder Basisklasse benutzen?

Für die hier vorgestellte Funktionalität benötigen wir eine Funktionalität aus dem BeanContext. Daher werden für dieses Beispiel unser neues Modul von BeanContextChildModuleBase im Paket de.netsysit.dataflowframework.modules ableiten. Das hat unter anderem zwei Vorteile: zum einen muss man dann nicht mehr selbst die Methoden zum Listener-Management implementieren (add/remove), zum anderen existieren in dieser Klasse bereits eine Menge überladener send-Methoden zum Versenden der PropertyChangeEvents.

```
import de.netsysit.dataflowframework.modules.BeanContextChildModuleBase;
import de.netsysit.util.beans.context.service.BackgroundExecutor;
import java.beans.beancontext.BeanContextServiceAvailableEvent;
import java.beans.beancontext.BeanContextServiceRevokedEvent;
import de.elbosso.util.threads.Workload;
import de.elbosso.util.threads.ParallelSequentialManager;

public class ParallelSequential extends BeanContextChildModuleBase
{
}
```

# BeanContext

Wir benötigen den Zugriff auf einen Service aus dem BeanContext: Der Service heißt BackgroundExecutor aus dem Paket de.netsysit.util.beans.context.service. Wir müssen auf zwei Events reagieren: Immer dann, wenn eine Implementierung dieses Service zur Verfügung gestellt wird und wenn eine Implementierung dieses Service zurückgezogen wird.

Wenn eine Implementierung des Service verfügbar wird, wird eine Instanz der Klasse ParallelSequentialManager aus dem Paket de.elbosso.util.threads erzeugt, wenn der Service zurückgezogen wird, wird diese Instanz ebenfalls vernichtet.

```
private BackgroundExecutor executor;
private ParallelSequentialManager psm;

public void serviceAvailable(BeanContextServiceAvailableEvent bcsae)
{
 super.serviceAvailable(bcsae);
 if(executor==null)
 {
  if (bcsae.getServiceClass()== BackgroundExecutor.class)
  {
   try
   {
    executor =
      (BackgroundExecutor) (bcsae.getSourceAsBeanContextServices()).getService(
      this, this, BackgroundExecutor.class, this, this);
   }
   catch (Exception e)
   {
    executor = null;
   }
   if(executor!=null)
    psm=new ParallelSequentialManager(executor);
  }
 }
}
@Override
public void serviceRevoked(BeanContextServiceRevokedEvent bcsre)
{
 super.serviceRevoked(bcsre);
 if(executor!=null)
 {
  if(bcsre.getServiceClass()==BackgroundExecutor.class)
  {
   executor=null;
  }
 }
}
```

# Workload

Die Garantie, dass die Ausgangssignale in der Reihenfolge versendet werden, in der die Threads gestartet wurden, wird durch Benutzung der Klasse Workload realisiert: Diese implementiert `Runnable`. In der Methode `run` wird die eigentliche Arbeit des Threads implementiert. Jede von Workload abgeleitete Klasse muss darüber hinaus eine Methode implementieren, die `createSequential` heißt. Diese Methode liefert ein `Runnable` zurück, dessen `run`-Methode die Aktionen beinhaltet, die in der Reihenfolge des Eintreffens der Inputs ausgeführt werden müssen.

```
private static int runningWorkloadNumber;

class MyWorkload extends Workload
{
 private int sleep;
 private int id=runningWorkloadNumber++;

 MyWorkload(int input)
 {
  super();
  sleep=input;
 }
 public void run()
 {
  send("msg",null,id+" sleeping for "+sleep+"ms");
  try
  {
   java.lang.Thread.currentThread().sleep(sleep);
  }
  catch (InterruptedException ex)
  {
   ex.printStackTrace();
  }
  send("msg",null,id+" awoke!");
 }
 public Runnable createSequential()
 {
  return new Runnable(){
   public void run()
   {
    send("msg",null,("sequential "+id));
   }
  };
 }
}
```

# Input

Der Input wird über die Implementierung einer entsprechenden Methode definiert - in unserem Fall hat diese Methode die Signatur `public void input(Number input)`. Im Körper dieser Methode wird nichts anderes getan, slz nachzusehen, ob der input nicht null ist un gegenwärtig eine Instanz der

Klasse ParallelSequentialManager existiert. Ist beides der Fall, wird eine Instanz der Klasse MyWorkload erzeugt und dem ParallelSequentialManager zur Abarbeitung übergeben.

```
public void input(Number input)
{
 if((psm!=null)&&(input!=null))
 {
  psm.enqueue(new MyWorkload(input.intValue()));
 }
}
```

# Output

Wir definierten einen Output in der Funktionsbeschreibung des Moduls. Die Implementierung geschieht über eine Property des Typs String, die lediglich über eine get-Methode verfügt.

```
private String msg;

public String getMsg()
{
 return msg;
}
```

# Kapitel 7. BeanContext und BeanContextServices konsumieren

## Warum?

Jedes Modul wird in einen Workspace geladen. Egal ob der Workspace aus einer Datei geladen wird oder ob ein Modul vom Modulbaum auf den Workspace gezogen wird - immer entsteht die Beziehung "das Modul gehört zum Workspace".

Java bietet schon immer an, JavaBeans sensibel für ihre Umgebung zu machen - die dahinter liegenden Konzepte lassen sich über das Schlagwort BeanContext nachlesen. Man kann JavaBeans in die Lage versetzen, zu erkennen wenn sie zu einem bestimmten Container - eben einem BeanContext - hinzugefügt oder daraus entfernt werden.

Das bloße Wissen um die Mitgliedschaft in einem Container ist aber noch keine wirklich interessante Information - das Konzept der JavaBeans im BeanContext geht darüber hinaus: Ein Context kann Dienste - Services - anbieten, die die JavaBeans als Komponenten des BeanContext nutzen dürfen. Der Context benachrichtigt seine JavaBeans über die Verfügbarkeit neuer Services, darüber, dass bestimmte Services nicht mehr existieren und er ermöglicht es den JavaBeans auf die Implementation der einzelnen Services zuzugreifen und sie zu benutzen.

Der Workspace, in dem die Module platziert werden, ist ein solcher BeanContext. Er bietet verschiedene Dienste an.

## Ein einfaches Beispiel

Dieses Kapitel wird die Möglichkeiten und das Vorgehen zur Nutzung von BeanContextServices demonstrieren, indem das einfache Beispielmodul um die Möglichkeit erweitert wird, den Anwender darüber zu informieren, wenn statt eines Strings eine NULL-Referenz als Input übergeben wird.

Diese Benachrichtigung soll über den Systembenachrichtigungsdienst geschehen - also als Sprechblase über dem System-Tray erscheinen.

Das könnte man selbst implementieren - oder einen Dienst nutzen, der im BeanContext - also im Workspace - zur Verfügung steht. Eine Übersicht über die im dWb+ mitgelieferten Dienste ist im Anhang B, *BeanContext Services* zu finden.

Den vollständigen Code für dieses Modul kann man im Anhang A, *Quellcode* in „Benutzung eines Dienstes aus dem BeanContext“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

## Selber machen oder Basisklasse benutzen?

Die Infrastruktur zur korrekten Integration eines Modules in einen Workspace, der zeitgleich ein BeanContext ist, ist recht aufwändig zu implementieren: man muss das Modul im Context registrieren, Handler für die verschiedenen Ereignisse etablieren,....

Es ist einfacher, eine der bereits vom dWb+ bereitgestellten Basisklassen zu benutzen, als diese Infrastruktur selber zu schaffen. Diese Methode scheitert natürlich, wenn das Modul aus fachlichen Gründen von einer anderen Klasse abgeleitet werden muss. Eine entsprechende Basisklasse existiert als threaded Variante und ebenso ohne Hintergrundthread zur Ausführung des Algorithmus.

Wir werden für dieses Beispiel einfachste Variante wählen und unser neues Modul von BeanContextChildModuleBase im Paket de.netsysit.dataflowframework.modules ableiten. Das führt dazu, dass einige Methoden überschrieben werden müssen, um unsere Anforderungen umzusetzen. Dieses Kapitel baut auf dem einfachen Beispiel ohne Threading auf - es werden allerdings nur die Erweiterungen für das Arbeiten mit dem BeanContext präsentiert - die komplette Klasse ist im Anhang zu finden.

```
import de.netsysit.dataflowframework.modules.BeanContextChildModuleBase;
import de.netsysit.util.beans.context.service.Notification;
import java.beans.PropertyVetoException;
import java.beans.beancontext.BeanContext;
import java.beans.beancontext.BeanContextServiceAvailableEvent;
import java.beans.beancontext.BeanContextServiceRevokedEvent;

public class BeanContextAwareCharacterCounter extends
 BeanContextChildModuleBase
{
}
```

# Dienst nutzen

Wir fangen das Beispiel damit an, wie der Dienst benutzt werden soll: Dazu ändern wir die Methode, die den Input modelliert ein wenig ab: Wenn der Input ungleich NULL ist, wird der Algorithmus gestartet. Ist der Input null, wird zunächst geschaut, ob ein Benachrichtigungs-Service im BeanContext verfügbar ist. Falls ja, wird dieser Service benutzt, um eine Meldung auszugeben. Natürlich braucht man noch eine Instanzvariable, um eine Referenz auf die Dienstimplementierung zu halten.

```
 private Notification notificationService;

 public void input(String in)
 {
  lastInput=in;
  if(lastInput!=null)
   countCharacters();
  else
  {
   if(notificationService!=null)
    notificationService.notifyInfo(
     this.getClass().getSimpleName(), "NULL input empfangen!");
  }
 }
```

# BeanContext-Mitgliedschaft

## Modul wird zu Context hinzugefügt

Dieses Ereignis tritt im dWb immer dann ein, wenn ein Modul in einen Workspace eingefügt wird. Der Context, zu dem das Modul hinzugefügt wird, wird als Parameter übergeben. Hier kann man also Arbeiten ausführen, die als Reaktion auf die Verbindung zum BeanContext ausgeführt werden müssen.

## Modul wird aus Context entfernt

Dieses Ereignis tritt im dWb immer dann ein, wenn ein Modul aus einem Workspace gelöscht wird. In diesem Fall ist der Parameter der Methode NULL; Hier kann man also Arbeiten ausführen, die als Reaktion auf das Ende der Lebenszeit eines Moduls durchgeführt werden müssen (beispielsweise Freigeben von Ressourcen).

Auch in unserem einfachen Beispiel ist es angeraten, die Instanz des von unserem Modul genutzten Services wieder zurückzugeben - nur so ist es nämlich möglich, daß der Provider Ressourcen wieder freigibt, die mit der benutzten Instanz zusammenhingen. Anschließend rufen wir hier einfach die Implementation der Basisklasse auf.

```
@Override
public void setBeanContext(BeanContext bc) throws PropertyVetoException
{
 if(notificationService!=null)
 {
  if(getBeanContext()!=null)
  {
   if(BeanContextServices.class.isAssignableFrom(
    getBeanContext().getClass()))
   {
    BeanContextServices bcs=(BeanContextServices)getBeanContext();
    bcs.releaseService(this, this, notificationService);
   }
  }
 }
 super.setBeanContext(bc);
}
```

# Dienst wird zur Verfügung gestellt

Hier wird getestet, ob der neue Dienst der Kategorie angehört, die wir benötigen. Ist das der Fall, wird eine Referenz der Implementierung in der dafür vorgesehenen Instanzvariable gespeichert.

Die Argumente der Methode `getService` hängen stark vom Service und dessen Implementation ab.

```
@Override
public void serviceAvailable(BeanContextServiceAvailableEvent bcsae)
{
 super.serviceAvailable(bcsae);
```

```
 if(notificationService==null)
 {
  if(bcsae.getServiceClass()==
   de.netsysit.util.beans.context.service.Notification.class)
  try
  {
   notificationService =
    (Notification)(bcsae.getSourceAsBeanContextServices()).getService(
    this, this,Notification.class, null, this);
  }
  catch(Exception e)
  {
   notificationService=null;
  }
 }
}
```

## Anmerkung

Zu beachten ist, dass `getService` für einen bestimmten Typ nur einmal durch eine JavaBean aufgerufen werden dürfen! Passiert dies für denselben Typ eines Service mehrfach, ohne dass zwischendurch die Referenz auf die erhaltene Implementierung mittels `releaseService` zurückgegeben wird, wird eine Exception vom Typ `jaav.util.TooManyListenersException` geworfen, da der `BeanContextServiceRevokedEvent` (siehe unten) ein Unicast-Event ist und nicht wie zum Beispiel ein `PropertyChangeEvent` ein Multicast-Event.

Man sollte also immer zunächst prüfen, ob eine Oberklasse die betreffende Serviceimplementierung gegebenenfalls bereits abgeholt hat, bevor man `getService` in der eigenen Klasse für einen bestimmten Servicetyp aufruft. Das führt dazu, dass es unbedingt notwendig ist, referenzen auf mittels `getService` erhaltene Implementierungen für abgeleitete Klassen verfügbar zu machen - ansonsten verhindert man durch Aufruf von `getService` effektiv die Nutzung bestimmter Services in abgeleiteten Klassen: Diese können keine neue Referenz über `getService`erhalten, aber auch nicht auf die der Oberklasse zugreifen.

Aus diesem Grund sind hier alle Basisklassen aufgelistet, die getService benutzen - zusammen mit den Services, die dadurch für abgeleitete Klassen zur Verfügung stehen:

**`de.elbosso.dataflowframework.modules.RemoteModule.`**

- `de.elbosso.util.beans.context.service.Registry`
- `de.netsysit.dataflowframework.logic.services.WorkspaceAPI`
- `de.elbosso.dataflowframework.logic.services.ContextManagerService`

**`de.netsysit.dataflowframework.modules.BeanContextChildModuleBase`**

.

- `de.netsysit.dataflowframework.logic.services.WorkspaceAPI`
- `de.elbosso.dataflowframework.logic.services.ContextManagerService`

**`de.netsysit.dataflowframework.modules.ThreadingBeanContextChildModuleBase`**

.

- `de.netsysit.dataflowframework.logic.services.WorkspaceAPI`
- `de.elbosso.dataflowframework.logic.services.ContextManagerService`

**`de.netsysit.dataflowframework.modules.Persistor.`**

- `de.netsysit.dataflowframework.logic.services.persistence.ReportingEngine`
- `de.netsysit.dataflowframework.logic.services.DialogParentFrame`

# Dienst wird zurückgezogen

Hier wird getestet, ob der zurückgezogene Dienst der Kategorie angehört, die wir benötigen. Ist das der Fall, heißt das, dass dieser Dienst und damit die Implementierung nicht mehr zur Verfügung stehen - daher wird die Referenz auf die Implementierung wieder verworfen.

```
@Override
public void serviceRevoked(BeanContextServiceRevokedEvent bcsre)
{
 super.serviceRevoked(bcsre);
 if(bcsre.getServiceClass()==Notification.class)
 {
  notificationService=null;
 }
}
```

# Kapitel 8. BeanContextServices bereitstellen

## Warum?

Wie bereits im vorangegangenen Abschnitt beschrieben, können JavaBeans Dienste in Anspruch nehmen, die der Container anbietet, dem die jeweilige Instanz zugeordnet wurde. In dWb+ ist dieser Container der jeweilige Workspace - es besteht immer die Beziehung "diese Modul gehört zu jenem Workspace".

Im vorangegangenen Abschnitt wurden Möglichkeiten vorgestellt, solche Dienste zu benutzen. Im vorliegenden Abschnitt soll es nun darum gehen, solche Dienste für interessierte Module anzubieten.

Dazu muß ein entsprechendes Modul darauf vorbereitet sein, Informationen darüber zu verarbeiten, daß es zu einem BeanContext hinzugefügt oder daraus entfernt wurde. Ist dieser Context einer, der Dienste anbieten kann, publiziert das Modul anschließend die von ihm zur Verfügung gestellten Services.

## Ein einfaches Beispiel

Dieses Kapitel wird die Möglichkeiten und das Vorgehen zur Bereitstellung von BeanContextServices demonstrieren. Es wird ein Modul vorgestellt, das einen Dienst anbietet, einen String in einen int-Wert zu transformieren.

Den vollständigen Code für dieses Modul kann man im Anhang A, *Quellcode* in „Bereitstellung eines Dienstes für einen BeanContext“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

## Selber machen oder Basisklasse benutzen?

Wir werden die für dieses Beispiel einfachste Variante wählen und unser neues Modul von BeanContextChildModuleBase im Paket de.netsysit.dataflowframework.modules ableiten. Das führt dazu, dass einige Methoden überschrieben werden müssen, um unsere Anforderungen umzusetzen. Dieses Kapitel baut auf dem einfachen Beispiel ohne Threading auf - es werden allerdings nur die Erweiterungen für das Arbeiten mit dem BeanContext präsentiert - die komplette Klasse ist im Anhang zu finden.

```
import de.netsysit.dataflowframework.modules.BeanContextChildModuleBase;
import java.beans.PropertyVetoException;
import java.beans.beancontext.BeanContext;
import java.beans.beancontext.BeanContextServiceAvailableEvent;
import java.beans.beancontext.BeanContextServiceProvider;
import java.beans.beancontext.BeanContextServiceRevokedEvent;
import java.beans.beancontext.BeanContextServices;
import java.util.Iterator;

public class ServiceProvider extends
 BeanContextChildModuleBase implements
 BeanContextServiceProvider
{
```

```
}
```

# Dienst definieren

Wir fangen das Beispiel damit an, den Dienst zu definieren, der erbracht werden soll. Das ist schnell getan - wir definieren einfach ein öffentliches Interface, das den Dienst beschreibt. Es enthält lediglich eine Methode, die einen String als Parameter erwartet und einen int-Wert als Resultat liefert:

```
public interface Service
{
 int calculate(String in);
}
```

# BeanContext-Mitgliedschaft

## Modul wird zu Context hinzugefügt

Dieses Ereignis tritt im dWb immer dann ein, wenn ein Modul in einen Workspace eingefügt wird. Der Context, zu dem das Modul hinzugefügt wird, wird als Parameter übergeben. Hier kann man also Arbeiten ausführen, die als Reaktion auf die Verbindung zum BeanContext ausgeführt werden müssen. Wollen wir Services bereitstellen, müssen wir prüfen, ob der neue BeanContext die Bereitstellung von Services unterstützt und in diesem Fall unseren Service hinzufügen.

## Modul wird aus Context entfernt

Dieses Ereignis tritt im dWb immer dann ein, wenn ein Modul aus einem Workspace gelöscht wird. In diesem Fall ist der Parameter der Methode NULL; Hier kann man also Arbeiten ausführen, die als Reaktion auf das Ende der Lebenszeit eines Moduls durchgeführt werden müssen. In unserem Fall müssen wir den von unserem Modul bereitgestellten Dienst zurückziehen - natürlich nur dann, wenn der BeanContext, zu dem unser Modul gehörte, das Management von Services unterstützt

BeanContext-Instanzen unterstützen das Management von Services, wenn sie BeanContextServices implementieren.

```
@Override
public void setBeanContext(BeanContext bc) throws PropertyVetoException
{
 BeanContext former=getBeanContext();
 super.setBeanContext(bc);
 if(bc==null)
 {
  if(former!=null)
  {
   if(BeanContextServices.class.isAssignableFrom(former.getClass()))
   {
    ((BeanContextServices)former).revokeService(
     Service.class, this, true);
   }
  }
```

```
 }
 if(getBeanContext()!=null)
 {
  if(BeanContextServices.class.isAssignableFrom(
   getBeanContext().getClass()))
  {
   ((BeanContextServices)getBeanContext()).addService(
    Service.class, this);
  }
 }
}
```

# Dienst wird zur Verfügung gestellt

Da unser Beispielmodul selbst keine Dienste in Anspruch nehmen soll, beschränkt sich der Inhalt dieser Methode auf den Aufruf der Basisklassen-Variante.

```
@Override
public void serviceAvailable(BeanContextServiceAvailableEvent bcsae)
{
 super.serviceAvailable(bcsae);
}
```

Soll unser Dienst nicht nur bei Instantiierung und Löschen in einen / aus einem Workspace als Service de-/registriert werden, müssen wir uns hier eine Referenz auf einen Dienst vom Typ `java.beans.beancontext.BeanContextServicesSupport` merken, mit dem man zu beliebigen Zeitpunkten Services de-/registrieren kann. Dies kann zum Beispiel dann nützlich werden, wenn zur Erbringung des Dienstes Ressourcen benötigt werden, die direkt nach der Instantiierung noch nicht zur Verfügung stehen (Internet- oder Datenbank-Verbindungen beispielsweise).

# Dienst wird zurückgezogen

Normalerweise würde auch hier der Aufruf der Basisklassen-Methode ausreichen. Allerdings existiert in der Implementierung der BeanContext-Serviceverwaltung eine Besonderheit, der Rechnung getragen werden soll: Es ist nicht möglich, zwei ServiceProvider für den gleichen Service zu registrieren. Man stelle sich nun folgende Situation vor: Der Anwender legt ein Modul auf den Workspace, das der Service Service konsumieren möchte. Er legt weiterhin ein Modul auf den Arbeitstisch, das die Implementierung A bereitstellt. Er ist aus irgendeinem Grund nicht mit dessen Arbeitsweise zufrieden und möchte es austauschen. Legt er nun ein weiteres Modul auf den Workspace, das den Service mittels Implementierung B umsetzt, schlägt die registrierung des Dienstes fehl, da schon eine Implementierung registriert ist. Wird jetzt das Modul, das die Implementierung A anbietet, entfernt, existiert keine Service-Implementierung mehr im Context - das Modul, das auf eine Implementierung angewiesen ist, hört auf zu arbeiten.

Daher wird in unserem ServiceProvider geprüft, ob der zurückgezogene Service dem von unserem Modul angebotenen Service entspricht. In diesem Fall wird versucht, die von unserem Modul angebotene Implementierung nunmehr zu registrieren.

```
@Override
public void serviceRevoked(BeanContextServiceRevokedEvent bcsre)
```

```
{
 super.serviceRevoked(bcsre);
 if(getBeanContext()!=null)
 {
  if(BeanContextServices.class.isAssignableFrom(
   getBeanContext().getClass()))
  {
   BeanContextServices bcs=(BeanContextServices)getBeanContext();
   if(bcsre.isServiceClass(Service.class))
   {
    try
    {
     bcs.addService(Service.class, this);
    }
    catch (Exception e)
    {
    }
   }
  }
 }
}
```

# Dienst bereitstellen

Fragt nun ein Modul oder eine andere JavaBean nach einer Implementierung des Dienstes, instantiieren wir die Klasse, die die Implementierung bereitstellt und geben diese Instanz zurück:

```
public Object getService(BeanContextServices bcs, Object requestor,
 Class serviceClass, Object serviceSelector)
{
 return new Impl();
}
```

# Ressourcen freigeben

JavaBeans oder Module, die den Dienst unseres Providers in Anspruch genommen haben, benachrichtigen uns, wenn die den Dienst nicht mehr benötigen - in diesem Fall kann man mit der nicht mehr benötigten Instanz verbundene Ressourcen freigeben:

```
public void releaseService(BeanContextServices bcs, Object requestor,
 Object service)
{
}
```

# ServiceSelectors definieren

Module können nach ServiceSelectoren fragen, mit deren Hilfe die durch den Provider bereitgestellten Implementierungen angepaßt werden können - im vorliegenden Beispiel existiert eine solche Möglichkeit der Anpassung nicht, daher wird ein leerer Iterator zurückgegeben:

```
public Iterator getCurrentServiceSelectors(BeanContextServices bcs,
 Class serviceClass)
{
 return java.util.Collections.emptyIterator();
}
```

# Kapitel 9. Generics

## Warum

Die Inputs und Outputs werden in dWb+ zum Beispiel per Drag'n'Drop verknüpft. Dabei prüft die Anwendung aber, ob die Typen des betroffenen Inputs und des avisierten Outputs zueinander passen. Diese Prüfung ist nicht auf Typgleichheit aufgebaut, sondern bezieht Basisklassen und Interfaces mit ein.

Das bedeutet zum Beispiel, dass ein Input vom Typ Object von jedem beliebigen Output kontaktiert werden kann - Object ist schließlich mittelbar oder unmittelbar immer eine Oberklasse jedes Typs. Die Fundamentaldatentypen werden für diese Tests automatisch auf die korrespondierenden Klassentypen abgebildet, so dass das hier Gesagte auch für sie gilt. Ein weiteres Beispiel wäre ein Input vom Typ java.lang.Number. Mit diesem Input kann man alle Outputs verbinden, die einen Typ aufweisen, der dieses Interface implementiert. Auch hier gilt, dass die Fundamentaltypen für diesen Test automatisch in ihre korrespondierenden Klassentypen gewandelt werden - also funktioniert die Verbindung in diesem Fall auch von einem Output vom Typ int oder double aus.

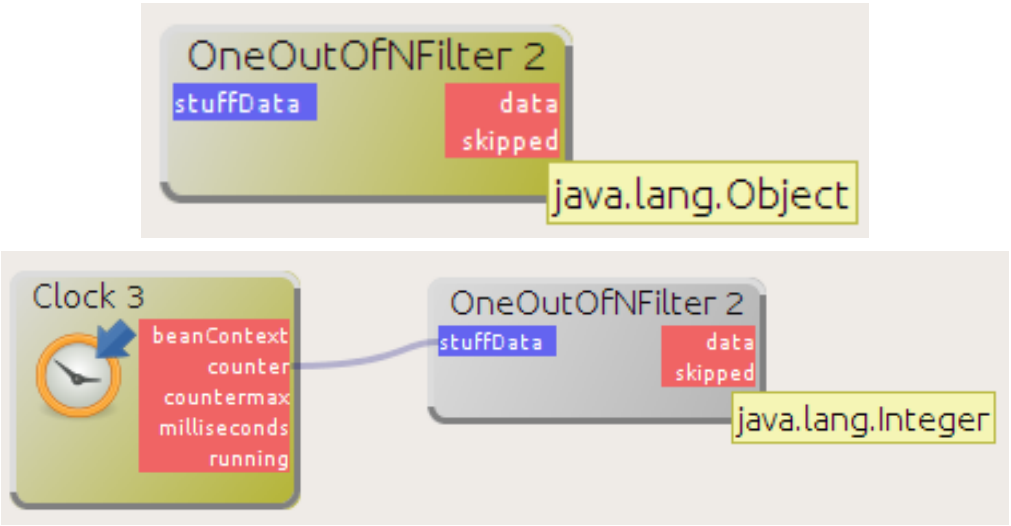
Darin liegt aber ein Problem, das - würde es nicht gelöst - für Probleme sorgen und die Benutzbarkeit stark herabsetzen könnte: Es existieren Algorithmen, die sich auf beliebige Datentypen anwenden lassen. Das scheint zunächst ein Fall für einen Input vom Typ Object zu sein. Nehmen wir als Beispiel ein Modul, das nur jedes zweite Datum, das am Input ankommt, am Output weiterleiten soll. Eine Funktionalität, die für jeden Typ nützlich und einsetzbar ist. Nehmen wir nun an, dass ein Output vom Typ double durch dieses Modul geleitet wird: Nachdem das Modul durchlaufen wurde, ist der Output aber nicht mehr vom Typ double, sondern vom Typ Object. Damit ist er in nachfolgenden Modulen, die eine Zahl als Input verlangen nicht mehr benutzbar.

Generics sind ein Mechanismus in dWb+, dieses Problem zu lösen. Kurz gesagt erlaubt es dieser Mechanismus, Module zu schaffen, deren Outputs im Typ abhängig vom Typ eines Inputs sind, wobei dieser erst dann festgelegt wird, wenn der betreffende Input mit einem Output verbunden wird.

### Gruppen

Gruppen - vergleiche „Gruppe“ in Kapitel 5, *Meta-Module* - verfügen über spezielle Metamodule, die dazu dienen, Daten in die Gruppe zu übernehmen oder aus der Gruppe auszuleiten. Auch diese haben zunächst den Typ `java.lang.Object` solange keine Verbindung zu einem anderen Modul besteht. Daraus ergibt sich, dass Ein- und Ausgänge generischer Module erst dann mit Input- oder Output-Modulen verbunden werden sollten, wenn der Typ für den zu verbindenden Slot am generischen Modul spezifiziert wurde. Geschieht das nicht, ändert sich beim Anschließen an das generische Modul zwar der Typ des verbundenen Slots, nicht aber der Typ des damit verbundenen Input- oder Output-Modules.

**Abbildung 9.1. Generisches Modul vor und nach Verbindung des Inputs**




In unserem Beispiel würde das Modul auf den Workspace gezogen und zunächst als Typ des Input und des Output java.lang.Object haben. Nachdem ein anderes Modul auf den Workspace gezogen wurde, das einen Output vom Typ int hat, und dieser mit dem Input unseres Beispielmoduls verbunden wurde, ändern sich der Typ des Inputs und des Outputs für das Beispielmodul automatisch auf int. Abbildung 9.1, „Generisches Modul vor und nach Verbindung des Inputs“ zeigt dies am Beispiel eines Filtermoduls. Diese Zuordnung ist danach nicht mehr änderbar - möchte man zum Beispiel doch Werte vom Typ double durch das Beispielmodul bearbeiten lassen, reicht es nicht aus, die Verbindung am Input zu lösen - der Typ wird dadurch nicht wieder zurück auf Object gestellt: Diese Zuordnung funktioniert für eine Modulinstanz nur einmal. Man kann natürlich mehrere Modulinstanzen auf den Workspace ziehen und jede dieser Instanzen mit einem Input eines anderen Typs verbinden - dadurch erhält man dann mehrere Module, die den gleichen Algorithmus aber für andere Typen abbilden.

# Ein einfaches Beispiel

Wir wollen in diesem Kapitel zeigen, wie man ein Modul implementieren muss, das es erlaubt, den Typ eines Inputs und eines Outputs erst zum Zeitpunkt des Schaltens einer Verbindung festzulegen. Diese Funktionalität ist für beliebige Module implementierbar, da sie auf der Implementierung eines Interfaces beruht. Das kann in Modulen, die von den dWb+ -Basisklassen abgeleitet wurden ebenso geschehen, wie in Modulen, bei denen andere Basisklassen gewählt wurden. Die Implementierung ist dabei schnell erledigt - es handelt sich dabei um ein Markerinterface ohne Methodendeklarationen. Der zweite Schritt verlangt zwingend eine BeanInfo-Klasse für dieses Modul.

Den vollständigen Code für dieses Modul kann man im Anhang A, *Quellcode* in „Generics“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

# Selber machen oder Basisklasse benutzen?

Die Klasse selbst wird wieder wie im einfachen Beispiel von ModuleBase abgeleitet, implementiert aber zusätzlich das Interface Generic.

```
import de.netsysit.dataflowframework.logic.Generic;
import de.netsysit.dataflowframework.modules.ModuleBase;
```

```
public class Generics extends ModuleBase implements
  Generic
{
}
```

# Input

Der Input wird mit dem Typ Object definiert. Die entsprechende Methode erhöht jedesmal wenn Daten empfangen werden einen Zähler um eins und ruft den Algorithmus des Moduls auf.

```
 private int counter;

 private Object lastInput;

 public void input(Object in)
 {
  lastInput=in;
  ++counter;
  process(in);
 }
```

# Output

Der Output wird ebenfalls mit Typ Object definiert.

```
 private Object processed;

 public Object getProcessed()
 {
  return processed;
 }
```

# Algorithmus

Der Algorithmus prüft, ob der Zähler durch 2 teilbar ist und sendet in diesem Fall das zuletzt erhaltene Datum als Output weiter.

```
 private void process(Object in)
 {
  if(counter%2==0)
  {
   Object old=getProcessed();
   send("processed", old, getProcessed());
  }
 }
```

# BeanInfo

Die BeanInfo muss nun noch erweitert werden - Sinn dieser Erweiterung ist die Verbindung von Inputs mit Outputs, damit Typänderungen an Inputs entsprechend auch für Typänderungen der zugeordneten Outputs

übernommen werden - ohne die Zuordnung per BeanInfo würde nur der Typ des Inputs sich bei der ersten Verbindung des Inputs mit einem Output ändern.

Diese Zuordnung geschieht über das Setzen eines Wertes namens GenericPortDescription an der Methode, die zu dem betreffenden Input gehört. Der Wert muss dabei eine Instanz der Klasse GenericPortDescription sein, die mittels eines String-Arrays konstruiert wird, das die abstrakten Namen der Properties enthält, deren Typ an den Typ des betreffenden Inputs gebunden werden sollen.

```
String[] boundProperties=new String[]{"processed"};
GenericPortDescription gpd=new
 GenericPortDescription(boundProperties);
someMethod.setValue("GenericPortDescription", gpd);
```

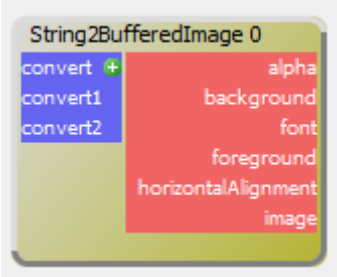
# Kapitel 10. Variable Anzahl von Inputs

## Warum

Bisher war es so, dass der Entwickler eines Moduls bereits bei der Erstellung wissen musste, welche und vor allem wie viele Inputs ein Modul haben soll, um einer gestellten Aufgabe gerecht zu werden. dWb + bietet jedoch auch für solche Fälle Unterstützung, in denen ein Algorithmus für beliebig viele Inputs funktioniere würde.

Es existiert nämlich die Möglichkeit, Inputs zu definieren, die sich vermehren lassen, wenn die Instanz auf den Workspace gezogen wurde. Solche Inputs weisen ein kleines grünes Pluszeichen neben ihrer Beschriftung auf - wenn der Anwender darauf klickt, entsteht ein weiterer Input am Modul mit dem selbem Typ. Der Name unterscheidet sich lediglich durch eine hinzugefügte Nummer. Abbildung 10.1, „Beispiel für ein Modul mit variabler Anzahl von Eingängen“ zeigt ein Beispiel für ein solches Modul, bei dem der Eingang bereits zweimal vervielfältigt wurde.

**Abbildung 10.1. Beispiel für ein Modul mit variabler Anzahl von Eingängen**




Ein Beispiel dafür wäre etwa eine Klasse, die eine Summe aus allen Inputs bilden soll. Dabei ist es egal, ob zwei oder fünf oder noch mehr Zahlen addiert werden sollen.

## Ein einfaches Beispiel

Wir wollen in diesem Kapitel zeigen, wie man ein Modul implementieren muss, das es erlaubt, einen Input zu vervielfältigen. Diese Funktionalität ist für beliebige Module implementierbar, da sie auf der Art und Weise, wie die dem Input zugeordnete Methode implementiert wird, beruht. Das kann in Modulen, die von den dWb+ -Basisklassen abgeleitet wurden ebenso geschehen, wie in Modulen, bei denen andere Basisklassen gewählt wurden. Der zweite Schritt verlangt zwingend eine BeanInfo-Klasse für dieses Modul. Den vollständigen Code für dieses Modul kann man im Anhang A, *Quellcode* in „Variable Anzahl von Inputs“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

## Selber machen oder Basisklasse benutzen?

Die Klasse selbst wird wieder wie im einfachen Beispiel von ModuleBase abgeleitet. Es wird ein Array angelegt, das in der Lage ist, eine variable Anzahl von Zahlen zu speichern. Im Konstruktor wird das Array zunächst als Array der Länge 0 initialisiert.

```
import de.netsysit.dataflowframework.modules.ModuleBase;
```

```
public class Adder extends ModuleBase
{
 Number[] numbers;

 public Adder()
 {
  super();
  numbers=new Number[0];
 }
}
```

# Input

Der Input wird mit dem Typ Number definiert. Weiterhin hat diese Methode einen zweiten Parameter, der den abstrakten Namen des Slots beinhaltet, der das aktuell zu verarbeitende Datum empfangen hat. Innerhalb der Methode wird zunächst die Nummer des Inputs festgestellt, der das zu verarbeitende Datum empfangen hat. Anschließend wird das Array bei Bedarf vergrößert und schließlich das empfangene Datum in das Array gespeichert. Zum Schluss wird der Algorithmus durch Aufruf der Methode process gestartet.

```
  public void input (Number in, String spec)
 {
  java.lang.String remainder=spec.substring("input".length());
  int i=remainder.length();
  if(i>0)
   i=java.lang.Integer.parseInt(remainder);
  while(i>=numbers.length)
  {
   Number[] nt=new Number[i+1];
   System.arraycopy(numbers, 0, nt, 0, numbers.length);
   numbers=nt;
  }
  numbers[i]=in;
  process();
 }
```

# Output

Der Output wird mit Typ double definiert.

```
 private double processed;

 public double getProcessed()
 {
  return processed;
 }
```

# Algorithmus

Der Algorithmus summiert alle Elemente im Array auf, sofern sie nicht NULL sind.

```
 private void process()
```

```
{
 double old=getProcessed();
 double t=0.0;
 for (Number number : numbers)
 {
  if(number!=null)
  {
   t+=number.doubleValue();
  }
 }
 processed=t;
 send("processed", old, getProcessed());
}
```

# BeanInfo

Die BeanInfo muss nun noch erweitert werden - Sinn dieser Erweiterung ist die Markierung eines Inputs, der zur Laufzeit vervielfältigbar sein soll - ohne die Markierung per BeanInfo würde die Methode wegen der zwei Parameter nicht als Input erkannt.

Diese Zuordnung geschieht über das Setzen eines Wertes namens VariablePortCount an der Methode, die zu dem betreffenden Input gehört. Der Wert muss dabei Boolean.TRUE sein.

```
someMethod.setValue("VariablePortCount", Boolean.TRUE);
```

Falls - wie zum Beispiel im Falle eines Moduls, das die logische Und-Funktion abbilden soll - für einen Input minimal mehr als ein Exemplar gebraucht wird, kann man das ebenfalls über die BeanInfo realisieren: Fügt man das unten angegebene Fragment in die BeanInfo ein, wird der Slot für die betreffende Methode beim Instantiieren bereits statt 1 mal 3 mal angelegt:

```
someMethod.setValue("VariablePortCountMin", Integer.valueOf(3));
```

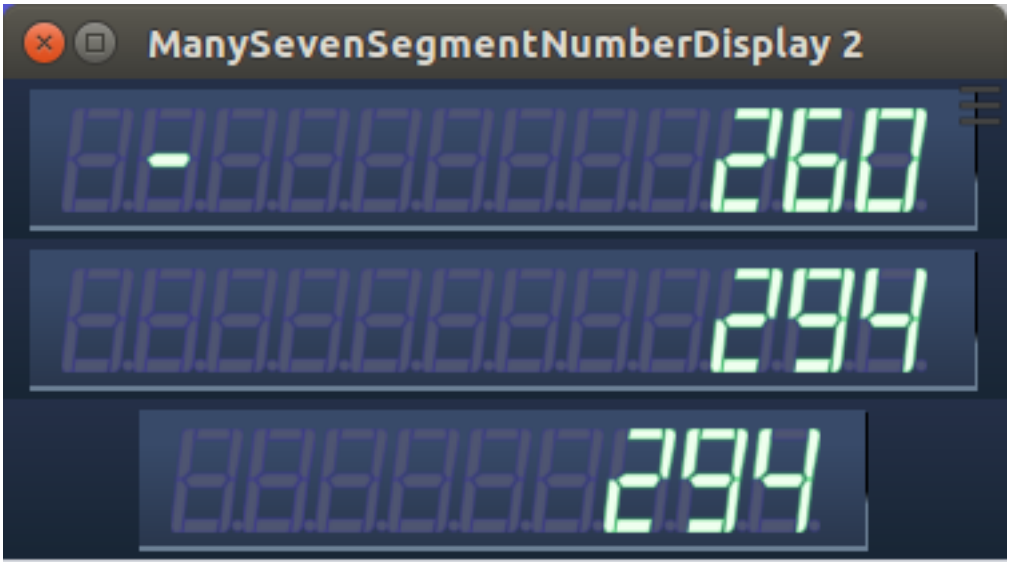
# Kapitel 11. Module zur Visualisierung mit einer variablen Anzahl von Inputs

## Warum

Manchmal ist es so, dass ein Modul geschaffen wird, das eine definierte Anzahl Eingänge - oft nur einen - aufweist. Hin und wieder jedoch soll ein Modul zur Visualisierung verschiedene gleichartige Daten visualisieren.

Würde man dann für die Visualisierung von 5 Ausgängen 5 Module instantiieren müssen, wäre die Arbeitsfläche schnell voll. Daher kann man sich Module vorstellen, die die Visualisierungskomponente einfach vervielfältigen können: Diese basieren auf Modulen mit einer variablen Anzahl von Eingängen und für jeden Eingang wird gleichzeitig auch eine entsprechende Visualisierungskomponente zum Parameterdialog hinzugefügt. Abbildung 11.1, „Beispiel für den Parameterdialog eines Modules zur Visualisierung mit variabler Anzahl von Eingängen“ zeigt ein Beispiel für das Aussehen des Parameterdialogs eines solchen Moduls, bei dem der Eingang bereits zweimal vervielfältigt wurde.

**Abbildung 11.1. Beispiel für den Parameterdialog eines Modules zur Visualisierung mit variabler Anzahl von Eingängen**




Ein Beispiel dafür wäre etwa eine Klasse, die beliebig viele numerische Werte visualisieren soll.

## Ein einfaches Beispiel

Wir wollen in diesem Kapitel zeigen, wie man ein Modul implementieren muss, das es erlaubt, beliebig viele numerische Werte in jeweils einer eigenen Komponente zu visualisieren. Dafür bringt dWb+ bereits eine Basisklasse mit, deren Einsatz dieses Beispiel veranschaulichen soll. Den vollständigen Code für dieses Modul kann man im Anhang A, *Quellcode* in „Variable Module zur Visualisierung“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

# Selber machen oder Basisklasse benutzen?

Die Klasse selbst wird unter Angabe zweier Templateparameter von der Basisklasse abgeleitet. Der erste beschreibt eine Klasse, die von komplizierten Komponenten in Model-View-Controller-Architektur als Modell benutzt werden soll. Der zweite Parameter beschreibt den Typ der variablen Input-Slots. Da wir in diesem Beispiel keine komplexe MVC-Komponente einsetzen wollen, genügt hier die Wiederholung des Input-Typs als Typ für das Model.

```
import de.elbosso.dataflowframework.modules.VariableNumberofInputsVisualization;
import java.awt.Component;
import javax.swing.JLabel;

public class MultiLabelVis extends
  VariableNumberofInputsVisualization<Number,Number>
{

 public MultiLabelVis()
 {
  super();
 }
}
```

# Input

Der Input wird mit dem Typ Number definiert. Weiterhin hat diese Methode einen zweiten Parameter, der den abstrakten Namen des Slots beinhaltet, der das aktuell zu verarbeitende Datum empfangen hat. Innerhalb der Methode wird die korrespondierende Methode der Oberklasse aufgerufen, der als dritter Parameter der Methodenname dieser Methode mitgegeben werden muss (die Oberklassenmethode muss wissen, wo sie im zweiten Parameter die Nummer findet).

```
   public void addNumber(Number newNumber,String spec)
 {
  super.addInput(newNumber,spec,"addNumber");
 }
```

# Hinzufügen neuer Komponenten durch Hinzufügen neuer Input-Slots

Dazu existiert eine Methode, die in abgeleitzeten Klassen implementiert werden muss- die abgeleitete Klasse agiert sozusagen als Factory für die Basisklasse: Die Bassisklasse verwaltet GUI-Komponenten (Hinzufügen, Entfernen, Layout, Persistenz,...) muss aber die korrekte Instantiierung den abgeleiteten Klassen überlassen. Der übergebene Parameter ist für komplexere Komponenten das Model - unser einfaches Beispiel kann den Parameter getrost ignorieren.

```
protected Component addComponent(Number model)
 {
  JLabel label=new JLabel();
  return label;
 }
```

# Schicken neuer Daten an die korrekte Komponente

Auch dazu existiert wieder eine abstrakte Methode, die abgeleitete Klassen überschreiben müssen: Die Basisklasse kümmert sich um alle Aspekte der Verwaltung, kann aber nicht wissen, wie die Komponte von dem neuen zu visualisierenden Wert erfahren soll. Der letzte Parameter sagt an, ob die Komponente (der erste Parameter) tatsächlich diejenige für dieses Datum verantwortliche ist. In komplexen MVC-Komponenten kann es sein, dass man anderenfalls noch benachbarte Modelle aktualisieren möchte - wir ignorieren das in diesem einfachen Beispiel.

```
protected void handleAddition(Component comp,Number model,
        Number newdata, boolean isDataSlot)
 {
  if(isDataSlot)
  {
   if(newdata==null)
   {
    ((JLabel)comp).setText("");
   }
   else
   {
    ((JLabel)comp).setText(newdata.toString());
   }
  }
 }
```

# Model

Schließlich existiert noch eine zu überschreibende Methode in der Basisklasse: Für jeden neuen Eingang wird eine Instanz der in der Definition der abgeleiteten Klasse spezifizierten Modelklasse benötigt. Da die Basisklasse nicht wissen kann, wie eine solche Instanz zu erzeugen ist, kommt die Verantwortung dafür wieder der abgeleiteten Klasse zu.

```
 protected Number createInstanceToBeStoredAtBackingStore()
 {
  return new Double(0);
 }
```

# BeanInfo

Für die BeanInfo trifft das bereits weiter oben über alle Module mit variabler Eingangsanzahl Gesagte zu.

# Persistenz der Konfiguration für die Komponenten

Jede der einzelnen Komponenten kann Eigenschaften haben, die der Nutzer anpassen können soll. Das wird erreicht, indem die Komponenten auf Druck mit der mittleren Maustaste einen Dialog öffnen, der ähnlich

der automatisch generierten Parameterdialoge für Bean-Module ein generiertes Frontend zur Manipulation der JavaBean-Properties der jeweiligen Kopmponente enthält. Dort vorgenommene Änderungen werden beim Speichern von Workspaces oder beim Kopieren uind Einfügen eines solchen Modules erhalten. Der Entwickler muss dazu nichts mehr tun - alle beschriebenen Funktionalitäten werden komplett durch die Basisklasse zur Verfügung gestellt.

# Kapitel 12. Variable Module

## Warum

Bisher wurden Programmiertechniken vorgestellt, die alle eine Sache gemeinsam hatten: Der Inhalt und die Schnittstellen der Module waren nicht nur auf technischer Ebene bei der Implementierung des jeweiligen Moduls vollständig definiert, sondern das traf auch auf die Ebene der Fachlichen Domäne zu.

Nun könnte es aber auch so sein, dass zum Zeitpunkt der Implementierung eines Moduls zwar die technische Ebene klar ist, die fachliche Domäne jedoch während der Implementierung noch nicht vollständig dokumentiert ist. Ein Beispiel dafür wäre die Anbindung von Hardwaremodulen, bei denen die technische Ebene das Kommunikationsprotokoll wäre. Die Module selbst geben über definierte Botschaften erst zur Laufzeit an, welche Features auf ihnen implementiert sind - also etwa die Anzahl digitaler Ein- und Ausgänge, die dann als Slots im Modul widergespiegelt sein müssten.

Ein anderes Beispiel ist die Fernsteuerung oder -überwachung von IT-Systemen mittels der Schnittstelle Java Management Extensions (JMX). Auch hier ist das Protokoll - die technische Ebene - vor der Implementierung klar. Man müsste aber für jede Managed Bean ein eigenes Modul schreiben, um es innerhalb von dWb+ abbilden zu können. Schöner wäre es da, die weiter oben bereits skizzierte Technik verwenden zu können: Man schafft ein Modul, das ganz allgemein die technische Ebene unterstützt und die Managed Bean nach ihren Attributen fragen kann. Aus diesen Informationen wird dann zur Laufzeit das Modul mit den passenden Slots erzeugt.

Dabei ist zu beachten, dass die generellen Mechanismen zur Erzeugung von Slots bestehen bleiben: Jede Property, die lesbar ist, erzeugt einen Ausgang am Modul und jede public Methode mit einem Parameter erzeugt einen Eingang.

Als Beispiel wird hier eine Klasse vorgestellt, deren Ausgänge (Anzahl, Namen und Typen) durch eine Properties-Datei bestimmt wird.

## Ein einfaches Beispiel

Zur Umsetzung oben genannter Ideen muss das Modul einen Kontrakt erfüllen, der im Wesentlichen aus zwei Forderungen besteht:

- Implementierung des Interface VariableBean aus dem Namensraum de.netsysit.dataflowframework.ui.variable
- Versenden eines PropertyChangeEvents wann immer sich die Input- oder Output-Slots ändern. Dabei ist der Name mit insAndOuts und der Typ der Daten, die mit diesem Ereignis versendet werden ConnectionEndPointDescriptionCollections aus dem Namensraum de.netsysit.dataflowframework.logic festgelegt.

Den vollständigen Code für dieses Modul kann man im Anhang A, *Quellcode* in „Variable Module“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

## Selber machen oder Basisklasse benutzen?

Die Klasse selbst wird wieder wie im einfachen Beispiel von ModuleBase abgeleitet. Zusätzlich wird vereinbart, dass diese Klasse das Interface VariableBean implementiert.

```
import de.netsysit.dataflowframework.logic.ConnectionEndPointDescription;
import de.netsysit.dataflowframework.logic.ConnectionEndPointDescriptionCollection
import de.netsysit.dataflowframework.modules.ModuleBase;
import de.netsysit.dataflowframework.ui.variable.VariableBean;
import java.io.File;

public class VariableDemo extends ModuleBase implements VariableBean
{
}
```

# Input

Wir werden diesem Modul der Übersichtlichkeit halber keinen Input spendieren.

# Konfigurationsparameter

Als Typ des einzigen Konfigurationsparameters wird File aus dem Namensraum java.io festgelegt:

```
 private File definition;

 public File getDefinition()
 {
  return definition;
 }

 public void setDefinition(File definition)
 {
  this.definition = definition;
  reReadDocument();
 }
```

# Output

Das Modul verfügt durch die Festlegung des Konfigurationsparameters definition bereits über einen Ausgang. Für dieses Beispiel wird kein weiterer hinzugefügt.

# Versenden des PropertyChangeEvents "insAndOuts"

Das Setzen einer neuen Datei für das Property definition ruft die Methode reReadDocument auf. Die folgende Implementierung kürzt stark ab: wegen der Übersichtlichkeit wurde Code zur Fehlermanagement komplett entfernt.

```
 private ConnectionEndPointDescriptionCollections inAndOuts;

 private void reReadDocument()
 {
  ConnectionEndPointDescriptionCollections old=getInAndOuts();
```

```
 inAndOuts=null;
 java.util.Properties props=new java.util.Properties();
 java.io.FileInputStream fis=null;
 try
 {
  fis=new java.io.FileInputStream(definition);
  props.load(fis);
  java.util.List<ConnectionEndPointDescription> l=
   new java.util.LinkedList();
  for (java.lang.String key : props.stringPropertyNames())
  {
   ConnectionEndPointDescription cepd=
    new ConnectionEndPointDescription();
   cepd.setPortName(key);
   cepd.setTypename(props.getProperty(key));
   l.add(cepd);
  }
  inAndOuts=new ConnectionEndPointDescriptionCollections(null, l);
  send("inAndOuts",old,getInAndOuts());
  fis.close();
 }
 catch(java.io.IOException exp){}
}
```

# Interface VariableBean

Nunmehr erfolgt noch die Implementierung der Methoden des Interfaces VariableBean um den Kontrakt vollständig umzusetzen. Zuerst die Methode zum Zugriff auf die Definitionen der Ein- und Ausgänge.

```
public ConnectionEndPointDescriptionCollections getInAndOuts()
{
 return inAndOuts;
}
```

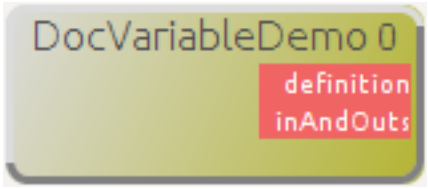
Die Methode genericOperation muss für unser Beispiel eigentlich nicht implementiert werden, da sie nur benötigt wird, wenn tatsächlich variable Eingänge im Modul möglich sind: Sie wird immer dann aufgerufen, wenn ein Signal an einem der variablen Eingänge eingeht und verarbeitet werden muss. Der erste Parameter ist dabei eine Referenz auf das empfangene Datum, der zweite der Name des Input-Slots, über den dieses Datum empfangen wurde. Diese Methode ähnelt damit sehr stark dem Mechanismus in Modulen mit einer variablen Anzahl von Eingängen wie er in Kapitel 10, *Variable Anzahl von Inputs* in „Input“ beschrieben wird. Im vorliegenden Beispiel wird diese Methode mit einem leeren Rumpf initialisiert:

```
public void genericOperation(Object input, String name)
{
}
```

# Ergebnis

Wird das Modul auf den Arbeitsbereich gezogen, stellt es sich ohne Konfigurationsänderung zunächst wie in Abbildung 12.1, „Variables Modul direkt nach Hinzufügen zum Arbeitsbereich“ dar:

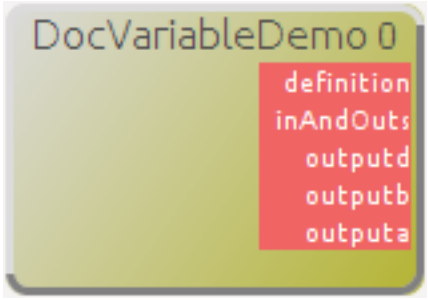
**Abbildung 12.1. Variables Modul direkt nach Hinzufügen zum Arbeitsbereich**




Speichert man die folgenden Zeilen in einer Datei und setzt man den Namen dieser Datei anschließend als Wert der Eigenschaft definition im Parameterdialog, ergibt sich die neue Gestalt des Modul mit drei weiteren Eingängen wie in Abbildung 12.2, „Variables Modul nach Hinzufügen dreier weiterer Ausgänge“ zu sehen:

```
outputa=java.lang.Integer
outputb=java.lang.String
outputd=de.netsysit.util.geo.Position
```

**Abbildung 12.2. Variables Modul nach Hinzufügen dreier weiterer Ausgänge**



# Kapitel 13. Variable Module für User Eingaben

## Warum

Manche Workspaces erfordern eine hohe Anzahl an Eingabemöglichkeiten für den Anwender. Man kann dazu Module schaffen, die jeweils eine Eingabe gestatten und also einen Output Slot aufweisen. Damit wäre der Anwender aber gezwungen, sehr viele Parameterfenster geöffnet zu halten. Gestaltet man solche Module für die Eingabe mehrerer Parameter, der Anwender benötigt aber nicht so viele, erhält man den Fall, dass die Parametermodule unnütz viel Platz beanspruchen.

Optimaler wäre es, Module zu haben, die als Voreinstellung genau einen Parameter konfigurieren können, aber wenn der Anwender es wünscht, weitere zur Verfügung stellen könnten.

Ein Beispiel Beispiel dafür ist der Fall, in dem in einem Workspace viele boolesche Werte durch den Anwender einstellbar sein müssen. Er benötigt also ein Modul, mit dem er beliebig viele boolesche Parameter konfigurieren kann.

Als Beispiel wird hier eine Klasse vorgestellt, bei dem die Anzahl Output Slots durch den Anwender bestimmt wird. Jeder Ausgang erhält im Parameterfenster eine entsprechende Komponente, mit der der zugehörige Wert beeinflusst werden kann.

## Ein einfaches Beispiel

Wir wollen in diesem Kapitel zeigen, wie man ein Modul implementieren muss, das es erlaubt, beliebig viele boolesche Werte mittels jeweils einer eigenen Komponente zu parametrieren Dafür bringt dWb+ bereits eine Basisklasse mit, deren Einsatz dieses Beispiel veranschaulichen soll. Den vollständigen Code für dieses Modul kann man im Anhang A, *Quellcode* in „Variable Module für User Eingaben“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

## Selber machen oder Basisklasse benutzen?

Die Klasse selbst wird unter Angabe eines Templateparameters von der Basisklasse abgeleitet. Dieser beschreibt die Klasse, die an den Output Slots zur Verfügung gestellt werden soll. Er muss im Aufruf des Basisklassenkonstruktors neben der Festlegung, ob die Komponenten horizontal oder Vertikal angeordnet werden sollen, übergeben werden.

```
import de.elbosso.dataflowframework.modules.VariableNumberofParameters;
import java.awt.event.ActionEvent;
import java.awt.Component;
import java.awt.event.ActionListener;
import javax.swing.JCheckBox;
import de.netsysit.dataflowframework.modules.ModuleBase;

public class VariableNumberOfBooleans extends VariableNumberofParameters<Boolean>
{
 public VariableNumberOfBooleans
 {
```

```
  super(Boolean.class,false);
 }

}
```

# Input

Wir werden diesem Modul der Übersichtlichkeit halber keinen Input spendieren.

# Konfigurationsparameter

Wir werden diesem Modul der Übersichtlichkeit halber keine Konfigurationsparameter spendieren

# Output

Das Modul erhält einen Output-Parameter namens output bereits durch seine Basisklasse.

# Erzeugen der Komponenten zur Konfiguration der Ausgänge

Jedesmal, wenn ein neuer Ausgang zum Modul hinzugefügt wird, muss auch eine Komponente zum Parameterfenster hinzugefügt werden, mit der dieser Ausgang konfiguriert werden kann. Dazu ruft die Oberklasse eine abstrakte Methode auf, die unsere abgeleitete Klasse implementieren muss. Dabei ist es so, dass dieser Fall auch auftritt, wenn das Modul aus einer XML-Repräsentation deserialisiert werden soll. Dann wird die bereits instantiierte Komponente durch die Basisklasse an unsere Klasse übergeben und wir müssen sie entsprechend verknüpfen. Dazu müssen zwei Methoden implementiert werden:

```
protected VariableNumberofParameters<Boolean>.Glue<Boolean>
 addComponent(String name)
{
 return new Glue(name, this);
}
protected VariableNumberofParameters<Boolean>.Glue<Boolean>
 addComponent(Component comp, String string)
{
 return new Glue(comp, string, this);
}
```

# Glue

Die konkrete Implementierung dieser Klasse sorgt dafür, dass Events in der Komponente jedes Slots die entsprechenden PropertyChangeEvents senden. Das meiste dazu wird bereits in der Basisklasse erledigt, der Programmierer muss hier lediglich noch zwei Methoden implementieren: Zum einen die Erzeugung der Komponente an sich und zum anderen das Event-Handling für Events aus der Komponente.

```
private class Glue extends VariableNumberofParameters<Boolean>.Glue<Boolean>
 implements ActionListener
```

```
{
 private JCheckBox comp;

 Glue(Component comp,String propertyName,ModuleBase module)
 {
  super(propertyName,module);
  this.comp=(JCheckBox)comp;
  this.comp.addActionListener(this);
 }
 Glue(String propertyName,ModuleBase module)
 {
  super(propertyName,module);
 }
 public Component getComponent()
 {
  if(comp==null)
  {
   comp=new JCheckBox();
   comp.addActionListener(this);
  }
  return comp;
 }
 public Boolean performDataChange()
 {
  return comp.isSelected();
 }

 public void actionPerformed(ActionEvent e)
 {
  dataChange();
 }
}
```

# Kapitel 14. Module zur Filterung

## Funktionsbeschreibung

Ich habe bereits ein Framework geschrieben, das es erlaubt, beliebige Regeln zur Validierung von Daten anzuwenden. Diese Regeln sind auch hierarchisch kombinierbar, oder mittels boolescher Algebra zu verknüpfen. Letztendlich ist jede Regel die Implementierung des einfachen Interface Rule:

```
import java.util.Collection;
import java.util.Map;

public interface Rule
{
 // einige Konstanten weggelassen...
 public Collection validate(Object value);
 public Collection validate(Object value, Map context);
}
```

Ich verfertigte eine Basisklasse für Module, die mittels solcher Regeln eingehende Daten in zwei Klassen aufteilen: solche, für die die Regel zutrifft und solche, für die sie nicht zutrifft.

Damit ist es einfach, neue Module für neue Regeln zu erstellen, wie dieses Kapitel aufzeigen soll. Den vollständigen Code für dieses Modul kann man im Anhang A, *Quellcode* in „Module zur Filterung“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

## Selber machen oder Basisklasse benutzen?

Die Basisklasse, die geschaffen wurde, heißt RuleBase und befindet sich im package de.elbosso.dataflowframework.modules.filter.rules. Beim Ableiten muss man lediglich zwei Templateparameter spezifizieren, die ansagen, auf welchen Datentyp der Filter angewendet werden kann und welcher Filter es sein soll:

```
import de.elbosso.dataflowframework.modules.filter.rules.RuleBase;
import de.netsysit.util.validator.rules.FloatingPointMinMaxRule;
import de.netsysit.util.beans.InterfaceFactory;
import de.netsysit.dataflowframework.modules.BeanContextChildModuleBase;
public class FloatingPointMinMaxFilter extends
  RuleBase<Number,FloatingPointMinMaxRule>
{
}
```

Weiterhin muss man die Interfacefactory anweisen, wie viele Schritte sie in der Vererbungshierarchie nach oben gehen soll, um das ParameterPanel zu konstruieren:

```
 static
```

```
{
 InterfaceFactory.setSuperclassAssociationForEventDispatchThread
   (FloatingPointMinMaxFilter.class, BeanContextChildModuleBase.class);
}
```

# Konstruktor

Im Konstruktor schließlich ist der Basisklassenkonstruktor aufzurufen, dem der Datentyp und eine Instanz der Filterregel übergeben wird:

```
public FloatingPointMinMaxFilter()
{
 super(Number.class,new FloatingPointMinMaxRule());
}
```

# Kapitel 15. Eigene Bedienoberflächen für ein Modul

## Anpassung der Bedienoberfläche für Module

Wie bereits weiter vorn beschrieben wird aus der Information, die in den Properties des Moduls steckt, automatisch eine Bedienoberfläche generiert, die bei Doppelklick auf den Titel des Moduls angezeigt wird. Diese Bedienoberfläche lässt sich über die BeanInfo des jeweiligen Moduls beeinflussen: Das hört nicht bei den Namen der jeweiligen Properties auf. So kann man zum Beispiel mittels der Eigenschaft hidden auch gezielt Properties aus dem Parameterpanel ausschließen.

Dabei ist zu beachten, dass auf diese Weise nur die Properties in die Produktion der Bedienoberfläche einbezogen werden, die genau in der betreffenden Klasse deklariert wurden. Properties der direkten und aller weiteren Oberklassen werden nicht berücksichtigt. Um das zu erreichen, ist ein Modul um folgenden Code zu erweitern:

```
import de.netsysit.util.beans.InterfaceFactory;
...
static
{
 InterfaceFactory.setSuperclassAssociationForEventDispatchThread(
  Module.class,Stop.class);
}
```

Diese Anweisung sorgt dafür, dass alle Properties der Modulklasse Module und die aller Oberklassen bis zur Klasse Stop in die Generierung der Bedienoberfläche einbezogen werden. Die Properties der Klasse Stop werden nicht mit einbezogen. Diese Klasse ist also die erste in der Hierarchie, die dann ignoriert wird.

Genügen die Anpassungsmöglichkeiten über die Beaninfo nicht, kann der Entwickler eines Moduls auch eine eigene bedienoberfläche definieren, die dann bei Doppelklick auf den Modultitel angezeigt wird. Dafür ist lediglich ein Interface zu implementieren.

Den vollständigen Code für dieses Modul kann man im Anhang A, *Quellcode* in „Modul“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

```
import de.netsysit.dataflowframework.modules.ModuleBase;
import de.netsysit.dataflowframework.ui.VisualComponentProvider;
import java.awt.Container;

public class VCPCharacterCounter extends ModuleBase implements
 VisualComponentProvider
{
}
```

# Vorarbeit

Wir gehen von dem einfachen Modul aus, das wir als erstes implementiert haben. Diesmal soll die Klasse aber über eine eigene Parametrierungsoberfläche verfügen. Dazu erstellen wir eine Klasse, die wir später dazu benutzen wollen.

Den vollständigen Code für diese Klasse kann man im Anhang A, *Quellcode* in „Angepasster JToggleButton“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import javax.swing.JToggleButton;

public class IgnoreSpaceToggle extends JToggleButton implements
  PropertyChangeListener,
  ActionListener
{
 private VCPCharacterCounter bean;

 public IgnoreSpaceToggle(VCPCharacterCounter bean)
 {
  super("Ignore Spaces");
  this.bean = bean;
  this.setSelected(bean.isIgnoreSpaces());
  bean.addPropertyChangeListener(this);
  addActionListener(this);
 }

 public void propertyChange(PropertyChangeEvent evt)
 {
  this.setSelected(bean.isIgnoreSpaces());
 }

 public void actionPerformed(ActionEvent e)
 {
  bean.setIgnoreSpaces(this.isSelected());
 }


}
```

Diese Klasse stellt einfach einen JToggleButton dar, der an eine Instanz unseres Moduls gebunden wird: immer wenn sich eine Property der Modulinstanz ändert, aktualisiert der Button seinen Status und wenn der Status des Buttons geändert wird, wird dieser Status an die Eigenschaft ignoreSpaces der Bean weitergeleitet.

# Implementierung

In der Modulklasse muss das Interface implementiert werden. Die dort festgelegte Methode wird genau einmal aufgerufen. Es ist nicht nötig, sich eine Referenz auf die zurückgegebene Instanz der Klasse Component zu merken - dWb+ benötigt diese Referenz nur als Rückgabewert für die Methode. Man kann sie sich natürlich merken, falls der Algorithmus zum Beispiel in der Lage sein soll, die visuellen Komponenten zu modifizieren.

In unserem Beispiel erstellen wir eine Instanz unserer neuen Button-Klasse, fügen ihn in ein JPanel ein und geben eine Referenz auf dieses JPanel zurück. Damit sind wir auch schon fertig - jetzt wird der Status der Konfigurationseigenschaft ignoreSpaces nicht mehr durch die generierte CheckBox sondern durch den ToggleButton bestimmt.

```
public Container getVisualComponent()
{
 IgnoreSpaceToggle ignoreSpaceToggle=new IgnoreSpaceToggle(this);
 JPanel p=new JPanel(new BorderLayout());
 p.add(ignoreSpaceToggle);
 return p;
}
```

# Kapitel 16. Actions zur Steuerung eines Moduls

## Warum

Wie bereits weiter vorn beschrieben wird aus der Information, die in den Properties des Moduls steckt, automatisch eine Bedienoberfläche generiert, die bei Doppelklick auf den Titel des Moduls angezeigt wird. Diese Bedienoberfläche lässt sich über die BeanInfo des jeweiligen Moduls beeinflussen: Das hört nicht bei den Namen der jeweiligen Properties auf. So kann man zum Beispiel mittels der Eigenschaft hidden auch gezielt Properties aus dem Parameterpanel ausschließen.

Man kann das generierte Parameterpanel auch vollständig durch eine eigene Implementation ersetzen, wie im vorausgehenden Kapitel gezeigt. Manchmal jedoch kann man eine Konfigurationseinstellung nicht so sehr als Änderung eines Parameterwertes ausdrücken, sondern eher als auszuführende Action. Wenn diese Action nicht durch Daten eines anderen Moduls sondern durch den Anwender ausgelöst werden soll, braucht man einen Mechanismus, der es erlaubt, dem Anwender Knöpfe, Werkzeugleisten oder Menüs mit diesen Actions anzubieten.

In dWb+ verfügt jedes Modul bereits über ein Kontextmenü, das über Drücken der rechten Maustaste über dem Modultitel erreichbar ist. In dieses Menü können sehr einfach eigene Actions eingeklinkt werden. Dazu ist es nötig, ein Interface zu implementieren.

```
import de.netsysit.dataflowframework.modules.ModuleBase;
import de.netsysit.dataflowframework.ui.ActionsProvider;
import javax.swing.Action;

public class ActionsCharacterCounter extends ModuleBase implements
  ActionsProvider
{
}
```

Dieses Interface deklariert eine Methode, deren Rückgabewert ein Array von Actions ist. Jede dieser Actions erscheint als Menüpunkt innerhalb eines Untermenüs im Kontextmenü. Dieses Kontextmenü trägt den konkreten Namen des Moduls.

Den vollständigen Code für dieses Modul kann man im Anhang A, *Quellcode* in „Actions für Module“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

# Implementierung

## Actions erzeugen

Diese Methode wird genau einmal aufgerufen. Es ist nicht nötig, sich Referenzen auf die Actions zu merken - dWb+ benötigt diese Referenzen nur als Rückgabewert für die Methode. Man kann sie sich natürlich

merken, falls der Algorithmus zum Beispiel in der Lage sein soll, die Eigenschaft enabled der einzelnen Actions zu modifizieren.

*Falls man von einer Basisklasse ableitet, die diese Methode bereits implementiert, muß man dafür sorgen, daß die Actions, die die Basisklasse bereitstellt, ebenfalls mit zurückgegeben werden - Also zuerst die Actions der Basisklasse mittels super.provideCustomActions() ermitteln und diese dann zusammen mit den eigenen zurückgeben!*

In unserem Beispiel erstellen wir eine anonyme inner Class, die das aktuelle Datum und Uhrzeit über einen Dialog anzeigt.

```
public Action[] provideCustomActions()
{
 return new Action[]{
  new AbstractAction("action") {

   public void actionPerformed(ActionEvent e)
   {
    JOptionPane.showMessageDialog(null, new Date().toString());
   }
  }
 };
}
```

# Vorbereitung Anzeige Menü

Diese Methode wird jedes Mal aufgerufen bevor das Menü mit den Actions angezeigt wird. Hier kann man zum Beispiel Code unterbringen, der überprüft, ob der Status der Eigenschaft enabled bestimmter Actions abhängig vom Zustand des Moduls geändert werden muss. In unserem Fall ist das nicht nötig - daher bleibt die Implementation der Methode leer.

```
public void preparePopupShow()
{

}
```

# Kapitel 17. Spezielle Properties zur besseren Integration von Modulen

## Warum

Es kann möglich sein, daß das Framework über Statusänderungen eines Moduls informiert werden soll.

Der Mechanismus solcher Benachrichtigungen beruht zur Zeit auf Properties.

In dWb+ ist aktuell eine solche Property definiert. Diese Property heißt dWb+::GuiState. Sie ist als symbolische Konstante in SpecialModuleWidgetProperties definiert:

```
import de.netsysit.dataflowframework.modules.ModuleBase;
import de.netsysit.dataflowframework.ui.SpecialModuleWidgetProperties;
import javax.swing.Action;

public class ActionsCharacterCounter extends ModuleBase implements
  SpecialModuleWidgetProperties
{
}
```

Diese Property zeigt mit einem Wert von false an, daß die Bedienoberfläche des Moduls für Änderungen gesperrt werden soll. Ein Wert von true signalisiert, daß die Bedienoberfläche wieder entsperrt werden soll.

Hat der Autor des Moduls eine eigene Bedienoberfläche erstellt (vergleiche Kapitel 15, *Eigene Bedienoberflächen für ein Modul*), muß er dafür keine besonderen Vorkehrunen treffen: Das Framework realisiert das Sperren und Entsperren automatisch über die Konfiguration der GlassPane des betreffenden Parameterdialogs.

# Kapitel 18. Module, die mit externen Ressourcen arbeiten müssen

## Motivation

Die komplexeste Aufgabe aus technischer Sicht gesehen, Module in die Anwendung zu integrieren, ist sicher die Arbeit mit Modulen, die begrenzt verfügbare externe Ressourcen benutzen müssen. Beispiele für solche Ressourcen sind alle Schnittstellen, die an den betreffenden Rechner angeschlossen sein können - seien es Netzwerk-Sockets, serielle Schnittstellen oder andere.

Eine der Herausforderungen ist dabei die Aufgabe, die Ressource verläßlich zu allokieren (zu öffnen). Das kann durch Anwenderinteraktion geschehen. Allerdings wäre es komfortabler, wenn ein Modul erkennen würde, dass es zu einem Workspace hinzugefügt wird und es daraufhin selbstständig seine Konfiguration überprüft und die Verbindung aufbaut. Ein weiterer wichtiger Punkt ist die Anforderung, dass das Modul immer - auch wenn der Anwender dies vergisst - die allokierten Ressourcen freigibt, wenn es vom Arbeitstisch entfernt wird.

Dieser Abschnitt beschreibt die Möglichkeiten, die dWb+ dem Programmierer zur Verfügung stellt, um die oben genannten Anforderungen zu bewältigen und den Administrationsaufwand weitgehend von den fachlichen Anforderungen zu trennen.

Die Basisklasse stellt ebenfalls zwei Aktionen für den Anwender bereit, mit denen der Verbindungsaufbau gesteuert werden kann:

**Tabelle 18.1. Actions für das Verbindungsmanagement**

| | |
|---|---|
| | Öffnet die Verbindung |
| | Schließt die Verbindung und gibt alle damit zusammenhängenden Ressourcen wieder frei |

## Funktionsbeschreibung

Dieses einfache Modul soll eine Verarbeitung realisieren, die eingehende Strings über einen Socket weiterversendet. Den vollständigen Code für dieses Modul kann man im Anhang A, *Quellcode* in „SocketOut“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

## Die Basisklasse

Die Basisklasse, die dWb+ zur Verfügung stellt, heißt CommunicationTemplate.

```
import java.io.IOException;
import java.net.Socket;
import java.io.PrintWriter;
import java.net.InetAddress;

public class SocketOut extends
```

```
 de.netsysit.dataflowframework.modules.CommunicationTemplate
{
}
```

# Input

Der Input wird über die Implementierung einer entsprechenden Methode definiert - in unserem Fall hat diese Methode die Signatur `public void sendData(java.lang.String data)`.

```
public void sendData(java.lang.String data)
{
 if(isConnectionEstablished())
 {
  pw.println(data);
 }
}
```

Die beiden Methode `boolean isConnectionEstablished()` wird von der Oberklasse bereitgestellt. Die Methode `void closeDown()` wird von der Oberklasse deklariert und muss in einer davon abgeleiteten Klasse definiert werden.

# Konfiguration

Wir benötigen zwei Konfigurationsparameter: Zum einen den Host, an den die Daten gesendet werden sollen, und zum anderen den Port als genaue Adresse für die Daten.

```
private String host;

public String getHost()
{
 return host;
}

public void setHost(String adr)
{
 String oldAdr = getHost();
 this.host = adr;
 send("host", oldAdr, getHost());
}
private int port;

public void setPort(int p)
{
 int oldPort = getPort();
 this.port = p;
 send("port", oldPort, getPort());
}

public int getPort()
```

```
{
 return port;
}
```

# Deklarierte Basisklassenmethoden definieren

Die Methode `void manageConnectionImpl()` muss in unserer neuen Klasse definiert werden. Sie wird immer dann aufgerufen, wenn die Verbindung hergestellt werden soll. Der Aufruf der Methode `manageConnectionEstablished` sorgt dafür, dass das Property gesetzt wird, an dem der Anwender erkennen kann, ob der Verbindungsaufbau erfolgreich war. Falls eine Exception auftritt, wird der Anwender darüber informiert, daß es zu einem Problem gekommen ist.

```
private Socket socket;
private PrintWriter pw;

protected void manageConnectionImpl()
{
 boolean old=isConnectionEstablished();
 try
 {
  if(port>-1)
  {
   socket = new Socket(InetAddress.getByName(host), port);
   pw=new PrintWriter(socket.getOutputStream());
  }
 }
 catch (IOException ex)
 {
  //warn für die Meldung eines Problems
  warn(null,ex.getMessage());
 }
 manageConnectionEstablished(old,socket!=null);
}
```

Die Methode `void closeDown()` muss wie oben schon beschrieben in unserer neuen Klasse definiert werden. Sie hat die Aufgabe, die allokierten Ressourcen freizugeben. Sie wird an verschiedenen Stellen aufgerufen: Sie ist das Arbeitspferd der Action zum Beenden der Verbindung und sie wird auch aufgerufen, wenn das Modul vom Arbeitsplatz entfernt wird. Die Implementierung muss damit zurechtkommen, dass sie aufgerufen wird, wenn gar keine Verbindung besteht (die Ressourcen nicht allokiert sind). Falls eine Exception auftritt, wird der Anwender darüber informiert, daß es zu einem kritischen Fehler gekommen ist.

```
protected void closeDown()
{
 boolean old=isConnectionEstablished();
 if(socket!=null)
 {
  try
  {
   pw.close();
   socket.close();
```

```
  }
  catch (IOException ex)
  {
   //error für die Meldung eines kritischen Fehlers
   error(null,ex.getMessage());
  }
  socket=null;
 }
 manageConnectionEstablished(old,socket!=null);
}
```

# Kapitel 19. De-/aktivierbare Module

## Funktionsbeschreibung

Die hier beschriebene Funktionalität hat gewisse Bezüge zum im Kapitel 18, *Module, die mit externen Ressourcen arbeiten müssen* beschriebenen Szenario: In der Datenfluss-Programmierung wird die Datenverarbeitung in einer Verarbeitungseinheit oder einem Modul fast immer durch ankommende Eingangsdaten oder Konfigurationsänderungen angestoßen.

Manchmal ist es aber so, dass Module entwickelt werden müssen, die Daten aus sich heraus produzieren und dazu nicht auf einen externen Stimulus warten können oder wollen. Solche Module verbrauchen aber unter Umständen viel Rechenzeit und so wäre es eine gute Sache, wenn der Anwender, der einen Workspace zusammenbaut, solche Module auf Wunsch ab- und dann wieder zuschalten könnte.

Wir stellen hier ein sehr einfaches solches Modul vor, das nach dem Start einfach eine Reihe von skalaren Zufallszahlen erzeugt. Den vollständigen Code für dieses Modul kann man im Anhang A, *Quellcode* in „Start/Stop“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

## Selber machen oder Basisklasse benutzen?

Das Modul muss für den Anwender zwei Actions im Kontextmenü anbieten - nämlich die zum Starten und Stoppen. Wir benutzen daher eine bereits im Framework mitgelieferte Klasse als Basisklasse unserer Implementierung, die diese Actions bereits mitbringt:

### Tabelle 19.1. Actions zum Starten und Stoppen der Verarbeitung

| |
|---|
| Starten der Verarbeitung |
| Stoppen der Verarbeitung |

Das hat unter anderem zwei Vorteile: zum einen muss man dann nicht mehr selbst die Methoden zum Listener-Management implementieren (add/remove), zum anderen existieren in dieser Klasse bereits eine Menge überladener send-Methoden zum Versenden der PropertyChangeEvents.

```
import de.elbosso.dataflowframework.modules.StartStopModule;
import java.util.Random;

public class FastRandomSkalar extends StartStopModule
{
 public FastRandomSkalar()
 {
  super(FastRandomSkalar.class.getName());
 }
}
```

# Output

Wir definierten einen Output in der Funktionsbeschreibung des Moduls. Die Implementierung geschieht über eine Property des Typs `double`, die lediglich über eine `get`-Methode verfügt.

```
private double value;

public double getValue()
{
 return value;
}
```

# Start des Hintergrundprozesses

Die benutzte Basisklasse ist eine Spezialisierung des Moduls, das Arbeiten im Hintergrund erleichtert (vergleiche Pogrammierhandbuch dWb+ in Kapitel 5, *Arbeiten im Hintergrund (Threads)*). Allerdings startet der Hintergrundprozeß nicht automatisch - daher wird die Funktion `setRunning` überschrieben.

```
public void setRunning(boolean newrunning)
{
 super.setRunning(newrunning);
 if(isRunning()==true)
 {
  processData(java.lang.Boolean.TRUE);
 }
 else
 {
  processData(null);
 }
}
```

# Workload

Weiterhin muss die eigentliche Logik des Moduls noch implementiert werden - das bedeutet in diesem Fall die Erzeugung der Zufallszahl und deren Versenden an die angeschlossenen Konsumenten:

```
private final static Random rand=new Random(System.nanoTime());

protected void doWork(java.lang.Object ref)
{
 if(disposed==false)
 {
  double old=getValue();
  value=rand.nextDouble();
  send("value",old,getValue());
 }
}
```

# Erweiterung: Einzelschritt

Besonders während der Entwicklung der Funktionalität kann es nützlich sein, das Moul in einen Modus zu versetzen, in dem es auf Anforderung exakt ein Datum erzeugt und versendet. Da Module dieser Art wie bereits oben angesprochen sobald sie gestartet werden, Ausgaben in extrem hoher Frequenz produzieren, wäre es für den Anwender unmöglich, so schnell hintereinader die Aktionen zum Starten und Stoppen auszulösen.

Daher existiert eine weitere Basisklasse namens `StartStopModuleWithDoOnce`. Diese stellt eine weitere Action zur Verfügung, die die Produktion genau eines Datum auslöst:

### Tabelle 19.2. Actions zum Starten und Stoppen der Verarbeitung

| | |
|---|---|
| | Ausführen eines einzelnen Verarbeitungsschritts zur Erzeugung exakt eines Datums |

Um das hier gegebene Beispiel entsprechend zu ändern, müssen wir von dieser Basisklasse ableiten:

```
import de.elbosso.dataflowframework.modules.StartStopModuleWithDoOnce;
import java.util.Random;

public class FastRandomSkalarWithDoOnce extends StartStopModuleWithDoOnce
{
 public FastRandomSkalarWithDoOnce()
 {
  super(FastRandomSkalarWithDoOnce.class.getName());
 }
}
```

Die restlichen Aspekte der Klasse bleiben gleich - lediglich die Erzeugung des Outputs wird wie folgt abgeändert:

```
protected void doWork(java.lang.Object ref)
{
 if(disposed==false)
 {
  performWork();
 }
}
```

Die nun noch fehlenden Teile der Implementierung folgen hier:

```
private void performWork()
{
 double old=getValue();
 value=rand.nextDouble();
 send("value",old,getValue());
}
```

```
public void doOnce()
{
 performWork();
}
```

Den vollständigen Code für dieses Modul kann man im Anhang A, *Quellcode* in „Start/Stop mit Einzelschritt“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

# Kapitel 20. Java Api for XML Binding

## Marshaling

Unter Marshaling versteht man die Erzeugung eines XML-Dokumentes aus einem Java-Objekts (einer Instanz einer Java-Klasse). dWb+ bietet dazu ein Modul, das in der Lage ist, Instanzen jeglicher mittels des Werkzeugs xjc instrumentierten Java-Klassen in einen XML-String umzuwandeln.

Falls also die Notwendigkeit entsteht, für die Umsetzung einer bestimmten Anforderung neue Klassen zu schaffen, deren Instanzen per JAXB in XML-Dokumente gewandelt werden sollen, genügt es, diese Klassen zu kompilieren und dem Modul-ClassLoader zugänglich zu machen (vergleiche dazu „modules“)

## Unmarshaling

Unter Unmarshaling versteht man das Einlesen und Parsen eines XML-Dokumentes, um mittels der gewonnenen Informationen eine Instanz einer Java-Klasse daraus erzeugen und ihren internen Status rekonstruieren zu können.

Leider kann man dazu kein allgemeingültiges Modul schaffen, da ein solches Modul ja naturgemäß immer einen aneren Ausgangstyp haben müsste. Möchte man XML-Dokumente in dWb+ benutzen können, um daraus Instanzen von Java-Klassen zu erzeugen und diese dann weiterzuverwenden, muß man diese Klassen zunächst schaffen. Das kann zum Beispiel durch Anwendung des Werkzeugs xjc auf das entsprechende XML-Schema geschehen. Anschließend müssen diese Klassen kompiliert und dem Modul-ClassLoader zugänglich gemacht werden (vergleiche dazu „modules“)

Zusätzlich dazu muß noch ein Modul pro Java-Klasse geschrieben werden, die aus XML-Dokumenten rekonstruiert werden sollen.

## Selber machen oder Basisklasse benutzen?

Man kann natürlich einen JAXB-Parser auch selbst schreiben - allerdings besteht dazu kein Grund, solange man damit leben kann, daß der Inhalt des XML-Dokuments als Java-String vorliegen muß: Es existiert bereits eine Basisklasse, von der man einfach ableiten kann. Diese abgeleitete Klasse muß einen Typparameter spezifizieren und die Klasse angeben, für deren Unmarshaling sie verantwortlich sein soll: In dem folgenden Beispiel soll ein Modul entstehen, das in der Lage ist, Dokumente in Instanzen der Klasse Shiporder aus dem Namensraum generated.example1 zu wandeln:

```
import de.elbosso.dataflowframework.modules.JaxbBase;
import generated.example1.Shiporder;

public class ShipOrder extends JaxbBase<Shiporder>
{
 public ShipOrder()
 {
  super(Shiporder.class);
 }
}
```

Das ist tatsächlich schon alles - Damit können jetzt Dokumente, die dem Schema folgen, aus dem die Klasse Shiporder im Namensraum generated.example1 generiert wurde, in Instanzen dieser Klasse umgewandelt werden.

# Kapitel 21. Verteiltes Arbeiten (Remoting)

## Warum?

Manche Aufgaben sind auf dedizierten Rechnern besser aufgehoben. Gründe können übermäßiger Bedarf an Rechenleistung ebenso sein, wie Zugriff auf spezielle Ressourcen, die nur in gewissen Rechnern zur Verfügung stehen. dWb+ stellt für diese Fälle eine spezielle Modulkategorie zur Verfügung, die vom Master (dem Rechner, der dWb+ ausführt) einfach auf andere Rechner (sogenannte Remoting Server) verlagert werden können. Alles was der Anwender dazu tun muß, ist, per Mausklick zu bestimmen, auf welchem Remoting Server die Funktionalität des Moduls bearbeitet werden soll.

## Ein einfaches Beispiel

Dieses Kapitel wird die Möglichkeiten und Fallstricke der Arbeit mit solchen Modulen an einem einfachen Beispiel beleuchten. Es gibt auf Anfrage lediglich diverse Angaben über sich selbst zurück, unter anderem, auf welchem Rechner es gerade ausgeführt wird. Den vollständigen Code für dieses Modul kann man im Anhang A, *Quellcode* in „Modul“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

## Selber machen oder Basisklasse benutzen?

Das Erstellen eines solchen Modules erfordert sehr viel Hintergrundwissen über dWb+. Daher ist das einzige praktikable Herangehen die Benutzung der Basisklasse RemoteModule im Paket de.elbosso.dataflowframework.modules. Dieses Modul ist eine Ableitung der Klasse ThreadingBeanContextChildModuleBase aus Paket de.netsysit.dataflowframework.modules

Wird von dieser Klasse abgeleitet, ist zu beachten, dass sie keinen parameterlosen Konstruktor besitzt - Der Basisklassenkonstruktor erwartet einen Namen, den dann der gestartete Thread erhält. Im Beispiel benutzen wir den Namen der Modulklasse.

```
import de.elbosso.dataflowframework.modules.RemoteModule;
import de.elbosso.dataflowframework.modules.helper.rmi.Hello;

public class RemotingExample extends RemoteModule<Hello>
{
 public RemotingExample()
 {
  super(RemotingExample.class.getName());
 }
}
```

Die Kommunikation zwischen dem Thread, der für die Kommunikation der Module untereinander verantwortlich ist und in dessen Kontext zum Beispiel auch die Methoden ausgeführt werden, die die Input-Slots modellieren und dem Thread, der den Algorithmus ausführt, geschieht über sogenannte CubbyHoles. Da die Basisklasse, bereits von ThreadingModuleBase erbt, wird die entsprechende Methode bereits dort

geeignet überschrieben. Für dieses Beispiel stellt uns sie Basisklassen-Variante zufrieden, daher müssen wir hier keine eigene Implementierung der Methode createCubbyHole() vorsehen.

Basis der Verlagerung von Fuktionalität auf einen Remoting Server ist die Implementierung dieser Funktionalität in einer konkreten Klasse und die Vereinbarung eines Interface, das durch diese Klasse implementiert wird. Das Interface ist der Template-Parameter für unsere neue Klasse - im Beispiel also Hello. Die Implementierung erfolgt in der Klasse HelloImpl:

```
import java.io.Serializable;

public interface Hello extends Serializable
{
   public String hello();
}

import java.net.Inet4Address;
import java.net.UnknownHostException;

public class HelloImpl implements Hello
{
 public String hello()
 {
  String rv="Hi there! - dont know where I am! ";
//  System.out.println("hello() called");
  try
  {
   rv="Hi there! - from "+
    Inet4Address.getLocalHost().getHostName()+
    " ";
  }
  catch (UnknownHostException ex)
  {
  }
  return rv+Thread.currentThread().getName()+" "+toString();
 }
}
```

Zu beachten isst, daß das Interface und die implementierende Klasse, sowie alle weiteren Klassen, die zur korrekten Umsetzung der beabsichtigten Funktionalität benötigt werden, in eine eigene Jar-Datei zu packen sind. Diese Jar-Datei ist dann in das Konfigurationsverzeichnis von dWb+ im Unterverzeichnis remoting zu platzieren!

Die Basisklasse benötigt Informationen darüber, welche Funktionalität bereitgestellt werden soll. Dies geschieht über die Implementierung zweier abstrakten Methoden - einer für die Bereitstellung der Funktionalität auf einem Remoting Server und einer für die lokale Abarbeitung:

```
protected void installWorker(java.lang.String newLocation)
{
 installWorker(newLocation,HelloImpl.class,new Class[]
  {
   Hello.class
  });
```

```
}
protected Hello createLocalInstance()
{
 return new HelloImpl();
}
```

# Input

Der Input wird über die Implementierung einer entsprechenden Methode definiert - in unserem Fall hat diese Methode die Signatur `public void input(java.lang.Object in)`. Im Körper dieser Methode wird nichts anderes getan, als den Algorithmus zu starten - weiter unten dazu mehr.

Der Start der Bearbeitung im Thread geschieht durch Senden eines Datenobjektes an das CubbyHole dieses Moduls.

```
public void input(Object in)
{
 processData(in);//Hierdurch Start des Algorithmus
}
```

# Konfiguration

Unser Beispiel verfügt über keine speziellen Konfigurationseinstellungen

# Output

Die Implementierung des Output wird über einen Monitor abgesichert.

```
private Object data;

public synchronized Object getData()
{
 return data;
}
public synchronized void setData(Object data)
{
 java.lang.Object old=getData();
 this.data=data;
 send("data",old,getData());
}
```

# Algorithmus

Der Algorithmus für unser Beispiel ist denkbar einfach - Der Programmierer muß dazu eine Referenz auf die Klasse holen, die die Funktionalität implementiert.

Die Durchführung geschieht in der Methode `doWork`. Diese Methode hat einen Parameter - hier bekommt die Methode das Objekt übergeben, das in das CubbyHole geschrieben wurde, um die Ausführung zu starten.

Der Zugriff auf alle Variablen, die mit Inputs, der Konfiguration oder Outputs zu tun haben erfolgt ausschließlich über Monitore aus dem Algorithmus-Thread heraus.

```
@Override
protected void doWork(Object ref) throws InterruptedException
{
 Hello h=getRemoteObject();
 if(h!=null)
 {
  setData(h.hello());
 }
}
```

# Deployment

Ein wichtiger Unterschied zu normalen Modulen ist die Form des Deployments. Die Funktionalität solcher Module ist in zwei Jar-Dateien aufzuteilen: Das Modul als solches kann mit allen benötigten Klassen in eine Jar-Datei gebündelt werden und ganz normal in das Unterverzeichnis modules im Datenverzeichnis abgelegt werden (siehe Anhang A, *Verzeichnis-Layout*).

Die Klassen, die die Funktionalitäten enthalten, die auf andere Rechner verlagert werden sollen, müssen aber auch von den anderen Rechnern geladen werden können. Dies geschieht über den HTTP-Server in dWb+. Damit das gelingt, muß der HTTP-Server diese Klassen kennen. Daher werden diese Klassen in einer eigenen Jar-Datei zusammengefasst und im Datenverzeichnis im Unterverzeichnis remoting abgelegt. In unserem Beispiel betrifft das die Klasse HelloImpl und das Interface Hello. Sind die Funktionalitäten in einem komplexeren, realen Anwendungsfall über mehrere Klassen verteilt, müssen alle diese in der Jar-Datei zusammengefasst und im Verzeichnis remoting abgelegt werden.

# Kapitel 22. Getaktete Module

## Funktionsbeschreibung

Bisher haben wir Module beschrieben und entwickelt, die nach Eingang irgendeines Datums an einem der Eingänge eine Vararbeitung der Daten gestartet haben. Dieser Abschnitt soll ein Beispiel für ein Modul zeigen, das die eingehenden Daten speichert und die Verarbeitung erst startet, wenn an einem bestimmten, ausgezeichneten Eingang ein Datum empfangen wird.

Im Prinzip kann man das mit normalen Bean-Properties erreichen, die im Setter die übergebenen Werte in Instanzvariablen speichern. Eine weitere Alternative sind ganz normale private Instanzvariablen. In beiden Fällen würde das den Quelltext stark aufblähen und - speziell im Fall von ThreadingModules - es müssten spezielle Vorkehrungen getroffen werden, um die Zugriffe threadsafe zu machen. Daher bietet dWb+ eine Klasse an, die für genau diesen Zweck geschaffen wurde.

Als Beispiel soll ein Modul mit drei Eingängen dienen: Das Modul soll zwei boolesche Eingangssignale mittels einer logischen UND-Funktion zu einem Ergebnis verrechnen. Diese Berechnung soll nicht sofort stattfinden, wenn sich eines der Eingangssignale ändert, sondern nur, wenn sich der Status des Takt-Eingangs ändert. Den vollständigen Code für dieses Modul kann man im Anhang A, *Quellcode* in „Getaktete Module“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

## Selber machen oder Basisklasse benutzen?

Die hier beschriebene Methode der Speicherung eingehender Daten bis zur tatsächlichen Durchführung der Berechnung kann in beliebigen Klassen zur Anwendung kommen. Sie beeinflusst nicht die Auswahl der Basisklasse.

Wir werden für dieses Beispiel wieder die einfachste Variante wählen und unser neues Modul von ModuleBase im Paket de.netsysit.dataflowframework.modules ableiten. Wir legen auch gleich eine Instanz der Klasse an, die uns beim Speichern der Eingangsdaten helfen wird:

```
import de.netsysit.dataflowframework.modules.ModuleBase;
import de.elbosso.util.Latch;

public class ClockedAnd extends ModuleBase
{
 private Latch latch;

 public ClockedAnd()
 {
  super();
  latch=new Latch();
 }
}
```

# Input

Der Input wird über die Implementierung von drei Methoden modelliert: zwei für die Inputs. Im Körper dieser Methoden wird nichts anderes getan, als die Eingangsdaten zu speichern.

```
public void inputA(boolean in)
{
 latch.latch("inputA",in);
}
public void inputB(boolean in)
{
 latch.latch("inputB",in);
}
```

Die dritte Methode, die den Takteingang darstellt, löst schließlich die Verarbeitung aus. Für dieses Beispiel soll es egal sein, was an Daten in diesen Eingang fließt - sie werden in der Verarbeitung ignoriert.

```
public void clock(java.lang.Object in)
{
 compute();
}
```

# Konfiguration

Wir geben diesem Modul keinerlein Konfigurationsmöglichkeiten mit auf den Weg.

# Output

Wir definierten einen Output in der Funktionsbeschreibung des Moduls. Die Implementierung geschieht über eine Property des entsprechenden Typs, die lediglich über eine get-Methode verfügt. Weiterhin wird eine Instanzvariable angelegt, die den letzten berechneten Wert aufnimmt, da man zum Versenden eines PropertyChangeEvents ja immer den alten und den neuen Wert benötigt.

```
private boolean result;

public boolean getResult()
{
 return result;
}
```

# Algorithmus

Der Algorithmus für unser Beispiel ist denkbar einfach - der Programmierer muss die gespeicherten Werte für die beiden Inputs logisch verknüpfen und PropertyChangeEvents für alle gewonnenen Resultate versendet.

```
private void compute()
{
 boolean old=getResult();
 //inputA holen
 boolean a = latch.fetchBoolean("inputA");
 //inputb holen
 boolean b = latch.fetchBoolean("inputB");
 result=a&&b;
 //Events versenden!
 send("result",old,getResult());
}
```

# Kapitel 23. Mandantenfähigkeit

## Warum?

Beim Stichwort Mandantenfähigkeit geht es darum, dass es in datenflussorientierten Umgebungen durchaus so sein kann, dass neben den eigentlichen Daten noch Informationen zu ihrer Entstehung für die Verarbeitungseinheiten weiter hinten in der Verarbeitungskette interessant sein könnten. Einige Szenarien dafür werden in „Context (Mandantenfähigkeit)“ vorgeschlagen.

Die Marker oder Kontextinformationen werden nicht einfach an das erzeugte Datum gehängt: in der Verarbeitungskette werden in jeder Verarbeitungseinheit typischerweise neue Daten aus den Eingangsdaten erzeugt - Wenn man Kontextinformationen nur an die eigentlichen Nutzdaten anhängen würde, sind diese spätestens nach der nächsten Verarbeitungseinheit verloren, wenn diese neue Daten aus den ursprünglichen Nutzdaten erzeugt.

Es wird sichergestellt, dass die Kontextinformationen, die in einer Verarbeitungseinheit mit den Input-Daten ankommen, an die Resultate angehängt werden. Die Kontextdaten müssen also entlang der Verarbeitungskette von Nutzdatum zu Nutzdatum übertragen werden. Der Entwickler eines Modules, das weder Daten in den Context einspeist, noch Daten daraus entnimmt, muss sich mit dem Thema Context nicht auseinandersetzen - die Behandlung ist völlig transparent für ihn.

## Selber machen oder Basisklasse benutzen?

Um den Kontext nutzen zu können, ist es zwingend notwendig, Module von den Basisklassen BeanContextChildModuleBase oder ThreadingBeanContextChildModuleBase mittelbar oder unmittelbar abzuleiten. Später wird eventuell eine Möglichkeit geschaffen, auch mit Modulen umgehen zu können, die nicht von diesen beiden Basisklassen abgeleitet wurden.

## Daten in den Context einspeisen

### BeanContextChildModuleBase

Daten werden in den Context eingespeist, indem die Methode `setNamedContextItemForEntity` mit drei Parametern aufgerufen wird: der erste ist die `this`-Referenz auf das Modul selbst, der zweite der Name unter dem die Daten im Context gespeichert werden sollen und der dritte eine Referenz auf die Daten, die in den Context eingespeist werden sollen.

### ThreadingBeanContextChildModuleBase

Daten werden in den Context eingespeist, indem die Methode `setCurrentContextItem` mit zwei Parametern aufgerufen wird: der erste ist der Name unter dem die Daten im Context gespeichert werden sollen und der zweite eine Referenz auf die Daten, die in den Context eingespeist werden sollen.

beide Methoden müssen jeweils vor dem Versenden eines Datums mittels PropertyChangeEvents aufgerufen werden: Der direkt darauf folgende Aufruf von `send` markiert die zu versendenden Daten dann mit dem angegebenen Context.

# Daten aus dem Context auslesen

## BeanContextChildModuleBase

Daten werden aus dem Contest ausgelesen, indem die Methode `fetchNamedItemFromContextForEntity` mit drei Parametern aufgerufen wird: der erste Parameter ist die `this`-Referenz auf das Modul selbst, der zweite der Name, unter dem der Wert im Context abgelegt wurde und der dritte die Klasse des abgelegten Wertes.

## ThreadingBeanContextChildModuleBase

Daten werden aus dem Contest ausgelesen, indem die Methode `fetchNamedItemFromCurrentContext` mit zwei Parametern aufgerufen wird: der erste Parameter ist der Name, unter dem der Wert im Context abgelegt wurde und der zweite die Klasse des abgelegten Wertes.

Existiert kein Datum des angegebenen Wertes oder ist der Typ des unter diesem Namen abgelegten Datums nicht kompatibel mit dem angegebenen Klassen-Parameter, liefert diese Methode `null` zurück.

# Kapitel 24. Enterprise JavaBeans (EJBs) als Module

## Funktionsbeschreibung

Die hier beschriebene Funktionalität hat gewisse Bezüge zum im Kapitel 21, *Verteiltes Arbeiten (Remoting)* beschriebenen Szenario: Komponenten, die auf einem anderen Rechner ausgeführt werden, stellen ihre Funktionalität für die in dWb+ orchestrierten Workspaces zur Verfügung

Dieses Mal geht es allerdings um Komponenten, die bereits in Enterprise-Anwendungen zum Einsatz kommen. Es handelt sich um Enterprise JavaBeans, die auf einem oder mehreren Application-Servern (etwa JBoss Wildfly oder ähnlichen) ausgerollt sind.

Verfügen solche EJBs über mindestens ein Remote-Interface, steht dieses Interface und alle mittelbar und unmittelbar benötigten Klassen (Parameter und Rückgabewerte der definierten Methoden, sowie gegebenenfalls Super-Interfaces) zur Laufzeit von dWb+ zur Verfügung (vergleiche Anhang A, *Verzeichnis-Layout* und „Programmstart“ im Anwenderhandbuch dWb+), kann man das betreffende EJB in dWb+ benutzen, indem man ein Wrapper-Modul erschafft.

Wir stellen hier ein sehr einfaches solches Modul vor, das eine sehr einfache stateless Bean kapselt: die gekapselte EJB addiert zwei Zahlen und liefert das Ergebnis zurück. Damit ist das Remote-Interface sehr übersichtlich:

```
public interface RemoteCalculator
{
    int add(int a, int b);
}
```

Das Modul selbst wird über zwei Eingäge für die Summanden und einen Ausgang für das Ergebnis verfügen. Den vollständigen Code für dieses Modul kann man im Anhang A, *Quellcode* in „Enterprise JavaBeans als Module“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

## Selber machen oder Basisklasse benutzen?

Referenzen auf Proxies zum Aufruf der Funktionen der EJBs werden per JNDI bezogen. Dazu wird ein InitialContext benötigt. Die Erstellung dieses Contexts und der Bezug der Referenz, bzw. die Konstruktion desselben übernimmt eine Basisklasse. Diese Basisklasse ist generisch - abgeleitete Klassen müssen den Typ des Remot-Interface bei der Ableitung als Parameter angeben. Dieser wird auch als erster Parameter des Konstruktors benötigt.

```
import de.elbosso.dataflowframework.modules.EJBModuleBase;
import java.util.Properties;
import de.netsysit.util.threads.CubbyHole;
import de.netsysit.util.threads.SimpleBufferingCubbyHole;
```

```
public class EJBWrapper extends  EJBModuleBase<RemoteCalculator>
{
 public EJBWrapper()
 {
  super(RemoteCalculator.class,EJBWrapper.class.getName());
 }
}
```

Die benutzte Basisklasse ist eine Spezialisierung des Moduls, das Arbeiten im Hintergrund erleichtert (vergleiche Pogrammierhandbuch dWb+ in Kapitel 5, *Arbeiten im Hintergrund (Threads)*).

Die Kommunikation zwischen dem Thread, der für die Kommunikation der Module untereinander verantwortlich ist und in dessen Kontext zum Beispiel auch die Methoden ausgeführt werden, die die Input-Slots modellieren und dem Thread, der den Algorithmus ausführt, geschieht über sogenannte CubbyHoles. Jede Klasse, die von ThreadingModuleBase erbt, muss daher eine Instanz zur Verfügung stellen, die dieses Interface implementiert. Wir benutzen für dieses Beispiel eine sehr einfache Implementierung.

```
@Override
protected CubbyHole createCubbyHole()
{
 return new SimpleBufferingCubbyHole();
}
```

# JNDI Lookup

Alles, was der Wrapper für den Bezug der Referenz, bzw. die Konstruktion des Proxy beisteuern muss, ist ein Objekt vom Typ `Properties`, das die erforderlichen Properties zur Konfiguration des InitialContext enthält und den Basis-JNDI-Namen des EJB. Im Beispiel wird eine Konfiguration zum Zugriff auf eine EJB in einem Wildfly 10 (JBoss) ApplicationServergezeigt:

```
protected Properties getEnvironment()
{
 java.util.Properties env = new java.util.Properties();
 env.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
  "org.jboss.naming.remote.client.InitialContextFactory");
 env.put(javax.naming.Context.PROVIDER_URL,
  "http-remoting://da_host:da_port");
 env.put(javax.naming.Context.SECURITY_PRINCIPAL,
  "da_user");
 env.put(javax.naming.Context.SECURITY_CREDENTIALS,
  "da_password");
 env.put(javax.naming.Context.URL_PKG_PREFIXES,
  "org.jboss.ejb.client.naming");
 env.put("jboss.naming.client.ejb.context",
  true);
 return env;
}
protected String getJndiName()
{
```

```
 return "ejb:/wildfly-ejb-remote-server-side/CalculatorBean!";
}
```

Host, Port, Login und Passwort sind natürlich den lokalen Gegebenheiten anzupassen!

# Output

Wir definierten einen Output in der Funktionsbeschreibung des Moduls. Die Implementierung geschieht über eine Property des Typs `int`, die lediglich über eine `get`-Methode verfügt.

```
private int result;

public int getResult()
{
 return result;
}
```

# Input

Der Input wird über die Implementierung zweier entsprechender Methoden definiert. Im Körper dieser Methoden wird nichts anderes getan, als den Algorithmus zu starten - weiter unten dazu mehr.

Der Start der Bearbeitung im Thread geschieht durch Senden eines Datenobjektes an das CubbyHole dieses Moduls.

```
private Number a;
private Number b;

public void inputA(Number in)
{
 a=in;
 processData(in);Hierdurch Start des Algorithmus
}
public void inputB(Number in)
{
 b=in;
 processData(in);Hierdurch Start des Algorithmus
}
```

## Workload

Weiterhin muss die eigentliche Logik des Moduls noch implementiert werden - das bedeutet in diesem Fall die Addition der Summanden und das Versenden der Summe an die angeschlossenen Konsumenten:

```
protected void doWork(Object ref)
{
 if((a!=null)&&(b!=null))
```

```
 {
  int old=getResult();
  result=ejb.add(a.intValue(),b.intValue());
  send("result",old,getResult());
 }
}
```

# Benutzung als Skripted Module

Wem die Arbeit des Compilierens, des Packaging und des Ausrollens (Kapitel 26, *Packaging*) eines Wrappers zu viel Arbeit für die simple Nutzung einer EJB erscheint, kann ein solches Modul natürlich auch als Skripted Modul benutzen - vergleiche dazu „Skript-Module“ im Anwenderhandbuch dWb+.

# Kapitel 25. Module mit geänderter Kommunikationsmetapher

## Warum?

dWb+ entstand eigentlich als System, in dem die Ein- und Ausgänge der Verarbeitungseinheiten streng typisiert sind. Es gibt aber natürlich Anwender, die solch eine strenge Typisierung als einschränkend odr hinderlich ansehen.

Aus diesem Grund soll hier die Möglichkeit einer typlosen Datenübertragung beschrieben werden und ein Beispiel für ein Modul, das diese nutzt, implementiert werden.

Grundlage der Idee ist die, dass zwischen den Modulen immer nur ein Datentyp ausgetauscht wird - bei diesem Datentyp handelt es sich um assoziative Maps oder Dictionaries: Jedes Modul entnimmt die Daten, die es als Input benötigt aus der eingehenden Map und schreibt seine Ergebnisse in die Map bevor es sie an nachfolgende Module weiterreicht.

## Ein einfaches Beispiel

Dieses Kapitel wird die Möglichkeiten und Fallstricke der Arbeit mit solchen Modulen an einem Beispiel verdeutlichen, das die Aufgabe hat, zwei Zahlen zu addieren. Den vollständigen Code für dieses Modul kann man im Anhang A, *Quellcode* in „Module mit geänderter Kommunikationsmetapher“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

## Selber machen oder Basisklasse benutzen?

Da es bei untypisierten Ein- und Ausgangsdaten einiges mehr zu beachten gilt - schließlich muss der Autor eines Moduls selbst checken, ob die in der Map vorliegenden. Daten dem entsprechen, was sein Modul benötigt, um arbeitsfähig zu sein - wird vorgeschlagen, eine Basisklasse zu benutzen, die einige der anfallenden Aufgaben bereits übernimmt. Diese Basisklasse wiederum leitet sich ab von `ThreadingModuleBase`, die bereits im Kapitel 5, *Arbeiten im Hintergrund (Threads)* besprochen wurde.

Zunächst also geht es an die Erstellung des Gerüsts der neuen Klasse:

```
import de.netsysit.util.beans.InterfaceFactory;
import de.elbosso.dataflowframework.modules.MapMessageModule;
import de.netsysit.dataflowframework.modules.ThreadingBeanContextChildModuleBase;
import de.netsysit.util.threads.CubbyHole;
import de.netsysit.util.threads.SimpleNonBlockingCubbyHole;

public class MappedAdder extends MapMessageModule
{
 static
 {
  InterfaceFactory.setSuperclassAssociationForEventDispatchThread(
```

```
  MappedAdder.class, ThreadingBeanContextChildModuleBase.class);
}

private double result;

public MappedAdder()
{
 super(MappedAdder.class.getName());
 java.util.Properties props=new java.util.Properties();
 props.setProperty("a", "a");
 props.setProperty("b", "b");
 setProps(props);
}
```

Der Konstruktor erfüllt hierbei eine neue Aufgabe: Jedes der Module, die von `MapMessageModule` abgeleitet werden, bietet dem Anwender die Möglichkeit, ein Mapping der Daten, die in der eingehenden Map enthalten sind auf die Parameternamen, die das Modul erwartet, vorzunehmen. Damit der Anwender später nicht ganz orientierungslos vor dem entsprechenden Dialog sitzt, füllen wir hier ein Properties-Objekt mit den von unserem neuen Modul erwarteten Namen. Der Anwender kann diesen erwarteten Namen dann diejenigen zuordnen, die in der Map die korrekten Parameter darstellen.

Die Kommunikation zwischen dem Thread, der für die Kommunikation der Module untereinander verantwortlich ist und in dessen Kontext zum Beispiel auch die Methoden ausgeführt werden, die die Input-Slots modellieren und dem Thread, der den Algorithmus ausführt, geschieht über sogenannte CubbyHoles. Jede Klasse, die von ThreadingModuleBase erbt, muss daher eine Instanz zur Verfügung stellen, die dieses Interface implementiert. Wir benutzen für dieses Beispiel eine sehr einfache Implementierung.

```
@Override
protected CubbyHole createCubbyHole()
{
 return new SimpleNonBlockingCubbyHole();
}
```

# Input

Der Input wird über die Basisklasse definiert - per Definition existiert beidiesen Modulen lediglich ein einziger Import - daher braucht ein Entwickler sich darum überhaupt nicht zu kümmern.

# Konfiguration

Dieses einfache Modul benötigt keine weitere Parametrierung - für die GUI zur Spezifikation des Mappings zwischen den erwarteten und tatsächlich in der Map enthaltenen Schlüssel sorgt - wie bereits weiter oben ausgeführt - die Basisklasse.

# Output

Die Implementierung des Output wird ebenfalls von der Basisklasse übernommen.

# Algorithmus

Der Algorithmus für unser Beispiel ist denkbar einfach - der Programmierer muss lediglich die korrekten Methoden zum Zugriff auf die Elemente der Map aus der Basisklasse verwenden.

Die Implementierung geschieht in der Methode `doWork`. Diese Methode hat einen Parameter - dieser Parameter ist eine Referenz auf das Modul selbst - zum Zugriff auf die Elemente der Map existieren dedizierte Methoden!

```
@Override
protected void doWork(Object ref) throws InterruptedException
{
 //Kopie der eingehenden Map
 java.util.Map map=(java.util.Map)ref;
 Number a = null;
 Number b = null;
 //Versuch, die erwarteten Daten aus der Map zu holen
 a=(Number)getData(map,"a");
 b=(Number)getData(map,"b");
 if((a!=null)&&(b!=null))
 {
  result=a.doubleValue()+b.doubleValue();
  //Hinzufügen der Ergebnisses zur Map und Versenden
  java.util.Map old=null;
  map.put("result",result);
  send("output",old,map);
 }
}
```

# Kapitel 26. Packaging

Das Ausrollen fertiger Module ist einfach - Man muss lediglich eine Jar-Datei erzeugen, die die Klassendateien für die fertigen Module sowie alle unter Umständen von den Modulen benutzten Klassen enthält und diese Jar-Datei den potentiellen Anwendern zur Verfügung stellen.

Die Anwender benutzen die neuen Module, indem sie die Jar-Dateien in das entsprechende Unterverzeichnis ihres Konfigurationsverzeichnisses kopieren - vergleiche dazu auch „modules“ im Anwenderhandbuch dWb+.

Zur Erstellung der Jar-Dateien kann man zum Beispiel Maven-Projekte benutzen - Anregungen dazu findet man im Anhang C, *Erstellen von Komponenten für dWb+ mittels Maven* im Pogrammierhandbuch dWb+

# Kapitel 27. StateUpdaters

## Einführung

StateUpdaters dienen der Darstellung der zuletzt versendeten Daten an den Output-Slots. Sie ersetzen dazu die normalen Tooltips an den Slots, sobald das erste Datum versendet wurde - vergleiche dazu auch „Tooltips für Outputs“. Dazu existiert eine Registrierungsdatenbank, die für jeden Datentyp, der versendet werden kann, vermerkt, ob es einen passenden StateUpdater gibt.

Man kann sehr einfach eigene StateUpdater erstellen und in dWb+ einklinken. Alles, was man dazu zu tun hat ist, die entsprechende Implementierung abzuschließen, eine Jar-Datei mit allen benötigten Klassen zu erzeugen und in dieser Jar-Datei eine Liste zu integrieren, die die Zuordnung zwischen Typ des Output-Slots und StateUpdater-Implementierung festlegt. Diese Datei muss dann vom Anwender in das korrekte Unterverzeichnis des Konfigurationsverzeichnisses kopiert werden und nach dem nächsten Neustart steht der StateUpdater zur Verfügung.

Die Zuordnung der benutzten StateUpdater nimmt dabei nicht nur die exakte Klasse des Output-Slot-Typs, sondern alle Superklassen bis hin zu Object. Dabei wird immer diejenige Assoziation (und damit derjenige StateUpdater) benutzt, dessen Klasse der des Output-Slots in der Vererbungshierarchie am nächsten ist. Ein kleinen Beispiel soll das veranschaulichen: Die Registrü enthält StateUpdaterA für den Output-Slot-Typ Number und StateUpdaterB für OutputSlot-Typ Integer. Dann wird StateUpdaterA für die OutputSlots Double und Long benutzt, da deren Superklasse (beziehungsweise Superinterface) in der registry mit diesem StateUpdater verbunden ist. Für die Klasse Integer trifft das zwar auch zu, allerdings existiert eine Assoziation zwischen Integer und Stateupdater B. Da Integer in der Vererbungshierarchie näher an der Klasse des Output-Slots liegt (beziehungsweise in diesem Beispiel damit identisch ist), wird für Output-Slots vom Typ Integer StateUpdaterB benutzt.

Die soeben beschriebenen Mechanismen lassen sich weiter flexibilisieren: Wenn in der BeanInfo für die Klasse des Moduls, dessen ModulWidget den betreffenden Output-Slot enthält, für die korrespondierende Property ein Wert unter dem Schlüssel dWb::StateUpdaterClass hinterlegt ist, wird dieser Wert nach Aufruf von toString() als Klassenname der zu benutzenden StateUpdater-Implementierung benutzt. So kann man erreichen, daß für ein bestimmtes Modul die Visualisierung der Daten eines bestimmten Output-Slots von der Visualisierung von Daten gleichen Typs in anderen abweicht.

## Implementierung

### Ein einfaches Beispiel

Wir wollen einen Stateupdater schaffen, der für Ausgänge des Typs Number den letzten und aktuell verschickten Wert anzeigt.

Den vollständigen Code für dieses Modul kann man im Anhang A, *Quellcode* in „StateUpdater“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

### Selber machen oder Basisklasse benutzen?

Wir gehen hier von der einfachsten Variante der Implementierung einer neuen StateUpdater-Klasse aus: wir benutzen die bereits vorhandene Basisklasse PopupStateUpdater im Paket de.netsysit.dataflowframework.ui.beans:

```
import java.awt.GridLayout;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JList;
import de.netsysit.dataflowframework.ui.beans.PopupStateUpdater;
import de.elbosso.ui.moduleworkspace.connected.Slot;

public class NumberStateUpdaterDemo extends PopupStateUpdater
{
}
```

# Der Konstruktor

In unserem Beispiel setzen wir die Komponente, die den Tooltip darstellen soll, im Konstruktor zusammen - man könnte das auch mittels lazy initialization so lange hinausschieben, bis der Tooltip tatsächlich zum ersten Mal dargestellt werden soll. Da das ein einfaches Beispiel werden soll, werden nicht sehr viele Ressourcen verbraucht, so dass man mit der Initialisierung im Konstruktor leben kann.

```
 public NumberStateUpdaterDemo(Slot slot, JList list)
 {
  super(slot,list);
  JPanel p=new JPanel(new GridLayout(0, 2));
  JLabel l=new JLabel("letztes:");
  l.setOpaque(false);
  p.add(l);
  last=new javax.swing.JLabel("");
  p.add(last);
  l=new JLabel("aktuelles:");
  l.setOpaque(false);
  p.add(l);
  current=new javax.swing.JLabel("");
  p.add(current);
  toplevel.add(p);
  last.setOpaque(false);
  current.setOpaque(false);
  p.setOpaque(false);
 }
```

Der Konstruktor muss als Parameter eine Instanz vom Typ JLabel entgegennehmen und diese an die Basisklasse weiterreichen.

Im Konstruktor wird ein JPanel mit einem GridLayout konstruiert, das in 2x2 Zellen links jeweils einen Text enthält - die beiden rechten Zellen nehmen den jeweiligen Wert auf. Die in diesen Zellen vorhandenen Instanzen vom Typ JLabel sind - genauso wie die Referenz auf das letzte versendete Object - als Instanzvariablen ausgelegt.

Die Eigenschaft opaque sämtlicher benutzter Komponenten wird auf false gesetzt, damit die die Farbe des Tooltips so wie sie im System festgelegt ist annehmen.

Die Basisklasse stellt eine Komponente zur Verfügung, in die abgeleitete Klassen ihre GUI einklinken können. Diese Komponente heißt toplevel, ist vom Typ JPanel und hat als LayoutManager ein BorderLayout.

## Das Update

Die Methode, die immer dann aufgerufen wird, wenn der Output-Slot ein neues Datum versendet hat, sieht bei unserem Beispiel wie folgt aus:

```
protected void update(Object object)
{
 last.setText(old!=null?old.toString():"--");
 current.setText(object!=null?object.toString():"--");
 old=object;
}
```

In der Methode wird zunächst der Text der beiden JLabel unter Berücksichtigung eventuell vorhandener Null-Referenzen aktualisiert und am Ende der aktuelle Wert in der Instanzvariable old gespeichert.

# Rollout

Das Rollout neuer StateUpdater geschieht, indem alle benötigten Klassen in eine Jar-Datei gepackt werden. Zusätzlich müssen noch Registrierungsinformationen zu der Jar-Datei hinzugefügt werden: Dazu wird im Verzeichnis META-INF des Archivs eine Properties-Datei namens stateupdaters.properties angelegt, die für jeden StateUpdater eine Zeile enthält, die den Namen des Typs, für den der StateUpdater verantwortlich sein soll und den Namen der Klasse, die den StateUpdater darstellt, enthält. Beide Namen sind jeweils voll qualifiziert anzugeben - also mit sämtlichen Paketnamen.

In unserem Beispiel enthielte die Datei also lediglich eine Zeile:

```
java.lang.Number=ihr.paketname.hier.NumberStateUpdaterDemo
```

Zur Erstellung der Jar-Dateien kann man zum Beispiel Maven-Projekte benutzen - Anregungen dazu findet man im Anhang C, *Erstellen von Komponenten für dWb+ mittels Maven* im Pogrammierhandbuch dWb+

# Kapitel 28. Aviator Templates

## Gestaltung

Aviator Templates sind Teil des Mechanismus, um die virtuellen Cockpits aus den Meta-Modulen vom Typ Aviator in eigene Webseiten einbauen zu können. Diese Templates entsprechen normalen XHTML-Seiten, enthalten aber als Besonderheit einige Platzhalter, die von dWb+ beziehungsweise der dafür verantwortlichen Komponente während der Auslieferung der Seite an den Client entsprechend korrekt ersetzt werden müssen.

| | |
|---|---|
| $allscripts | Dieser Platzhalter wird durch Text ersetzt, der alle benötigten JavaScript-Fragmente in die Seite integriert. |
| $userId | Dieser Platzhalter wird mit dem Wert des Request-Parameters userId besetzt (so vorhanden) |
| $fileName | Dies ist der Platzhalter, der durch den Namen der anzuzeigenden SVG-Graphik ersetzt wird, so wie er auf dem Server abgelegt wurde. |

Der Entwickler muss lediglich eine XHTML-Seite schaffen, in der diese Platzhalter an den korrekten Stellen stehen. Als Beispiel dafür ist hier das Default-Template angegeben, das immer dann benutzt wird, wenn der Anwender kein eigenes Template spezifiziert

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang="en" xmlns="http://www.w3.org/1999/xhtml"
xmlns:svg="http://www.w3.org/2000/svg" xml:lang="en">
<head>
<meta http-equiv="content-type" content="text/html;
charset=ISO-8859-1" />
 <title>Aviator Demonstration</title>
<link href="css/my_layout.css" rel="stylesheet" type="text/css" />
 #foreach($script in $allscripts)1
<script type="text/javascript" src="js/$script">1
</script>1
 #end1
</head>
<body onload="dmcc_onload('$userId','$fileName')"2
onunload="dmcc_dynamic_update_off()">2

 <div id="dmcc_graphic"></div>3
<div><form>
<input type="button" value="Updaterate verringern"
onclick="dmcc_dynamic_update_inc()" />
<input type="button" value="Updaterate erhöhen"
onclick="dmcc_dynamic_update_dec()" />
</form></div>
<p id="rate">1000</p>
<p id="counter"></p>
<div><p id="System.out.println">
```

```
</p></div>

</body>
</html>
```

1. Diese vier Zeilen sorgen für die Einbindung der benötigten Skripts
2. Die Ereignisse onload und onunload des Body-Tag müssen mit diesen Funktionen verbunden werden
3. Dieser Div legt fest, wo das virtuelle Cockpit eingeblendet werden soll. Hier ist besonders die korrekte Benennung der ID des Div-Tag wichtig!

# Kapitel 29. Exportieren in eigenen Dateiformaten

## Warum?

Manchmal möchte man den konfigurierten Workspace auch anderweitig verwenden. Sei es zu Zwecken der Dokumentation, zur Generierung von Quelltext oder als Input für nachgeschaltete IT-Systeme. All das ist möglich, wenn der Programmierer eine Komponente implementiert, die die Datenstruktur des Workspace in das gewünschte Format transformiert.

## Wie?

Das Kontextmenü des Workspace verfügt über eine Action namens Export, die vom Anwender benutzt zunächst einen Dateiauswahldialog öffnet, in dem der Anwender festlegen kann, wie der Name der entstehenden Datei lauten soll. Die Anwendung wählt anschließend anhand der Endung dieses Dateinamens die für diesen Exportvorgang verantwortliche Komponente aus und übergibt dieser die Datenstruktur des Workspace. Diese Komponente hat wiederum die Aufgabe, aus der übergebenen Datenstruktur das gewünschte Format zu erzeugen und in die gewählte Datei zu speichern.

Um neue Formate zu unterstützen, muss der Programmierer also lediglich eine weitere dieser Komponenten schaffen und mit der Export-Action registrieren.

Den vollständigen Code für diese Komponente kann man im Anhang A, *Quellcode* in „DemoWorkspaceExporter“ finden.

Außerdem wurde der Code so wie alle anderen Beispiele auch auf GitHub [https://github.com/] im zugehörigen Repository [https://github.com/elbosso/dWb_custom_modules] veröffentlicht

## Implementierung

Um eine Komponente zu erstellen, die sich in das beschriebene Vorgehen einbinden lässt, muss man lediglich eine Klasse schaffen, die das dafür vorgesehene Interface mit lediglich zwei Methoden implementiert:

```
import de.netsysit.dataflowframework.logic.services.workspace.WorkspaceExporter.Su
import de.netsysit.dataflowframework.ui.LinkDescription;
import de.netsysit.dataflowframework.ui.ModuleWidgetDescription;
import de.netsysit.dataflowframework.ui.WorkspaceDescription;
import java.io.OutputStream;
import de.netsysit.dataflowframework.logic.services.workspace.WorkspaceExporter;
import de.netsysit.dataflowframework.ui.GraSim;

public class DemoWorkspaceExporter extends java.lang.Object implements
 WorkspaceExporter
{
}
```

Die Methode namens getSuffix ist für die Eingliederung verantwortlich, indem sie die Verbindung zwischen dem Dateinamen (beziehungsweise seiner Endung) und der verantwortlichen Komponente

herstellt. Dabei ist zu beachten, dass diese Methode die Endung ohne vorangestellten Punkt zurückliefern muss:

```
public String getSuffix()
{
 return "exdemo";
}
```

Die zweite Methode dieses Interface ist dann dafür verantwortlich, die Datenstruktur in das gewünschte Zielformat zu transformieren und in die Datei zu schreiben. Die Schnittstelle ist dabei nicht die Datei, sondern allgemeiner der Stream. Der Programmierer darf den Stream nicht schließen - das Freigeben von Ressourcen erledigt derjenige, der sie allokiert hat - in dem Falle also der Aufrufer der Methode!

Ausgehend von der WorkspaceDescription wird die Transformation auf die Ausgabe des Titels und der Modulklasse für jedes Modul gefolgt von der Spezifikation der Verbindung zwischen den Modulen beschränkt. Verschiedene weitere in der Datenstruktur enthaltene Informationen, wie etwa die Sichtbarkeit der verschiedenen Slots, die Typen der Slots oder auch den Aktivitätsstatus der Verbindungen werden ignoriert.

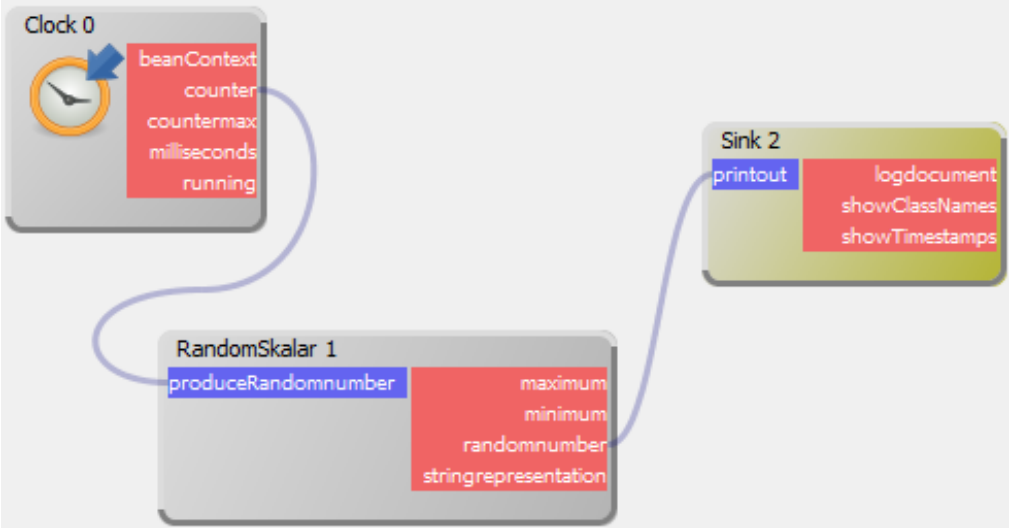
Die Datenstruktur enthält wesentlich mehr Informationen, die sich beim Studium der betreffenden API ergründen lassen - für das hier gezeigte einfache Beispiel sollen die Informationen genügen.

```
public boolean save(Support support,
 WorkspaceDescription[] wda, OutputStream os)
{
 boolean rv=false;
 java.io.PrintWriter pw=null;
 pw=new java.io.PrintWriter(os);
 if(wda!=null)
 {
  for (WorkspaceDescription workspaceDescription : wda)
  {
   ModuleWidgetDescription mwd[] =
    workspaceDescription.getModuleWidgetDescription();
   if(mwd!=null)
   {
    for (ModuleWidgetDescription moduleWidgetDescription : mwd)
    {
     pw.print(moduleWidgetDescription.getTitle());
     pw.print("\t");
     Object module=moduleWidgetDescription.getModule();
     pw.println(module.getClass().getName());
    }
   }
   LinkDescription ld[]=workspaceDescription.getLinkDescription();
   if(ld!=null)
   {
    for (LinkDescription linkDescription : ld)
    {
     pw.println(linkDescription);
    }
   }
```

```
  }
 }
 rv=true;
 if(pw!=null)
  pw.close();
 return rv;
}
```

# Ergebnisse

**Abbildung 29.1. Beispielworkspace zur Demonstration einer einfachen Export-Komponente**




Die beschriebene einfache Komponente erzeugt für den Beispielworkspace in Abbildung 29.1, „Beispielworkspace zur Demonstration einer einfachen Export-Komponente“ eine Datei folgenden Inhalts:

```
Clock 0 de.netsysit.dataflowframework.modules.common.Clock
RandomSkalar 1 de.netsysit.dataflowframework.modules.generator.RandomSkalar
Sink 2 de.netsysit.dataflowframework.modules.common.Sink
Clock 0/counter -> RandomSkalar 1/produceRandomnumber
RandomSkalar 1/randomnumber -> Sink 2/printout
```

# Registrierung/Packaging

Damit ein neues Export-Format in der Anwendung zur Verfügung steht, muss man die benötigten Klassen in eine JAR-Datei verpacken und ihr im Unterverzeichnis `META-INF/services` eine Datei hinzufügen, die den Namen des implementierten Interfaces (voll qualifiziert - also mit vollständigem Package-Namen) trägt. Der Inhalt der Datei besteht aus dem ebenfalls voll qualifizierten Namen der das Interface implementierenden Klasse ( *Creating Extensible Applications* [https://docs.oracle.com/javase/tutorial/ext/basics/spi.html]).

# Weiterführende Informationen

Weiterführende Informationen sind in der API zum Thema zu finden.

# Kapitel 30. Graphische Programmierung für beliebige Komponenten

## Warum?

Die Anwendung wurde ursprünglich geschaffen, um durch die Verschaltung beliebiger JavaBeans Algorithmen zu modellieren, die gestellte Probleme lösen helfen sollten. Entwickler sollte die Möglichkeit geboten werden, sich nur auf die Funktionalität der einzelnen Module zu konzentrieren. Die Anwendung sollte sich um die umgebende Infrastruktur kümmern - beginnend bei der Instantiierung der Komponenten über die Versorgung mit Eingansdaten und den Abtransport der Ergebnisse bis hin zu grundlegenden Funktionalitäten. Dazu zählen unter anderem Parallelisiering, Priorisierung, Persistenz, Verteilung, Clustering,...

Es existieren verschiedene andere Umgebungen und Sprachen, die sich die graphische Programmierung auf die Fahnen geschrieben haben. Wie könnte man diese hier aus der Masse herausheben? Systeme zur graphischen Programmierung gehen oft den Weg, dass sie ein gestelltes Problem optimal lösen wollen. Dazu werden spezifische Komponentendefinitionen vereinbart, die in dieser Problemdomäne besonders nützlich sind. Diese Definitionen schränken aber zum einen den Entwickler der Komponenten ein und erschweren es bis zur Unmöglichkeit, das entstandene System auf andere Problemdomänen anzuwenden.

Die vorliegende Lösung entstand unter der zentralen Maßgabe, dem Entwickler keinerlei Vorschriften zu machen - er sollte idealerweise bestehende JavaBeans ungeändert als Komponenten im System einsetzen können. Diese Vorgabe wurde nicht ganz eingehalten (Modellierung der Moduleingänge) - aber selbst wenn: es wäre immer noch nötig, 'JavaBeans zu erstellen.

Wie wäre es nun aber, wenn man mit der vorliegenden Lösung graphische Modellierung betreiben könnte - egal, um welche Komponenten es sich handelte?

## Wie?

Komponenten bestehen aus diversen Teilen, die - unabhängig von der eingesetzten Technologie - immer wieder auftauchen. Die hier beschriebenen Komponenten beziehen sich auf die sogenannten Außenkanten der Komponenten. Darunter fallen alle die Eigenschaften, die die Komponente zur Interaktion mit dem umgebenden Kontext, anderen Komponenten und Anwendern exponiert.

Diese Eigenschaften und ihre Bedeutung können in einer formalen Spezifikation erfasst werden. Anschließend kann man beliebig viele technische Spezifikationen daraus ableiten. Für diese technischen Spezifikationen wiederum lassen sich Adapter gestalten, die es erlauben, Entities, die der technischen Spezifikation folgen, als Module in der vorliegenden Anwendung einzusetzen.

## Formale Spezifikation

### Metadaten

Die Metadaten enthalten unter anderem

| | |
|---|---|
| Name | Der Name der Komponente - dieser sollte über alle möglichen Komponenten eindeutig sein - daher könnte man etwa überlegen, hier Namespaces einzufügen. Diese Information wird nur vom Framework genutzt - daher kann hier durchaus ein technischer Name verwendet werden, der einem potentiellen Anwender nicht unbedingt etwas sagen muss.<br><br>obligatorisch, darf genau einmal auftreten |
| GUID | Ein global eindeutiger Identifier - dieser wird benötigt, um die Komponente eindeutig referenzieren zu können. Global eindeutig bedeutet hierbei, dass dieser Identifier innerhalb des Spezifikationsfragmentes nur genau einmal auftauchen darf.<br><br>obligatorisch, darf genau einmal auftreten |
| Displayname | Der Displayname wird dem Anwender präsentiert. Daher sollte er auf verständliche Art und Weise die Funktion der Komponente umreißen.<br><br>obligatorisch, darf genau einmal auftreten |
| Lokalisierter Displayname | Der lokalisierte Displayname macht die Benutzung für den Anwender einfacher - er beschreibt den Displaynamen der Komponente in der Sprache des Anwenders.<br><br>fakultativ, darf mehrfach auftreten, jedoch für jede Sprache höchstens einmal. Tritt diese Information für für eine Komponente und eine Sprache mehrfach auf, wählt die Anwendung eine aus, die dem Anwender präsentiert wird. Liegt die Information nicht in der Sprache des Anwenders vor, nutzt das System den Displaynamen zur Präsentation. |
| Beschreibung. | Die Beschreibung wird dem Anwender präsentiert. Diese darf mehr Text enthalten als der Displayname und sollte die Funktion der Komponente genauer erläutern.<br><br>fakultativ |
| Lokalisierte Beschreibung | Die lokalisierte Beschreibung macht die Benutzung für den Anwender einfacher - sie beschreibt die Funktion der Komponente in der Sprache des Anwenders.<br><br>fakultativ, darf mehrfach auftreten, jedoch für jede Sprache höchstens einmal. Tritt diese Information für für eine Komponente und eine Sprache mehrfach auf, wählt die Anwendung eine aus, die dem Anwender präsentiert wird. Liegt die Information nicht in der Sprache des Anwenders vor, nutzt das System die Beschreibung zur Präsentation, falls eine angegeben wurde. |
| Icon-URL | Das Icon bietet eine zusätzliche Möglichkeit zur Abgrenzung von Kopmponenten untereinander. Es sollte im Format PNG und idealerweise in einer Größe von 64x64 Pixeln vorliegen<br><br>fakultativ |

# Verbindungsendpunkte

Verbindungsendpunkte charakterisieren, welche Daten eine Komponente konsumiert und welche sie produziert. Jeder einzelne Verbindungsendpunkt hat einen Namen, eine Richtung und eine Bedeutung - Näheres siehe unten:

| | |
|---|---|
| Name | Der Name des Verbindungsendpunktes - dieser sollte innerhalb einer Komponente eindeutig sein. Der Name des Verbindungsendpunktes setzt zusammen mit dem Namen der Komponente die eindeutige Adresse zusammen, mittels derer die Verbindungen definiert werden. Diese Information wird nur vom Framework genutzt - daher kann hier durchaus ein technischer Name verwendet werden, der einem potentiellen Anwender nicht unbedingt etwas sagen muss.<br><br>obligatorisch, darf genau einmal auftreten |
| Displayname | Der Displayname wird dem Anwender präsentiert. Daher sollte er auf verständliche Art und Weise die Funktion des Verbindungsendpunktes darstellen.<br><br>obligatorisch, darf genau einmal auftreten |
| Lokalisierter Displayname | Der lokalisierte Displayname macht die Benutzung für den Anwender einfacher - er beschreibt den Displaynamen des Verbindungsendpunktes in der Sprache des Anwenders.<br><br>fakultativ, darf mehrfach auftreten, jedoch für jede Sprache höchstens einmal. Tritt diese Information für für einen Verbindungsendpunkt und eine Sprache mehrfach auf, wählt die Anwendung eine aus, die dem Anwender präsentiert wird. Liegt die Information nicht in der Sprache des Anwenders vor, nutzt das System den Displaynamen zur Präsentation. |
| Beschreibung. | Die Beschreibung wird dem Anwender präsentiert. Diese darf mehr Text enthalten als der Displayname und sollte die Funktion des Verbindungsendpunktes genauer erläutern.<br><br>fakultativ |
| Lokalisierte Beschreibung | Die lokalisierte Beschreibung macht die Benutzung für den Anwender einfacher - sie beschreibt die Funktion des Verbindungsendpunktes in der Sprache des Anwenders.<br><br>fakultativ, darf mehrfach auftreten, jedoch für jede Sprache höchstens einmal. Tritt diese Information für für einen Verbindungsendpunkt und eine Sprache mehrfach auf, wählt die Anwendung eine aus, die dem Anwender präsentiert wird. Liegt die Information nicht in der Sprache des Anwenders vor, nutzt das System die Beschreibung zur Präsentation, falls eine angegeben wurde. |
| Richtung | Die Richtung legt fest, ob dieser Verbindungsendpunkt Daten konsumiert (Eingang) oder produziert (Ausgang) oder beides. |

| | |
|---|---|
| | fakultativ - wenn nicht angegeben, wird angenommen, dass dieser Verbindungsendpunkt Daten sowohl konsumiert, wie auch produziert (bidirektionaler Modus). |
| Displayname (Eingang) | Falls ein Displayname für den Modus als Konsument (Eingang) angegeben wurde, wird dieser dem Anwender präsentiert, wenn es sich beim Verbindungsendpunkt um einen Eingang handelt.<br><br>fakultativ - falls nicht vorhanden, nutzt das System den Displaynamen zur Präsentation. |
| Lokalisierter Displayname (Eingang) | Falls ein lokalisierter Displayname für den Modus als Konsument (Eingang) angegeben wurde, wird dieser dem Anwender präsentiert, wenn es sich beim Verbindungsendpunkt um einen Eingang handelt.<br><br>fakultativ, darf mehrfach auftreten, jedoch für jede Sprache höchstens einmal. Tritt diese Information für für einen Verbindungsendpunkt und eine Sprache mehrfach auf, wählt die Anwendung eine aus, die dem Anwender präsentiert wird. Liegt die Information nicht in der Sprache des Anwenders vor, nutzt das System den lokalisierten Displaynamen zur Präsentation. |
| Displayname (Ausgang) | Falls ein Displayname für den Modus als Produzent (Ausgang) angegeben wurde, wird dieser dem Anwender präsentiert, wenn es sich beim Verbindungsendpunkt um einen Ausgang handelt.<br><br>fakultativ - falls nicht vorhanden, nutzt das System den Displaynamen zur Präsentation. |
| Lokalisierter Displayname (Eingang) | Falls ein lokalisierter Displayname für den Modus als Produzent (Ausgang) angegeben wurde, wird dieser dem Anwender präsentiert, wenn es sich beim Verbindungsendpunkt um einen Ausgang handelt.<br><br>fakultativ, darf mehrfach auftreten, jedoch für jede Sprache höchstens einmal. Tritt diese Information für für einen Verbindungsendpunkt und eine Sprache mehrfach auf, wählt die Anwendung eine aus, die dem Anwender präsentiert wird. Liegt die Information nicht in der Sprache des Anwenders vor, nutzt das System den lokalisierten Displaynamen zur Präsentation. |
| Beschreibung (Eingang) | Falls eine Beschreibung für den Modus als Konsument (Eingang) angegeben wurde, wird dieser dem Anwender präsentiert, wenn es sich beim Verbindungsendpunkt um einen Eingang handelt.<br><br>fakultativ - falls nicht vorhanden, nutzt das System die Beschreibung zur Präsentation. |
| Lokalisierte Beschreibung (Eingang) | Falls eine lokalisierter Beschreibung für den Modus als Konsument (Eingang) angegeben wurde, wird diese dem Anwender präsentiert, wenn es sich beim Verbindungsendpunkt um einen Eingang handelt.<br><br>fakultativ, darf mehrfach auftreten, jedoch für jede Sprache höchstens einmal. Tritt diese Information für für einen |

Verbindungsendpunkt und eine Sprache mehrfach auf, wählt die Anwendung eine aus, die dem Anwender präsentiert wird. Liegt die Information nicht in der Sprache des Anwenders vor, nutzt das System die lokalisierte Beschreibung zur Präsentation.

Beschreibung (Ausgang)

Falls eine Beschreibung für den Modus als Produzent (Ausgang) angegeben wurde, wird dieser dem Anwender präsentiert, wenn es sich beim Verbindungsendpunkt um einen Ausgang handelt.

fakultativ - falls nicht vorhanden, nutzt das System die Beschreibung zur Präsentation.

Lokalisierte Beschreibung (Eingang)

Falls ein lokalisierte Beschreibung für den Modus als Produzent (Ausgang) angegeben wurde, wird diese dem Anwender präsentiert, wenn es sich beim Verbindungsendpunkt um einen Ausgang handelt.

fakultativ, darf mehrfach auftreten, jedoch für jede Sprache höchstens einmal. Tritt diese Information für für einen Verbindungsendpunkt und eine Sprache mehrfach auf, wählt die Anwendung eine aus, die dem Anwender präsentiert wird. Liegt die Information nicht in der Sprache des Anwenders vor, nutzt das System die lokalisierte Beschreibung zur Präsentation.

Bedeutung

Diese Information kann vom System dazu verwendet werden, den Anwender bei der Erstellung von Verbindungen zu unterstützen: Anhand der Bedeutung eines Konsumenten und eines Produzenten kann das System mittels definierter Regeln erkennen, ob eine Verbindung zwischen beiden sinnvoll wäre. Falls nicht, kann das System entsprechende Hinweise geben oder die Installation der betreffenden Verbindung gleich ganz unterbinden.

fakultativ - darf maximal einmal auftreten

Multiplizität

Diese Angabe weist das System daraufhin, dass die Anzahl dieses Endpunktes (der ein Eingang sein muss) vom Anwender nach der Instantiierung erhöht werden darf, wenn der angegebene Wert (es sind nur ganzzahlige Werte erlaubt) größer als 0 ist. Bei Instantiierung werden genau so viele Exemplare dieses Einganges erzeugt, wie dieser numerische Wert angibt.

Vergleiche dazu auch Kapitel 10, *Variable Anzahl von Inputs* in Pogrammierhandbuch dWb+.

fakultativ - darf maximal einmal auftreten

# VisualComponentSpec

Spezifikation

Dieses Element der Spezifikation beschreibt ein visuelles Gestaltungselement, das die Komponente visualisiert. Der Inhalt ist stark davon abhängig, wie die technische Spezifikation aussieht. Es kann sich dabei zum Beispiel um ein Bild, eine Vektorgraphik oder ein HTML-Fragment handeln.

fakultativ, darf maximal einmal auftreten

# Properties

Properties charakterisieren Daten, die der Anwender für die jeweilige Komponente eisehen und ändern kann. Dabei kann es sich um Daten handeln, mit deren Hilfe es möglich ist, den internen Status der Komponente einzusehen. Weiterhin kann es zum Beispiel möglich sein, durch Änderung von Properties das Verhalten der Komponente zu ändern.

| | |
|---|---|
| Name | Der Name der Property - dieser sollte innerhalb einer Komponente eindeutig sein. Diese Information wird nur vom Framework genutzt - daher kann hier durchaus ein technischer Name verwendet werden, der einem potentiellen Anwender nicht unbedingt etwas sagen muss.<br><br>obligatorisch, darf genau einmal auftreten |
| Displayname | Der Displayname wird dem Anwender präsentiert. Daher sollte er auf verständliche Art und Weise die Funktion der Property darstellen.<br><br>obligatorisch, darf genau einmal auftreten |
| Lokalisierter Displayname | Der lokalisierte Displayname macht die Benutzung für den Anwender einfacher - er beschreibt den Displaynamen der Property in der Sprache des Anwenders.<br><br>fakultativ, darf mehrfach auftreten, jedoch für jede Sprache höchstens einmal. Tritt diese Information für für eine Property und eine Sprache mehrfach auf, wählt die Anwendung eine aus, die dem Anwender präsentiert wird. Liegt die Information nicht in der Sprache des Anwenders vor, nutzt das System den Displaynamen zur Präsentation. |
| Beschreibung. | Die Beschreibung wird dem Anwender präsentiert. Diese darf mehr Text enthalten als der Displayname und sollte die Funktion der Property genauer erläutern.<br><br>fakultativ |
| Lokalisierte Beschreibung | Die lokalisierte Beschreibung macht die Benutzung für den Anwender einfacher - sie beschreibt die Funktion der Property in der Sprache des Anwenders.<br><br>fakultativ, darf mehrfach auftreten, jedoch für jede Sprache höchstens einmal. Tritt diese Information für für eine Property und eine Sprache mehrfach auf, wählt die Anwendung eine aus, die dem Anwender präsentiert wird. Liegt die Information nicht in der Sprache des Anwenders vor, nutzt das System die Beschreibung zur Präsentation, falls eine angegeben wurde. |
| Richtung | Die Richtung legt fest, ob diese Property vom Endanwender gelesen und / oder geändert werden kann.<br><br>fakultativ - wenn nicht angegeben, wird angenommen, dass diese Property vom Anwender gelesen und geändert werden darf. |

Bedeutung

Diese Information kann vom System dazu verwendet werden, den Anwender mit passenden GUI-Elementen für die Inspaktion und Manipulation von Properties zu versorgen. Anhand der Bedeutung der Properties könnten zum Beispiel bei Zahlen Eingabefelder zur Verfügung gestellt werden, die nur numerische Eingaben erlauben.

fakultativ - darf maximal einmal auftreten

# Connections

Spezifikation

Dieses Element der Spezifikation beschreibt die Verbindungen der eventuell vorhandenen Kind-Komponenten untereinander und mit dieser Komponente. Sie enthält ein Array von Elementen, die wiederum jeweils ein Element namens Source und eines namens destination enthalten müssen.

Diese beiden beschreiben die sogenannten Connection-Endpunkte. Connection-Endpunkte haben exakt zwei Attribute wie folgt:

fakultativ - darf maximal einmal auftreten, aber beliebig viele Verbindungsspezifikationen enthalten

## Connection-Endpunkte

guid

Dieses Element der Spezifikation beschreibt das Widget, zwischen dem durch die Connection eine Verbindung mit einem anderen hergestellt werden soll. Der Wert dieses Attributes muss einem Wert eines Attributes namens GUID einer Komponente innerhalb des Spezifikationsfragments entsprechen.

obligatorisch, muss genau einmal auftreten

slotName

Dieses Element der Spezifikation beschreibt den Slot innerhalb der durch die guid addressierten Komponente, zwischen dem durch die Connection eine Verbindung mit einem anderen Slot einer anderen Komponente hergestellt werden soll. Der Wert dieses Attributes muss einem Wert eines Slotnamens der durch die zugehörige guid addressierten Komponente entsprechen.

obligatorisch, muss genau einmal auftreten

# Children

Spezifikation

Dieses Array enthält eine Menge von Spezifikationen von Komponenten. Dies dient dazu, hierarchisch verschachtelte (Baum-)Strukturen von Komponenten abbilden zu können. Dies entspricht dem Konzept der „Gruppe“.

Tatsächlich werden solche Spezifikationen in dWb+ als GroupWidgets abgebildet.

fakultativ - darf maximal einmal auftreten, aber beliebig viele Kinder enthalten

# Technische Spezifikation

Als Beispiel für die Umsetzung wurde eine JSON-Spezifikation geschaffen, für deren praktische Umsetzung im Folgenden zwei Spezifikationsfragmente präsentiert werden:

## Simples Beispiel

Um die Übersichtlichkeit zu wahren, wurde bei diesem simplen Beispiel nicht nur auf Properties verzichtet: Dieses erste Beispiel dient der vereinfachten Veranschaulichung und verzichtet daher auf eine hierarchische Gliederung und Verbindungen zwischen den einzelnen Komponenten:

```
{
 "name":"JSONSpecDemo",
 "displayName":"Titel dieses Moduls",
 "displayName_en":"Module Title",
 "shortDescription":"Kurze Beschreibung",
 "shortDescription_en":"short description",
 "iconURL":"http://flori.github.io/json/json_logo.png",
 "guid":"1",
 "slotDefinitions":
 [
  {
   "name":"slot1",
   "displayName":"Eingang 1",
   "displayName_en":"Input 1",
   "direction":"in",
   "type":"string"
  },
  {
   "name":"slot2",
   "direction":"out",
   "type":"bool",
   "shortDescription":"Kurze Beschreibung",
   "shortDescription_en":"short description"
  },
  {
   "name":"slot3",
   "type":"numeric",
   "displayNameInput":"Eingang 3",
   "displayNameInput_en":"Input 3",
   "displayNameOutput":"Ausgang 3",
   "displayNameOutput_en":"Output 3",
   "variableInputPortCount":"1"
  },
  {
   "name":"slot4",
   "direction":"out"
  },
  {
   "name":"slot5",
   "type":"Type1",
   "variableInputPortCount":"2"
```

```
  }
 ]
}
```

Die Beziehung der einzelnen Elemente des JSON-Objektes zur formalen Spezifikation sollten sich aus dem Namen ableiten. Zur Erklärung sei hinzugefügt, dass das Element namens "type" dem Element "Bedeutung" und das Element namens "variableInputPortCount" dem Element "Multiplizität" der formalen Beschreibung entspricht.

# Komplexes Beispiel einschließlich hierarchischer Gliederung

Dieses Beipiel veranschaulicht die Umsetzung der Möglichkeiten zur hierarchischen Gliederung und fügt einige Verbindungen zwischen den einzelnen Komponenten hinzu:

```
{
 "name":"JSONSpecDemoHierarchy",
 "displayName":"JSON Hierarchie",
 "displayName_en":"JSON hierarchy",
 "shortDescription":"Test hierarchischer JSON-Spezifikationen",
 "shortDescription_en":"Test of hierarchical JSON specifications",
 "guid":"1",
 "slotDefinitions":
 [
  {
   "name":"slot1",
   "displayName":"Eingang 1",
   "displayName_en":"Input 1",
   "direction":"in",
   "type":"string"
  },
  {
   "name":"slot2",
   "direction":"out",
   "type":"bool",
   "shortDescription":"Kurze Beschreibung",
   "shortDescription_en":"short description"
  },
  {
   "name":"slot3",
   "type":"numeric",
   "displayNameInput":"Eingang 3",
   "displayNameInput_en":"Input 3",
   "displayNameOutput":"Ausgang 3",
   "displayNameOutput_en":"Output 3"
  },
  {
   "name":"slot4",
   "direction":"out"
  },
  {
```

```
  "name":"slot5",
  "type":"Type1"
 }
],
"children":
[
 {
  "name":"JSONSpecDemoHierarchy",
  "displayName":"JSON Hierarchie",
  "displayName_en":"JSON hierarchy",
  "shortDescription":"Test hierarchischer JSON-Spezifikationen",
  "shortDescription_en":"Test of hierarchical JSON specifications",
  "guid":"11",
  "slotDefinitions":
  [
   {
    "name":"slot1",
    "displayName":"Eingang 1",
    "displayName_en":"Input 1",
    "direction":"in",
    "type":"string"
   },
   {
    "name":"slot2",
    "direction":"out",
    "type":"bool",
    "shortDescription":"Kurze Beschreibung",
    "shortDescription_en":"short description"
   },
   {
    "name":"slot3",
    "type":"numeric",
    "displayNameInput":"Eingang 3",
    "displayNameInput_en":"Input 3",
    "displayNameOutput":"Ausgang 3",
    "displayNameOutput_en":"Output 3"
   },
   {
    "name":"slot4",
    "direction":"out"
   },
   {
    "name":"slot5",
    "type":"Type1"
   }
  ],
  "children":
  [
   {
    "name":"ChildA",
    "displayName":"Kind 1",
    "displayName_en":"Child 1",
    "shortDescription":"Test von hierarchischen JSON-Komponenten",
    "shortDescription_en":"Testing hierarchical JSON components",
```

```
 "guid":"12",
 "slotDefinitions":
 [
  {
   "name":"slot1",
   "displayName":"Eingang 1",
   "displayName_en":"Input 1",
   "direction":"in",
   "type":"string"
  },
  {
   "name":"slot2",
   "direction":"out",
   "type":"bool",
   "shortDescription":"Kurze Beschreibung",
   "shortDescription_en":"short description"
  },
  {
   "name":"slot3",
   "type":"numeric",
   "displayNameInput":"Eingang 3",
   "displayNameInput_en":"Input 3",
   "displayNameOutput":"Ausgang 3",
   "displayNameOutput_en":"Output 3"
  },
  {
   "name":"slot4",
   "direction":"out"
  },
  {
   "name":"slot5",
   "type":"Type1"
  }
 ]
},
{
 "name":"ChildB",
 "displayName":"Kind 2",
 "displayName_en":"Child 2",
 "shortDescription":"Test von hierarchischen JSON-Komponenten",
 "shortDescription_en":"Testing hierarchical JSON components",
 "guid":"13",
 "slotDefinitions":
 [
  {
   "name":"slot1",
   "displayName":"Eingang 1",
   "displayName_en":"Input 1",
   "direction":"in",
   "type":"string"
  },
  {
   "name":"slot2",
   "direction":"out",
```

```
     "type":"bool",
     "shortDescription":"Kurze Beschreibung",
     "shortDescription_en":"short description"
    },
    {
     "name":"slot3",
     "type":"numeric",
     "displayNameInput":"Eingang 3",
     "displayNameInput_en":"Input 3",
     "displayNameOutput":"Ausgang 3",
     "displayNameOutput_en":"Output 3"
    },
    {
     "name":"slot4",
     "direction":"out"
    },
    {
     "name":"slot5",
     "type":"Type1"
    }
   ]
  }
],
"connections":
[
 {
  "source":
  {
   "guid":"11",
   "slotName":"slot5"
  },
  "destination":
  {
   "guid":"12",
   "slotName":"slot5"
  }
 },
 {
  "source":
  {
   "guid":"13",
   "slotName":"slot4"
  },
  "destination":
  {
   "guid":"12",
   "slotName":"slot1"
  }
 },
 {
  "source":
  {
   "guid":"13",
   "slotName":"slot2"
```

```
    },
    "destination":
    {
     "guid":"11",
     "slotName":"slot2"
    }
   },
  ]
},
{
 "name":"ChildA",
 "displayName":"Kind 1",
 "displayName_en":"Child 1",
 "shortDescription":"Test von hierarchischen JSON-Komponenten",
 "shortDescription_en":"Testing hierarchical JSON components",
 "guid":"2",
 "slotDefinitions":
 [
  {
   "name":"slot1",
   "displayName":"Eingang 1",
   "displayName_en":"Input 1",
   "direction":"in",
   "type":"string"
  },
  {
   "name":"slot2",
   "direction":"out",
   "type":"bool",
   "shortDescription":"Kurze Beschreibung",
   "shortDescription_en":"short description"
  },
  {
   "name":"slot3",
   "type":"numeric",
   "displayNameInput":"Eingang 3",
   "displayNameInput_en":"Input 3",
   "displayNameOutput":"Ausgang 3",
   "displayNameOutput_en":"Output 3"
  },
  {
   "name":"slot4",
   "direction":"out"
  },
  {
   "name":"slot5",
   "type":"Type1"
  }
 ]
},
{
 "name":"ChildB",
 "displayName":"Kind 2",
 "displayName_en":"Child 2",
```

```
  "shortDescription":"Test von hierarchischen JSON-Komponenten",
  "shortDescription_en":"Testing hierarchical JSON components",
  "guid":"3",
  "slotDefinitions":
  [
   {
    "name":"slot1",
    "displayName":"Eingang 1",
    "displayName_en":"Input 1",
    "direction":"in",
    "type":"string"
   },
   {
    "name":"slot2",
    "direction":"out",
    "type":"bool",
    "shortDescription":"Kurze Beschreibung",
    "shortDescription_en":"short description"
   },
   {
    "name":"slot3",
    "type":"numeric",
    "displayNameInput":"Eingang 3",
    "displayNameInput_en":"Input 3",
    "displayNameOutput":"Ausgang 3",
    "displayNameOutput_en":"Output 3"
   },
   {
    "name":"slot4",
    "direction":"out"
   },
   {
    "name":"slot5",
    "type":"Type1"
   }
  ]
 }
],
"connections":
[
 {
  "source":
  {
   "guid":"1",
   "slotName":"slot5"
  },
  "destination":
  {
   "guid":"2",
   "slotName":"slot5"
  }
 },
 {
  "source":
```

```
   {
    "guid":"3",
    "slotName":"slot4"
   },
   "destination":
   {
    "guid":"2",
    "slotName":"slot1"
   }
  },
  {
   "source":
   {
    "guid":"3",
    "slotName":"slot2"
   },
   "destination":
   {
    "guid":"1",
    "slotName":"slot2"
   }
  },
 ]
}
```

Die Integration beliebiger, der formalen folgenden, technischen Spezifikationen geschieht über das Parsen des jeweiligen Datenpakets und seine Übersetzung in eine Instanz vom Typ BeanInfo. Dazu muss der Entwickler eine Implementation des Interface IntrospectorImplementation der Klasse Introspector aus dem Namensraum de.elbosso.dataflowframework.logic zur Verfügung stellen. Die weiteren Abschnitte gehen auf eine solche examplarische Implementation ein, die es erlaubt, der weiter oben dargestellten technischen Spezifikation folgende Komponenten mit dWb+ zu verarbeiten.

# Introspector Implementation

Um die neue Implementierung des Interface im System bekanntzumachen, muss eine statische Methode der Klasse Introspector aus dem Namensraum de.elbosso.dataflowframework.logic aufgerufen werden:

```
public static void addSpecialIntrospector(Class cls,
      IntrospectorImplementation ii)
```

Damit wird, sobald für eine Instanz der angegebenen Klasse die BeanInfo erfragt wird, nicht mehr der Java-interne Mechanismus genutzt, sondern die angegebene Implementierung.

Die angegebene Klasse kapselt ein JSON-Objekt und ist in der Lage, ein solches in eine gültige String-Repräsentation zu serialisieren und ein solches aus einer Stringrepräsentation zu rekonstruieren:

```
import org.json.JSONException;
import org.json.JSONObject;

public class JsonCapsule extends java.lang.Object
{
```

```
 private JSONObject jo;

 public JsonCapsule(JSONObject jo)
 {
  super();
  if(jo==null)
   throw new IllegalArgumentException("null not allowed here!");
  this.jo = jo;
 }
 public JsonCapsule(String encoded) throws JSONException
 {
  super();
  if(encoded==null)
   throw new IllegalArgumentException("null not allowed here!");
  this.jo = new JSONObject(encoded);
 }

 @Override
 public String toString()
 {
  return jo.optString("name");
 }

 public String getEncoded()
 {
  return jo.toString();
 }

 public void methodStub(java.lang.Object input,int index)
 {

 }
}
```

# Metadaten

Die Metadaten einer Komponente (Beschreibung, Name) werden über eine angepasste Implementierung der Klasse BeanDescriptor realisiert:

```
import java.util.Locale;
import java.beans.BeanDescriptor;

private static class BeanDescriptor extends BeanDescriptor
{
 private JsonCapsule jsonCapsule;
 public BeanDescriptor(JsonCapsule jsonCapsule)
 {
  super(JsonCapsule.class);
  this.jsonCapsule=jsonCapsule;
 }

 @Override
```

```
 public String getName()
 {
  return jsonCapsule.jo.optString("name");
 }

 @Override
 public String getDisplayName()
 {
  String rv=getName();
  String key="displayName_"+Locale.getDefault().getLanguage();
  if(jsonCapsule.jo.has(key))
   rv=jsonCapsule.jo.optString(key);
  else if(jsonCapsule.jo.has("displayName"))
   rv=jsonCapsule.jo.optString("displayName");
  return rv;
 }

}
```

Das Icon wird direkt über die angepasste BeanInfo geliefert, über die ebenfalls die Instanz der BeanDescription zugreifbar ist - dieser Teil der Implementation ist hier dargestellt:

```
import java.beans.BeanInfo;
import java.beans.PropertyDescriptor;
import java.beans.MethodDescriptor;
import java.awt.Image;
import java.net.URL;
import javax.imageio.ImageIO;

class BeanInfo extends Object implements BeanInfo
{
 private JsonCapsule jsonCapsule;
 private BeanDescriptor beanDescriptor;
 private PropertyDescriptor[] propertyDescriptors;
 private MethodDescriptor[] methodDescriptors;
 private Image icon;

 private BeanInfo(JsonCapsule jsonCapsule)
 {
  super();
  this.jsonCapsule=jsonCapsule;
 }

 public java.beans.BeanDescriptor getBeanDescriptor()
 {
  if(beanDescriptor==null)
   beanDescriptor=new BeanDescriptor(jsonCapsule);
  return beanDescriptor;
 }

 public Image getIcon(int iconKind)
 {
```

```
  Image rv=null;
  if(iconKind==ICON_COLOR_32x32)
  {
   if(icon==null)
   {
    try
    {
     URL url=null;
     if(jsonCapsule.jo.has("iconURL"))
      new URL(jsonCapsule.jo.optString("iconURL"));
     icon=ImageIO.read(url);
    }
    catch(Throwable t)
    {
     t.printStackTrace();
     icon=null;
    }
   }
   rv=icon;
  }
  return rv;
 }
}
```

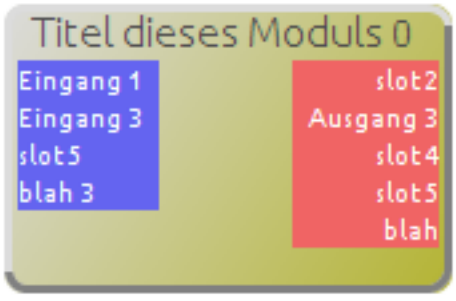
## Verbindungsendpunkte

Verbindungsendpunkte werden implementiert, indem für jeden Eingang ein entsprechender MethodDescriptor konstruiert wird, die in ihrer Gesamtheit über die angepasste BeanInfo zugreifbar gemacht werden.

Für jeden Ausgang wird ein PropertyDescriptor konstruiert. Auch diese werden über die angepasste BeanInfo zugreifbar gemacht.

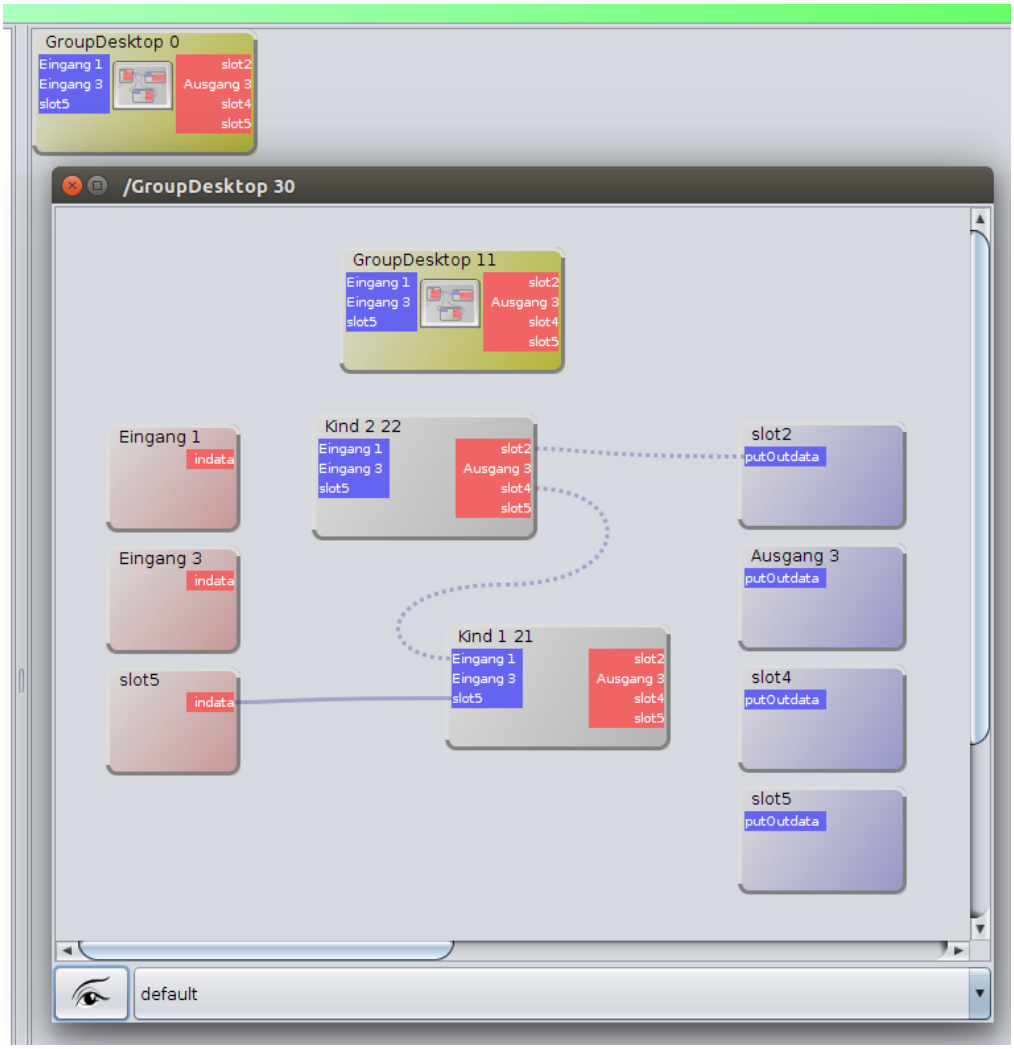
# Ergebnisse

**Abbildung 30.1. Beispiel der Komponente aus dem oben angegebenen simplen Fragment der Umsetzung der technischen Spezifikation**




Das oben angegebene simple JSON-Fragment erzeugt in dWb+ die in Abbildung 30.1, „Beispiel der Komponente aus dem oben angegebenen simplen Fragment der Umsetzung der technischen Spezifikation“ dargestellte Komponente.

**Abbildung 30.2. Beispiel der Komponente aus dem oben angegebenen simplen Fragment der Umsetzung der technischen Spezifikation**




Das oben angegebene komplexe JSON-Fragment erzeugt in dWb+ ein Group-Widget mit enthaltenem Group-Workspace, die beide in Abbildung 30.2, „Beispiel der Komponente aus dem oben angegebenen simplen Fragment der Umsetzung der technischen Spezifikation“ dargestellt sind.

# Weiterführende Informationen

Weiterführende Informationen sind in der exemplarischen Implementierung der JSON-Spezifikation zu finden. Diese wird als Plugin für dWb+ zur Verfügung gestellt. Der Quellcode kann beim Autor dieses Dokuments angefordert werden.

# Kapitel 31. Service Provider Infrastructure

## Einleitung

Die Anwendung kann durch Nutzung der Möglichkeiten der Java Service Provider Infrastructure (SPI) um neue Aspekte erweitert werden. Dazu ist keine Änderung am Code der Anwendung nötig: Es existieren Interfaces, die man implementiert. Wenn man die Implementierung dann nach den Bestimmungen des SPI in eine JAR-Datei verpackt und sie in das Unterverzeichnis `lib` des Konfigurationsverzeichnisses legt, wird diese Implementierung beim nächsten Neustart der Anwendung vom System registriert und benutzt.

Zum Konfigurationsverzeichnis der Anwendung vergleiche bitte Anhang A, *Verzeichnis-Layout* im Anwenderhandbuch dWb+.

Zur Erstellung der Jar-Dateien kann man zum Beispiel Maven-Projekte benutzen - Anregungen dazu findet man im Anhang C, *Erstellen von Komponenten für dWb+ mittels Maven* im Pogrammierhandbuch dWb+

# Module

## Überblick

Die Anwendung bietet die Möglichkeiten, die Funktionalität durch verschiedene SPI-Implementierung zu bereichern. Unter anderem existiert die Möglichkeit, sich in den Lebenszyklus von Modulen einzuhängen. Diese Tatsache wird durch das entsprechende Interface widergespiegelt.

## Interface

```
package de.elbosso.ui.moduleworkspace;

import de.netsysit.ui.moduleworkspace.ModuleWidget;

public interface SpecialCapabilitiesHandler<T extends ModuleWidget>
{
 public void unregister(T toremove);
 public void register(T widget);
}
```

Mit einer entsprechenden Implementierung dieses Interface ist es möglich, auf das Hinzufügen und Entfernen jedes Moduls zu oder aus einem Workspace zu reagieren. Damit kann die Funktionalität eines Workspace sehr stark erweitert werden, ohne die Anwendung selbst anpassen zu müssen.

## Beispielimplementierung

Eine beispielhafte Implementierung des im vorhergehenden Abschnitt vorgestellten Interface ist hier dargestellt.

```
import de.elbosso.ui.moduleworkspace.SpecialCapabilitiesHandler;
import java.util.List;
import java.util.LinkedList;
import de.netsysit.dataflowframework.ui.ModuleWidget;
import javax.swing.AbstractAction;
import java.awt.event.ActionEvent;

public class DummyService extends Object implements
  SpecialCapabilitiesHandler
{
 private List<String> l=new LinkedList();
 public void unregister(ModuleWidget toremove)
 {
  if(l.contains(toremove.getUniqueId()))
  {
   System.out.println("unregistering "+toremove.getHeading());
   l.remove(toremove.getUniqueId());
  }
  else
   System.out.println("Error - unregistered "
    +toremove.getHeading()+" error!");
 };
 public void register(ModuleWidget widget)
 {
  System.out.println("registering "+widget.getHeading());
  l.add(widget.getUniqueId());
  widget.addAdditionalAction(new AbstractAction("additionalAction")
  {
   public void actionPerformed(ActionEvent e)
   {
    throw new UnsupportedOperationException("Not supported yet.");
   }
  });
 }
}
```

Diese Implementierung tut nicht mehr, als Meldungen auf der Standardausgabe anzuzeigen, wenn Module zu einem Workspace hinzugefügt oder daraus entfernt werden. Weiterhin fügt die Implementierung jedem Modul eine weitere Custom-Action hinzu, die hier aber keine Auswirkungen hat. (zu Custom-Actions vergleiche bitte Kapitel 16, *Actions zur Steuerung eines Moduls* im Pogrammierhandbuch dWb+).

# Verbindungen

## Überblick

Die Anwendung bietet die Möglichkeiten, die Funktionalität durch verschiedene SPI-Implementierung zu bereichern. Unter anderem existiert die Möglichkeit, sich in den Lebenszyklus von Verbindungen einzuhängen. Diese Tatsache wird durch das entsprechende Interface widergespiegelt.

## Interface

```
package de.elbosso.ui.moduleworkspace;

import de.netsysit.ui.moduleworkspace.Link;
import de.netsysit.ui.moduleworkspace.ModuleWorkspace;

public interface LinkLifeCycleHandler
 <T extends Link, U extends ModuleWorkspace>
{
 public void linkAdded(T link,U moduleWorkspace);
 public void linkRemoved(T link,U moduleWorkspace);
}
```

Mit einer entsprechenden Implementierung dieses Interface ist es möglich, auf das Hinzufügen und Entfernen jeder Verbindung zu oder aus einem Workspace zu reagieren. Damit kann die Funktionalität eines Workspace sehr stark erweitert werden, ohne die Anwendung selbst anpassen zu müssen.

# Beispielimplementierung

Eine beispielhafte Implementierung des im vorhergehenden Abschnitt vorgestellten Interface ist hier dargestellt.

```
import de.netsysit.ui.moduleworkspace.Link;
import de.netsysit.ui.moduleworkspace.ModuleWorkspace;
import java.beans.PropertyChangeEvent;
import de.elbosso.ui.moduleworkspace.LinkLifeCycleHandler;
import java.beans.PropertyChangeListener;

public class ErrorWarnOkService extends Object
  implements LinkLifeCycleHandler
  ,PropertyChangeListener
{

 public void linkAdded(Link t, ModuleWorkspace u)
 {
  t.addPropertyChangeListener(Link.ACTIVITYSTATE, this);
 }

 public void linkRemoved(Link t, ModuleWorkspace u)
 {
  t.removePropertyChangeListener(this);
 }

 public void propertyChange(PropertyChangeEvent evt)
 {
  boolean cond=((Boolean)evt.getNewValue()).booleanValue();
  ((Link)evt.getSource()).setWarning(cond);
 }

}
```

Diese Implementierung tut nicht mehr, als an jedem neu hinzugefügten Link einen PropertyChangeListener zu registrieren, der auf Änderungen der Property reagiert, die anzeigt, ob der

jeweilige Link aktiv ist. Die Implementierung dekoriert dann jeden Link, der aktiv ist, mit einem Warnungs-Emblem.

# Kapitel 32. Modul-Wrapper

## Einleitung

Generell ist es hilfreich, sich an dieser Stelle nochmals einen der grundlegenden Eckpunkte von dWb+ ins Gedächtnis zu rufen: Es gibt einen ModuleWorkspace, in dem ModuleWidgets leben können. Diese ModuleWidgets dienen als Vermittler zwischen Modulen (plain old JavaBeans, Skriptfragmente,...) und dem ModuleWorkspace: Kein Modul kann direkt im ModuleWorkspace existieren. Eine anschauliche Metapher dafür wäre ein Taucher- oder - besser noch - Raumanzug: Der ModuleWorkspace (das Weltall) kann von Modulen (Astronauten odr Kosmonauten) nur dan betreten werden, wenn sie von einem ModuleWidget (Raumanzug) vollständig umhüllt sind.

Diese einfache Hierarchie kann man noch weiter denken - etwa wenn man als Erklärung die Aspekt-orientierte Programmierung heranzieht: Dabei geht es darum, unterschiedlichsten Konzepten (Klassen, Methoden, Objekten,...) gleichartige Funktionalität hinzuzufügen, ohne dafür deren Quelltexte ändern zu müssen.

Diese Analogie lässt sich vielleicht besser mit einem konkreten Anwendungsfall besser erklären: Module versenden Nachrichten, sobald sich ihr innerer Zustand ändert - diese Zustandsänderungen kommen aufgrund von Stimuli zustande. Das sind in den allermeisten Fällen entweder Nachrichten, die die Module von anderen empfangen, Daten, die über beliebige Schnittstellen empfangen werden oder einfach zeitgesteuerte Ereignisse. Wollte man mit dWb+ eine State-Machine umsetzen, könnte man die ModuleWidgets als States begreifen. Dann müsste man aber dafür sorgen, dass die Module nur dann Nachrichten versenden, wenn sie den aktuellen State verkörpern. Eine andere - ähnliche - Analogie wären Petrinetze.

Dies könnte man umsetzen, indem man die Module um einen Ausgang und einen Eingang für einen neuen abstrakten Typ namens Token ergänzt. Module dürfen nur dann Nachrichten senden, wenn sie ein Token "haben". Token dürfen nicht dupliziert werden - Wann immer ein Modul ein Token versendet, muss es dieses aus seinem internen Zustand löschen. Jeder Stimulus, der zu einer Änderung des internen Zustandes führt, muss in diesem Szenario außerdem zu einer Auswertung führen, an deren Ende die Entscheidung steht, ob das Token abgegeben werden soll (muss).

Man könnte des weiteren darüber nachdenken, die Anzahl eingehender und ausgehender Verbindungen auf 1 zu beschränken wie etwa in „Maximale Anzahl von Verbindungen“ in Kapitel 2, *BeanInfo-Verwendung in dWb+* im Pogrammierhandbuch dWb+ beschrieben.

Dies alles umzusetzen wäre eine extrem umfangreiche und fehlerträchtige Arbeit - die Alternative, diese Erweiterungen in der Klasse ModuleWidget direkt einzubauen ist ebenso unsinnig - schließlich würden alle Anwender, die diese Form der Umsetzung einer State-Machine nicht brauchen nur durch die zusätzlichen "technischen" Slots verwirrt. Man muss also eine andere Idee entwickeln...

## Die Idee

Die Idee besteht darin, einfach einen Wrapper um die ModuleWidgets zu legen. Damit können die ModuleWidgets sich um ihre Aufgabe der Anpassung der Module an den ModuleWorkspace kümmern und müssen nicht geändert werden. Benötigt man weitere Funktionalitäten ähnlich derer, die im vorhergehenden Abschnitt beschrieben wurden, fügt man die ModuleWidgets nicht mehr dirket zum ModuleWorkspace hinzu, sondern umhüllt diese wiederum mit speziellen ModuleWidgets - den ModuleWidgetWrappern. Aus Sicht der ModuleWidgetWrapper erscheinen die ModuleWidgets wie Module. Aus Sicht der ModuleWidgets sind die ModuleWidgetWrapper völlig transparent. Aus der Sicht

des ModuleWorkspace besteht kein Unterschied zwischen ModuleWidgets und ModuleWidgetWrappern. Module wissen nichts von ModuleWidgetWrappern.

# Anforderungen an ModuleWidgetWrapper

## GUI-Integration

Die GUI des ModuleWidgets muss mit eigenen GUI-Elementen in den Parameter-Dialog so integriert werden, dass beide leicht durch den Anwender erreicht und bedient werden können. Die GUI des ModuleWidget muss sowohl von der Gestaltung und Layout als auch von der Bedienung her der version ohne ModuleWidgetWrapper völlig gleichen.

## Link-Management

Der ModuleWidgetWrapper ist so zu gestalten, dass er Informationen über das Hinzufügen und Entfernen von Links (Verbindungen zwischen Modulen) erhält.

## Persistenz

Der ModuleWidgetWrapper muss in der Lage sein, Konfigurationsdaten zu lesen und zu schreiben. Diese Funktionalitäten sind so zu implementieren, dass sie an den passenden Stellen im umfassenden Workflow der Persistierung von ModuleWorkspaces angesprochen werden.

## Message-Management

Der ModuleWidgetWrapper muss das Interface `PropertyChangeProxy.Visitor` aus dem Namensraum `de.netsysit.dataflowframework.logic` implementieren - so ist sichergestellt, das die Übertragung von Botschaften zwischen Modulen sowohl inbound als auch outbound abhängig vom internen Status des Moduls gesteuert werden kann.

Alternativ dazu besteht die Möglichkeiot, eingehende und ausgehende Nachrichten zu steuern, indem man die Methode `doCommunicate` überlädt - dadurch kann man eingehende Daten kontrollieren - und die Methode `handlePropertyChangeFromModule` - dadurch werden ausgehende Daten kontrolliert.

Beide genannten Methoden kan man so gestalten, dass eingehende Daten gespeichert werden (zum Beispiel in einer Map) bis eine bestimmte Bedingung an einem der Eingänge eintritt und erst dann die gespeicherten Daten an das umhüllte Modul weiterleitet, beziehungsweise ausgehende Daten so lange puffert (auch hier gegebenenfalls in einer Map), bis eine bestimmte Bedingung eintritt, woraufhin dann die gepufferten Daten versendet werden.

# Die Einbindung

Es wurde ein Interface `ModuleWrapperCreatorService` im Namensraum `de.elbosso.dataflowframework.logic.services` geschaffen, der lediglich eine Methode definiert: Diese Methode bekommt drei Parameter übergeben, darunter ein ModuleWidget. Jede Implementierung entscheidet nun anhand der übergebenen Parameter, ob für das übergebene ModulWidget ein ModuleWidgetWrapper erzeugt werden soll. Die Implementierung kann auch entscheiden, dass bestimmte ModuleWidgets keine Wrapper erhalten sollen - daher ist als Typ des Rückgabewerts die Klasse `ModuleWidget` aus dem Namensraum `de.netsysit.dataflowframework.ui` festgelegt.

Implementierungen werden zur Laufzeit über die Java-spezifische Service Provider Infrastructure (SPI) gesucht und geladen. In einer VM darf jeweils nur eine Implementierung gleichzeitig registriert sein.

# Die Basisklasse

Zur Implementierung spezifischer ModuleWidgetWrapper wurde eine abstrakte Basisklasse geschrieben, die die notwendigen Methoden und Funktionalitäten zur Umsetzung der vorgenannten Anforderungen bereits enthält oder zumindest deklariert. Die Basisklasse trägt den Namen `ModuleWidgetWrapper` und residiert im Namensraum `de.elbosso.dataflowframework.ui`. Abgeleitete Klassen zur Implementierung spezifischer Verhaltensweisen müssen lediglich noch folgende Methoden implementieren:

# Abstrakte Methoden, die überschrieben werden müssen

## constructWrapperStuff

```
protected java.awt.Component constructWrapperStuff();
```

Diese Methode sollte alle GUI-Komponenten erzeugen, die zur Konfiguration des Wrappers benötigt werden. Damit kann man erreichen, dass gewisse Aspekte des Wrappers spezifisch für einzelne Module angepasst werden können. Die Anzahl der Komponenten ist prinzipiell nicht beschränkt - allerdings muss derjenige, der die Methode implementiert dafür sorgen, dass alle Komponenten durch den Anwender über die zurückzugebende `Component` erreichbar sind. Falls nicht alle Komponenten in einem Layout Platz finden muss also ein geeigneter Container gewählt (zum Beispiel `JTabbedPane`) und dieser gegebenenfalls mit Code zur Navigation ausgestattet werden.

## preparePropsForSerialization

```
protected void preparePropsForSerialization();
```

Es besteht die Möglichkeit, Konfigurationsparameter des Wrappers zu serialisieren. Damit erreicht man eine persistente Konfiguration der Wrapper auf Modulbasis. Der dazu genutzte Mechanismus ist der bereits bestehende zur Speicherung beliebiger Daten am Modul. Vor der Serialisierung einer `ModuleDescription` wird für jedes Modul, das von eimen Wrapper verwaltet wird, diese Methode aufgerufen, in der dann die zu persistierenden Eigenschaften des Wrappers als Properties am Modul gespeichert werden. Der Zugriff auf die Properties *muss* mittels der Methode `getProps()` erfolgen. Es wird dringend geraten, als Schlüssel Bezeichner zu wählen, die darauf hinweisen, dass diese Properties zum `ModuleWrapper` gehören.

**Beispiel 32.1. Beispiel für die Persistierung eines Konfigurationsparameters für einen `ModuleWrapper`**

```
getProps().setProperty("ModuleWidgetWrapper."+
    this.getClass().getName()+
    "maxDataBeforeGivingTokenAway",
    java.lang.Integer.toString(maxDataBeforeGivingTokenAway));
```

## cleanupPropsForSerialization

```
protected void cleanupPropsForSerialization();
```

Da zur Serialisierung von Wrapper-Parametern der bestehende Mechanismus zur Speicherung beliebiger Daten am Modul benutzt wird könnte es zu Irritation beim Anwender kommen, wenn dieser bei der

Bearbeitung dieser Daten per GUI die Schlüssel des ModuleWrappers zu Gesicht bekommt. Daher wird die Methode `preparePropsForSerialization` direkt vor der Serialisierung aufgerufen, um die Daten zu schreiben und die Methode `cleanupPropsForSerialization` direkt nach der Serialisierung um die Schlüssel wieder zu entfernen. Der Zugriff auf die Properties *muss* mittels der Methode `getProps()` erfolgen.

**Beispiel 32.2. Beispiel für das Löschen eines Konfigurationsparameters für einen `ModuleWrapper` nach der Serialisierung**

```
getProps().remove("ModuleWidgetWrapper."+
    this.getClass().getName()+
    "maxDataBeforeGivingTokenAway");
```

## dataCameThrough

```
  protected void dataCameThrough();
```

Diese Methode wird im Interface `PropertyChangeProxy.Visitor` im Namensraum `de.netsysit.dataflowframework.logic` definiert. Sie wird immer dann aufgerufen, wenn ein Datum an einem Slot in einem `ModuleWidget` ankommt. Im `ModuleWrapper` kann darüber gesteuert werden, ob die Datenübertragung im Modul tatsächlich etwas bewirkt.

Diese Methode gilt als *deprecated* - das Management ein- und ausgehender Daten sollte in einem Wrapper über `doCommunicate` und `handlePropertyChangeFromModule` erfolgen.

# Methoden der Basisklasse, die überschrieben werden sollten

## Konstruktor

Im Konstruktor kann man dafür sorgen, neue Slots zum umhüllten Modul hinzuzufügen. Das Beispiel zeigt dieses Vorgehen für einen `boolean` Input- und einen String Output-Slot.

**Beispiel 32.3. Beispiel für einen Konstruktor für `ModuleWrapper`**

```
java.lang.reflect.Method m =
    ExampleModuleWidgetWrapper.class.getMethod("wrapperInput", boolean.class);
java.beans.MethodDescriptor md = new java.beans.MethodDescriptor(m);
de.netsysit.dataflowframework.ui.beans.BeanMethodInputSlot bmis =
    new de.netsysit.dataflowframework.ui.beans.BeanMethodInputSlot(md);
inputPanel.addSlot(new VariableSlotDescription(md.getName(),md));
ConnectionEndPointDescription cepd=new ConnectionEndPointDescription();
cepd.setPortName("wrapperOutput");
cepd.setTypename(java.lang.String.class.getName());
cepd.setWidgetId(getWidgetId());
cepd.setWidgetUniqueId(getUniqueId());
cepd.setWidgetName(getHeading());
outputPanel.addSlot(cepd);
```

## doCommunicate

```
  public void doCommunicate(java.lang.Object dataItem,
                            java.lang.String destname);
```

Diese Methode wird aufgerufen, um Daten an das umhüllte Modul zu senden. Das Beispiel zeigt, dass die eingehenden Daten in einer Map zwischengespeichert werden.

**Beispiel 32.4. Beispiel das Management am `ModuleWrapper` eingehender Daten**

```
public void doCommunicate( Object dataItem, String destname) throws Exception
{
    if (dataItem != null)
    {
        inputLatch.put(destname, dataItem);
        nullLatch.remove(destname);
    }
    else
    {
        nullLatch.add(destname);
        inputLatch.remove(destname);
    }
}
```

Ein Signal am im Konstruktor erzeugten Input-Slot sorgt für die Weitergabe der gespeicherten Daten an das umhüllte Modul:

```
public void wrapperInput(boolean value)
{
    try
    {
        for (java.lang.String key : inputLatch.keySet())
        {
           super.doCommunicate(inputLatch.get(key), key);
        }
        for (java.lang.String key : nullLatch)
        {
           super.doCommunicate(null, key);
        }
        inputLatch.clear();
        nullLatch.clear();
    } catch (java.lang.Exception exp)
    {
        error(EXCEPTION_LOGGER, exp);
    }
}
```

## handlePropertyChangeFromModule

```
  public void handlePropertyChangeFromModule(java.beans.PropertyChangeEvent evt);
```

Diese Methode aufgerufen, wenn das Modul ein Ausgabedatum versenden möchte. Im Beispiel werden die ausgehenden Daten zwischengespeichert und ihre Weiterleitung unterdrückt, sofern die Bedingung nicht erfüllt ist.

### Beispiel 32.5. Beispiel für das Versenden eines Ausgabedatums durch einen `ModuleWrapper`

```
public void handlePropertyChangeFromModule(PropertyChangeEvent evt)
{
    if(condition==false)
    {
        outputLatch.put(evt.getPropertyName(),evt);
    }
    else
        super.handlePropertyChangeFromModule(evt);
}
```

Ein Signal am im Konstruktor erzeugten Input-Slot sorgt für die Weitergabe der gespeicherten Daten an die angekoppelten Empfänger:

```
public void wrapperInput(boolean value)
{
    for (java.lang.String key : outputLatch.keySet())
    {
        super.handlePropertyChangeFromModule(outputLatch.get(key));
    }
    outputLatch.clear();
}
```

## setProps

```
public void setProps(de.elbosso.model.table.PropertiesTable props);
```

Diese Methode wird nach erfolgreicher Deserialisierung aufgerufen. Hier kann der `ModuleWrapper` Informationen über mittels `preparePropsForSerialization` gespeicherte Konfigurationsparameter entnehmen. Es wird empfohlen, für den ModuleWrapper spezifische Schlüssel anschließend aus den Props zu löschen.

**Beispiel 32.6. Beispiel für das Übernehmen eines Konfigurationsparameters für einen `ModuleWrapper` nach der Deserialisierung**

```
if(props!=null)
{
    PropertiesTable pt = new PropertiesTable();
    pt.putAll(props);
    if (pt.containsKey("ModuleWidgetWrapper." + this.getClass().getName() + "maxDa
    {
        try
        {
            maxDataBeforeGivingTokenAway = java.lang.Integer.parseInt(pt.getProper
                    "ModuleWidgetWrapper." +
                    this.getClass().getName() +
                    "maxDataBeforeGivingTokenAway"));
            if(maxDataBeforeGivingTokenAwaytf!=null)
            maxDataBeforeGivingTokenAwaytf.setText(
                    java.lang.Integer.toString(maxDataBeforeGivingTokenAway));
            maxDataBeforeGivingTokenAwaySet();
        }
        finally
        {
            getProps().remove("ModuleWidgetWrapper." +
                    this.getClass().getName() +
                    "maxDataBeforeGivingTokenAway");
        }
    }
    super.setProps(pt);
}
else
super.setProps(props);
```

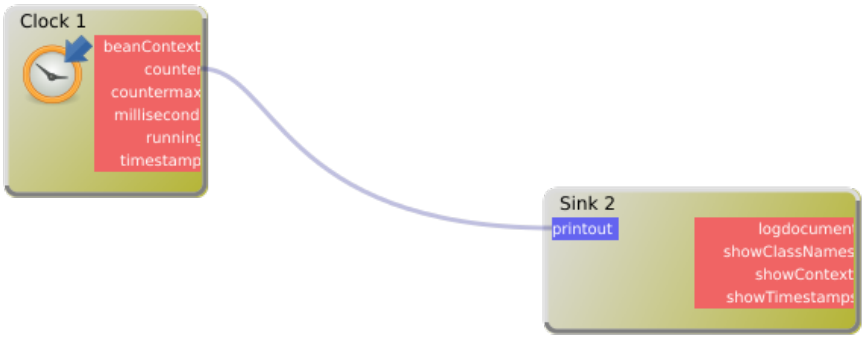
# Kapitel 33. Unter der Haube

## Kommunikation

### Grundlegendes

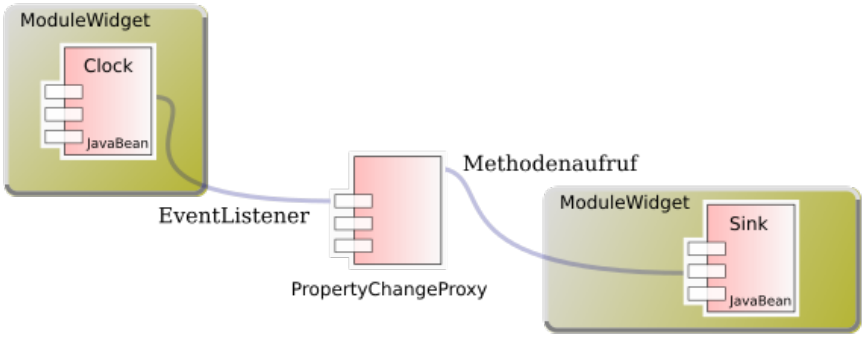
Dieser Abschnitt soll eine Beschreibung dessen geben, was unter der Haube vor sich geht, wenn der Anwender einen Datenfluss von einem Modul zu einem anderen definiert. Wir bemühen dabei das einfachste mögliche Beispiel: Ein Taktgeber erzeugt eine Folge von Zahlen und eine Konsole zeigt diese Folge an. Wir benutzen dazu die Module Clock (Taktgeber) und Sink (Konsole) wie in Abbildung 33.1, „Beispielworkspace zur Demonstration der Kopplung zweier Module“ dargestellt.

**Abbildung 33.1. Beispielworkspace zur Demonstration der Kopplung zweier Module**




Sieht man sich diesen Workspace etwas genauer unter Berücksichtigung der eingesetzten Komponenten an, so ergibt sich das in Abbildung 33.2, „Innere Abläufe bei der Kopplung zweier Module“ dargestellte Bild: Die Module auf dem Workspace sind Wrapper um die eigentliche Funktionalität, die von JavaBeans erbracht wird. Die Verbindung zwischen ihnen wird durch eine Instanz der Klasse PropertyChangeProxy erbracht. Diese lauscht an der JavaBean, die als Sender konfiguriert ist (linke Seite der Verbindung) auf PropertyChangeEvents. Tritt ein solcher auf, entpackt diese Instanz den neuen Wert (PropertyChangeEvents enthalten immer den alten und den neuen Wert der jeweiligen Property).

**Abbildung 33.2. Innere Abläufe bei der Kopplung zweier Module**




Nunmehr wird überprüft, ob für diese Verbindung ein Skript zur Datentransformation definiert wurde. Ist dies der Fall, setzt die PropertyChangeProxy-Instanz den soeben extrahierten neuen Wert der Property unter dem Namen _data_ im Ausführungskontext des Skripts. Anschließend wird das Skript ausgeführt. Möchte es die Daten tatsächlich ändern, muss das Skript die Variable _data_ setzen. Anschließend liest die PropertyChangeProxy-Instanz den Wert der Variablen _data_ aus dem Ausführungskontext des Skriptes und fährt wie folgt fort:

Die PropertyChangeProxy-Instanz ruft die Methode an der Empfänger-JavaBean (rechte Seite der Verbindung) auf, die dem empfangenden Slot entspricht. Beim Aufruf dieser Methode wird der aus dem PropertyChangeEvent entnommene neue (und gegebenenfalls durch das Skript angepasste) Wert als Parameter der Methode übergeben. Damit ist die Aufgabe des PropertyChangeProxy beendet. In Abbildung 33.3, „Innere Abläufe bei der Kopplung zweier Module als UseCase-Diagramm“ ist dieser Mechanismus nochmals als UseCase-Diagramm dargestellt, Abbildung 33.4, „Innere Abläufe bei der Kopplung zweier Module als Sequenzdiagramm“ zeigt ihn als Sequenzdiagramm.

**Abbildung 33.3. Innere Abläufe bei der Kopplung zweier Module als UseCase-Diagramm**

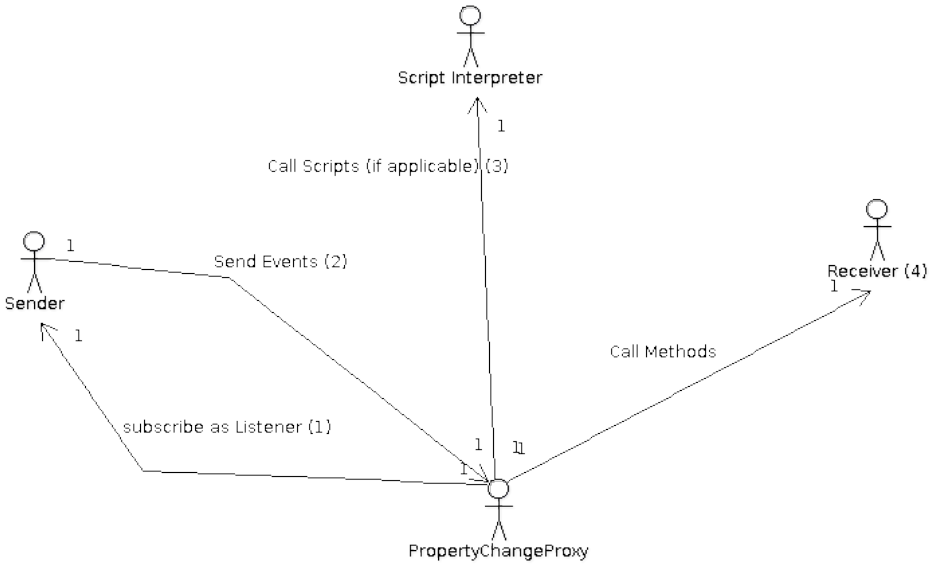
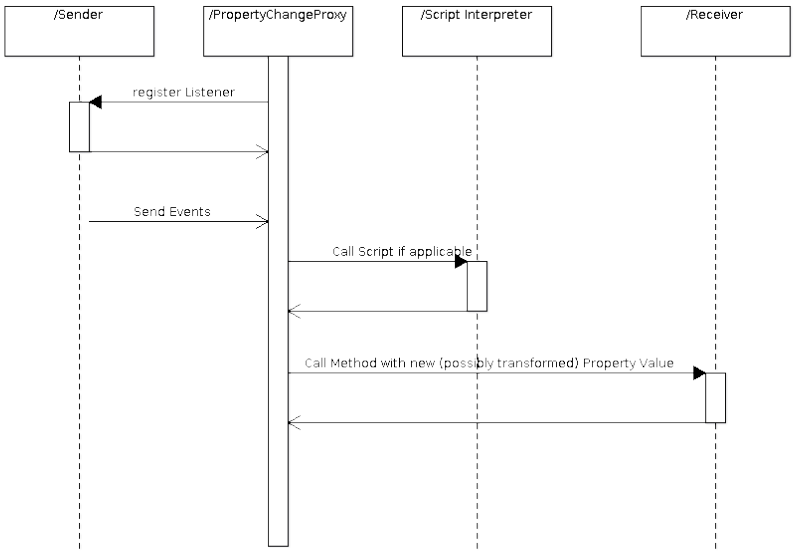



**Abbildung 33.4. Innere Abläufe bei der Kopplung zweier Module als Sequenzdiagramm**



# Context (Mandantenfähigkeit)

## Szenario

### Allgemein

Beim Stichwort Mandantenfähigkeit geht es darum, dass es in datenflussorientierten Umgebungen durchaus so sein kann, dass neben den eigentlichen Daten noch Informationen zu ihrer Entstehung für die Verarbeitungseinheiten weiter hinten in der Verarbeitungskette interessant sein könnten.

Es soll hier zunächst an einigen Beispielen gezeigt werden, wofür eine solche Mandantenfähigkeit nützlich sein kann.

### Szenarien

#### Szenario A - Serverdienst

In diesem ersten Szenario geht es um die Umsetzung eines einfachen HTTP-Servers, der basierend auf der abgeforderten URL eine Datenverarbeitung anstößt und das Ergebnis der Datenverarbeitung an den Anfragenden zurückliefern soll. Dabei stelle man sich den Workflow so vor, dass es einen BeanContextService gibt, der die HTTP-Protokollunterstützung liefert. Auf dem Arbeitstisch werden zwei Module abgelegt: HTTP-Sender und HTTP-Receiver. Das Receiver-Modul gibt die empfangene URL an das nächste Modul weiter. zwischen Sender und Receiver sind beliebig viele andere Module geschaltet, die entsprechend der URL verschiedene Arbeitsschritte durchführen, bis wieder ein String entsteht, der über den Sender an den Client zurückübertragen werden soll.

Da aber HTTP ein Protokoll ist, bei dem sich quasi gleichzeitig beliebig viele Clients mit einem Server verbinden können, existiert im beschriebenen Szenario ein Problem: Wenn sich während der Bearbeitung eines Requests bereits vier neue Clients mit dem Server verbinden, weiß das Sender-Modul am Ende nicht mehr, welchem Client es die Antwort zusenden soll. Das könnte man verhindern, indem man 503 (Service Unavailable) an den Client meldet, solange die Bearbeitung des letzten Requestes nicht abgeschlossen ist. Eine solche Lösung wäre arnselig und nicht mehr zeitgemäß.

Wenn man es aber schaffen könnte, mit den eigentlichen Nutzdaten einen Datenspeicher zu verknüpfen, der Daten zum Kontext der Operation enthält, der die Datenverarbeitung angestoßen hat, und diese Information über viele verschiedene Komponenten oder Verarbeitungseinheiten hinweg konsistent bliebe, könnte man im Receiver-Modul die Kennung des Client in den Kontext schreiben. Dieser würde bei jeder Kommunikation zweier Module weitergegeben und das Sender-Modul müsste lediglich in den Kontext schauen, um zu wissen, welchem Client es die in Rede stehende Antwortbotschaft übermitteln soll.

#### Szenario B - Synchronisation

Das zweite Beispiel ist die Synchronisation, die sich in wenigen Worten wie folgt zusammenfassen lässt: Wenn ein Modul zwei Daten erzeugt und diese über verschiedene Kanäle an unterschiedliche nachfolgende Verarbeitungsketten - also ein oder mehrere gekoppelte Verarbeitungseinheiten - zur Weiterverarbeitung übergibt, ist das ein normales Szenario, das keine besonderen Vorbereitungen bedarf.

Sollen aber die Ergebnisse der beiden Verarbeitungseinheiten am Ende miteinander verschmolzen werden, muss man wissen, welches Ergebnis aus Verarbeitungskette A mit welchem Ergebnis aus der Kette B kombiniert werden soll. Timestamps zur Kennzeichnung der Ergebnisse des ersten Modul fallen aus, da keine zwei Ereignisse in Rechnersystemen wirklich gleichzeitig geschehen. Man muss an die Daten Marker anfügen. Das letzte Modul könnte dann die Marker vergleichen und herausfinden, welche zwei seiner Inputs miteinander kombiniert werden müssten.

Auch hier ist es wieder so, dass diese Marker über die gesamte Kette der Verarbeitungseinheiten konsistent erhalten bleiben müssten.

#### Szenario C - Transaktionen

Das dritte Beispiel sind Transaktionen, die sich in datenflussgetriebenen Systemen mittels Mandantenfähigkeit wie folgt umsetzen ließen: In diesem Szenario sind die Verarbeitungen in den einzelnen Modulen einer Verarbeitungskette vergleichbar den atomaren Operationen in einer durch eine Transaktion gekapselten Operation. Das erste Atom eröffnet die Transaktion. Jedes weitere Glied in der Kette muss über die geöffnete Transaktion informiert sein, denn im Fehlerfall muss diese zurückgerollt werden. Das letzte Modul in der Kette muss schließlich dafür sorgen, die Transaktion "zu committen" - also die Ergebnisse persistent zu machen. Auch dazu muss es auf Wissen über die laufende Transaktion zugreifen können.

Dieses Szenario ist ebenfalls ein gutes Beispiel für die Nützlichkeit des Konzeptes der Mandantenfähigkeit oder eines globalen Kontextes:

Diesmal muss die Information über die Transaktion über die gesamte Kette der Verarbeitungseinheiten konsistent erhalten bleiben.

# Implementierung

## Vorüberlegungen

Es reicht in den beschriebenen Szenarien nicht, einfach einen Marker oder Kontextinformationen an das erzeugte Datum zu hängen: in der Verarbeitungskette werden in jeder Verarbeitungseinheit typischerweise neue Daten aus den Eingangsdaten erzeugt - Wenn man Kontextinformationen nur an die eigentlichen Nutzdaten anhängt, sind diese spätestens nach der nächsten Verarbeitungseinheit verloren, wenn diese neue Daten aus denursprünglichen Nutzdaten erzeugt.

Man muss also darüber hinaus sicherstellen, dass die Kontextinformationen, die in einer Verarbeitungseinheit mit den Input-Daten ankommen, an die Resultate angehängt werden. Die Kontextdaten müssen also entlang der Verarbeitungskette von Nutzdatum zu Nutzdatum übertragen werden. Das erinnert ein wenig an den guten alten Ponyexpress: Die Pakete und Depeschen stellen hier die Kontextinformationen dar und die Ponys, die an jeder Station getauscht werden, die Nutzdaten, die immer nur zwischen zwei Verarbeitungseinheiten gültig sind.

Wie setzt man diese Anforderungen nun um? Eine - triviale - Möglichkeit wäre die Forderung, dass keine "normalen" Daten mehr durch das System fließen dürfen, sondern Tupel: bestehend aus den eigentlichen Nutzdaten und dem Kontext. Dann müssten aber alle Module über dieses Konzept Bescheid wissen und bestehende Module dahingehend angepasst werden.

Diese Lösung würde auch bedeuten, dass der Entwickler eines Modules dafür Sorge tragen müsste, die Kontextinformationen von den Eingangsdaten in die Ausgangsdaten zu kopieren. dWb+ war aber gerade mit dem Anspruch angetreten, die Schwelle für den Einstieg in die datenflussgetriebene Programmierung dadurch besonders niedrig zu halten, dass keine Framework-Klempnerei zu betreiben ist, sondern sich der Entwickler einfach nur auf die Umsetzung der fachlichen Anforderungen konzentrieren kann.

Es wäre also wünschenswert, wenn der Entwickler eines solchen Moduls überhaupt nichts davon bemerken würde, dass Kontextinformationen fließen und alle bestehenden Module einfach wie bisher weiter funktionieren würden.

## Voraussetzungen

Die genannten Anforderungen wurden durch die vorliegende Implementation gelöst. Es sei aber an dieser Stelle darauf hingewiesen, dass die hier gemachten Aussagen aktuell lediglich innerhalb einer virtuellen Maschine gelten: bei der Datenübertragung von und zu Remoting Modulen gehen die Kontextinformationen im aktuellen Zustand verloren.

Um den Kontext nutzen zu können, ist es zwingend notwendig, Module von den Basisklassen BeanContextChildModuleBase oder ThreadingBeanContextChildModuleBase mittelbar oder unmittelbar abzuleiten. Später wird eventuell eine Möglichkeit geschaffen, auch mit Modulen umgehen zu können, die nicht von diesen beiden Basisklassen abgeleitet wurden. Falls innerhalb einer Verarbeitungskette Module existieren, die nicht von einer der beiden genannten Klassen abgeleitet wurden, kann es zum Verlust der Kontextinformationen kommen.

### Ablauf

Der Sender muss - möchte er mit den Daten, die er anschließend versendet, bestimmte Kontextinformationen verknüpfen - diese Kontextinformationen dem ContextManagerService mitteilen. Anschließend versendet er die Informationen durch Aufruf einer der send-Methoden. Innerhalb dieser Methoden wird nachgesehen, ob im ContextManager-Service für diese Bean Kontextinformationen hinterlegt wurden. ist das der Fall, werden diese wiederum über den ContextManager-Service mit dem zu versendenden Event verknüpft und die Verknüpfung mit der Bean gelöst.

Nachdem der PropertyChangeProxy einen PropertyChangeEvent empfangen hat, fragt er den zentralen ContextManager-Service nach eventuell mit diesem Event verknüpften Kontextinformationen. Existieren solche, werden diese Informationen mit dem Empfänger verknüpft und die Verknüpfung mit dem Event wird gelöst. Anschließend wird die Bearbeitung des Events genauso fortgeführt, wie weiter oben beschrieben.

Eine JavaBean, die nicht an den Kontextinformationen interessiert ist, kann unverändert weiterarbeiten: Die send-Methode, die zum Weiterreichen der Ergebnisse aufgerufen wird, schaut nach, welche Kontext-Informationen aktuell an der JavaBean hängen und propagiert diese mittels der Implementierung einfach an den oder die nächsten PropertyChangeProxy-Instanzen.

Ein Modul, das Informationen aus dem Kontext für seine Arbeit benötigt, kann einfach den ContextManager-Service aus dem BeanContext benutzen, um die Informationen abzufragen, die aktuell für es vorliegen und anschließend mit ihnen arbeiten.

Module, die eventuell vorhandene Kontextinformationen nicht benötigen, können diese neue Funktionalität ignorieren und bestehende Module, die vor deren Einführung erstellt wurden, können einfach ungeändert weiter genutzt werden: Kontextinformationen werden automatisch von den Eingangsdaten auf die Ausgangsdaten übertragen und so durch die bestehende Verarbeitungskette propagiert.

Module, die neue Informationen in den Kontext eintragen möchten oder Informationen aus dem Kontext auslesen müssen, können dies über den entsprechenden Context-MAnager-Service des BeanContext tun.

# Basisklassen

Wie bereits verschiedentlich in diesem Handbuch erwähnt, bringt dWb+ diverse Basisklassen mit, die bei der Erstellung von Modulen helfen sollen. Der Anwender kann auch eigene JavaBeans als Module benutzen oder sogar Skripts als Module einsetzen. Allerdings machen die Basisklassen Neuentwicklungen einfacher und prinzipiell sind kompilierte, echte Java-Klassen natürlich auch performanter als Scripts.

Aus allen diesen Gründen werden in diesem Kapitel nochmals alle zur Verfügung stehenden Basisklassen mit ihren besonderen Vorteilen uns speziellen Einsatzzwecken zusammengefasst.

# ModuleBase

Diese Klasse ist die einfachste der zur Verfügung stehenden Basisklassen. Sie bieten nichts weiter, als einen Mechanismus, um ein zentralisieres Fehler- und Warnungssystem umzusetzen und eine einfachere

Weise, die PropertyChangeEvents zu managen: sie hat bereits die Methoden zum Registrieren und Deregistrieren der Listener an Bord und verschiedene Convenience-Methoden zum Versenden von PropertyChangeEvents.

# ResetableModuleBase

Diese Klasse ist eine direkte Ableitung von `ModuleBase`: Sie kann CustomActions anbieten und tut das mit einer Action zum Zurücksetzen - der Anwender muss hier lediglich noch die Methode implementieren, die nach Auslösen dieser Action ausgeführt wird.

# BeanContextChildModuleBase

Diese Klasse macht genau das, was ihr Name sagt: Sie ist dafür vorbereitet, mit Diensten, die ein `BeanContext` anbietet, zu interagieren. Sie wird direkt von `Modulebase` abgeleitet und bietet damit alle dort bereits genannten Annehmlichkeiten. Darüber hinaus kümmert sie sich auch um die Verwaltung der Listener für VetoablePropertyChangeEvents. Weiterhin existieren vier Methoden, mit denen der Anwender arbeiten kann, um Dienste aus dem BeanContext in eigenen Modulen zu nutzen: im einzelnen sind das Methoden, die beim Announcement eines neuen Service und bei der Revocation eines Service aufgerufen werden und die Methoden, die beim Hinzufügen der Instanz zum BeanContext und beim Entfernen der Instanz aus dem BeanContext aufgerufen werden.

Erst diese Klasse (und alle abgeleiteten Klassen) kann auf Informationen des Contexts zugreifen und diese auch aktiv ändern.

Alle von dieser Klasse abgeleiteten Klassen zeigen Informationen über geringfügige (Warnungen ) oder schwerwiegende (Fehler ) Probleme durch entsprechende Dekorationen am Modul selbst an.

Kinder dieser Klasse bieten dem Anwender die Möglichkeit, Symbols und Embleme zur Laufzeit am Modul anzuzeigen. Diese Symbole können jeweils mit einem Tooltip versehen werden. Es ist möglich, diese Dekorationen zur Laufzeit wieder zu entfernen. Damit besteht eine effiziente Möglichkeit, den Anwender über den inneren Zustand des Moduls zu informieren. Die benutzten Symbole sollten nicht mehr als 24 Pixel Kantenlänge aufweisen und nur in Ausnahmefällen nicht quadratisch sein.

# CommunicationTemplate

Diese Klasse ist eine direkte Ableitung von BeanContextChildModuleBase. Sie kommt dort zum Einsatz, wo ein Modul mit einer bestimmten Ressource arbeiten muss und die Verbindugn durch den Benutzer initiiert werden soll. Ein Beispiel dafür wäre die Verbindung zu einem TCP-Server: Der Anwender hat nur noch zwei Methoden auszufüllen: die, die die Ressource allokiert und die, die sie wieder freigibt.

# VariableNumberofInputsVisualization

Diese Klasse ist eine direkte Ableitung von `BeanContextChildModuleBase`. Sie dient dazu, Anwender einfach in die Lage zu versetzen, Module zu schaffen, die eine variable Anzahl Eingänge des gleichen Typs besitzen und die Werte an den einzelnen Eingängen visualisieren sollen. Anwender müssen dafür in abgeleiteten Klassen lediglich Methoden implementieren, die das Hinzufügen einer Komponente (neuer Input-Slot wurde erzeugt) und das Eingehen eines neuen Datums an einem der Input-Slots realisieren.

# ThreadingModuleBase

Diese Klasse ist eine direkte Ableitung von `ModuleBase`. Sie dient dazu, Module zu unterstützen, deren Implementierung etwas mehr Rechenzeit benötigt: Alle anderen direkt oder indirekt von

ModuleBase abgeleiteten Module arbeiten im Event Dispatch Thread. Das ist aber nur gut, wenn die Verarbeitungsleistung klein ist und schnell abgearbeitet werden kann. Bei längeren Arbeiten ist es gut, wenn nur der Start der Abarbeitung im Event Dispatch Thread angestartet wird, die eigentliche Abarbeitung aber nebenläufig in einem anderen Thread durchgeführt wird. Genau dazu ist diese Basisklasse da. Der Anwender muss lediglich zwei Methoden erstellen: zum Einen ist das die Methode, mit der das sogenannte CubbyHole - die Kopplung zwischen Event Dispatch Thread und dem Hintergrund-Thread hergestellt wird - erzeugt wird und zum anderen die, die die eigentliche Arbeit leistet. Zu beachten ist noch, dass dem Hintergrund-Thread die neue Workload mittels des Aufrufes doWork() signalisiert werden muss - das bedeutet, dass diese Methode in Methoden aufgerufen werden muss, die Input-Slots darstellen.

# ThreadingBeanContextChildModuleBase

Diese Klasse macht genau das, was ihr Name sagt: Sie ist dafür vorbereitet, mit Diensten, die ein `BeanContext` anbietet, zu interagieren. Sie wird direkt von `ThreadingModuleBase` abgeleitet und bietet damit alle dort bereits genannten Annehmlichkeiten. Darüber hinaus kümmert sie sich auch um die Verwaltung der Listener für VetoablePropertyChangeEvents. Weiterhin existieren vier Methoden, mit denen der Anwender arbeiten kann, um Dienste aus dem BeanContext in eigenen Modulen zu nutzen: im einzelnen sind das Methoden, die beim Announcement eines neuen Service und bei der Revocation eines Service aufgerufen werden und die Methoden, die beim Hinzufügen der Instanz zum BeanContext und beim Entfernen der Instanz aus dem BeanContext aufgerufen werden.

Erst diese Klasse (und alle abgeleiteten Klassen) kann auf Informationen des Contexts zugreifen und diese auch aktiv ändern.

# ThreadingResetableModuleBase

Diese Klasse ist eine direkte Ableitung von `ThreadingBeanContextChildModuleBase`: Sie kann CustomActions anbieten und tut das mit einer Action zum Zurücksetzen - der Anwender muss hier lediglich noch die Methode implementieren, die nach Auslösen dieser Action ausgeführt wird.

# ThreadingCommunicationTemplate

Diese Klasse ist eine direkte Ableitung von `ThreadingBeanContextChildModuleBase`. Sie kommt dort zum Einsatz, wo ein Modul mit einer bestimmten Ressource arbeiten muss und die Verbindugn durch den Benutzer initiiert werden soll. Ein Beispiel dafür wäre die Verbindung zu einem TCP-Server: Der Anwender hat nur noch zwei Methoden auszufüllen: die, die die Ressource allokiert und die, die sie wieder freigibt.

# HostPortCommunicationTemplate

Diese Klasse ist eine direkte Ableitung von `ThreadingCommunicationTemplate`. Sie wurde lediglich um zwei Read/Write-Properties ergänzt: eine für den Hostnamen und einen für die Portnummer.

# StartStopModule

Diese Klasse ist eine direkte Ableitung von `ThreadingBeanContextChildModuleBase`. Sie dient als Basisklasse für Module, die zur Erzeugung ihrer Outputs keiner externen Stimuli bedürfen - sobald sie gestartet werden, produzieren sie Outputs so schnell es möglich ist. Daher existieren hier zwei Actions, um dem Anwender die Möglichkeit zu geben, die Produktion auf Wunsch unterbrechen zu können.

## StartStopModuleWithDoOnce

Diese Klasse ist eine direkte Ableitung von `StartStopModule`. Sie erweitert die Basisklasse um die Möglichkeit, die Erzeugung der Ausgaben noch feiner zu steuern: Dazu wird eine weitere Action hinzugefügt, deren Ausführung das jeweilige Modul dazu veranlasst, exakt ein Datum zu produzieren.

## JaxbBase

Diese Klasse ist eine direkte Ableitung von `BeanContextChildModuleBase`. Sie kann als Grundlage für Module dienen, die Daten von/nach XML de/serialisieren müssen.

## MapMessageModule

Diese Klasse ist eine direkte Ableitung von ThreadingBeanContextChildModuleBase. Sie kann als Grundlage für Module dienen, die verschiedene Daten nicht über mehrere Ausgabe- oder Eingabeslots versenden und empfangen wollen, sondern alle jeweils in eine `Map` verpackt. Diese Klasse stellt die Infrastruktur zum Versenden und Empfangen der `Map`, wie auch der Extraktion und des Verpackens von Daten aus/in der Map. Weiterhin bietet diese Klasse Möglichkeiten, zwischen Schlüsseln in der `Map` zu vermitteln: ist ein Wert in der Eingangsmap unter einem anderen Schlüssel abgelegt, als dies das empfangende Modul erwartet, bietet die Basisklasse eine entsprechende GUI um ein entsprechendes Mapping festzulegen.

## RemoteModule

Diese Klasse ist eine direkte Ableitung von ThreadingBeanContextChildModuleBase. Sie kann als Grundlage für Module dienen, die per Remoting auf andere Rechner verteilt werden sollen.

# Anhang A. Quellcode

## Ein einfaches Modul

```
import de.netsysit.dataflowframework.modules.ModuleBase;

public class CharacterCounter extends ModuleBase
{
 private String lastInput;

 public void input(String in)
 {
  lastInput=in;
  countCharacters();
 }

 private int characterCount;

 public int getCharacterCount()
 {
  return characterCount;
 }

 private boolean ignoreSpaces;

 public boolean isIgnoreSpaces()
 {
  return ignoreSpaces;
 }

 public void setIgnoreSpaces(boolean ignoreSpaces)
 {
  boolean old=isIgnoreSpaces();
  this.ignoreSpaces = ignoreSpaces;
  send("ignoreSpaces",old,isIgnoreSpaces());
 }

 private void countCharacters()
 {
  int old=getCharacterCount();
  //Algorithmus-Implementierung hier
  send("characterCount",old,getCharacterCount());
 }

}
```

# Eine JMX MBean als Modul

## Interface

```
public interface JMXBeanMXBean
{
 public double getThreshold();
 public void setThreshold(double threshold);
}
```

## Modul

```
import de.netsysit.dataflowframework.modules.ModuleBase;
import javax.management.AttributeChangeNotification;
import javax.management.ListenerNotFoundException;
import javax.management.MBeanNotificationInfo;
import javax.management.Notification;
import javax.management.NotificationBroadcasterSupport;
import javax.management.NotificationEmitter;
import javax.management.NotificationFilter;
import javax.management.NotificationListener;

public class JMXBean extends ModuleBase implements JMXBeanMXBean
  , NotificationEmitter
{

 private double threshold;

 public double getThreshold()
 {
  return threshold;
 }

 public void setThreshold(double threshold)
 {
  double old = getThreshold();
  this.threshold = threshold;
  send("threshold", old, getThreshold());
 }
 private long sequenceNumber = 1;
 private NotificationBroadcasterSupport notificationBroadcasterSupport;

 public JMXBean()
 {
  super();
  String[] types = new String[]
  {
   javax.management.AttributeChangeNotification.ATTRIBUTE_CHANGE
  };
```

```
 String name = AttributeChangeNotification.class.getName();
 String description = "Threshold violated";
 MBeanNotificationInfo info = new MBeanNotificationInfo(types,
 name, description);
 notificationBroadcasterSupport
   = new NotificationBroadcasterSupport(new MBeanNotificationInfo[]
     {
      info
   });
}

public void removeNotificationListener(NotificationListener listener,
  NotificationFilter filter, Object handback) throws ListenerNotFoundException
{
 notificationBroadcasterSupport.removeNotificationListener(listener,
   filter, handback);
}

public void addNotificationListener(NotificationListener listener,
  NotificationFilter filter, Object handback) throws IllegalArgumentException
{
 notificationBroadcasterSupport.addNotificationListener(listener,
   filter, handback);
}

public void removeNotificationListener(NotificationListener listener)
  throws ListenerNotFoundException
{
 notificationBroadcasterSupport.removeNotificationListener(listener);
}

public MBeanNotificationInfo[] getNotificationInfo()
{
 return notificationBroadcasterSupport.getNotificationInfo();
}

private boolean state;

public boolean isState()
{
 return state;
}

public void input(Number in)
{
 boolean old = isState();
 state = in.doubleValue() < threshold;
 send("state", old, isState());
 if (state)
 {
  Notification n = new AttributeChangeNotification(this,
    sequenceNumber++, System.currentTimeMillis(), "State changed",
    "State", "boolean", old, isState());
  notificationBroadcasterSupport.sendNotification(n);
```

```
  }
 }
}
```

# Ein Modul mit Algorithmusbearbeitung im eigenen Thread

```
import de.netsysit.dataflowframework.modules.ThreadingModuleBase;
import de.netsysit.util.threads.CubbyHole;
import de.netsysit.util.threads.SimpleBufferingCubbyHole;

public class ThreadingCharacterCounter extends ThreadingModuleBase
{
 public ThreadingCharacterCounter()
 {
  super(ThreadingCharacterCounter.class.getName());
 }


 private String lastInput;

 public void input(String in)
 {
  setLastInput(in);
  processData(in);
 }

 private synchronized String getLastInput()
 {
  return lastInput;
 }

 private synchronized void setLastInput(String lastInput)
 {
  this.lastInput = lastInput;
 }

 private int characterCount;

 public synchronized int getCharacterCount()
 {
  return characterCount;
 }

 private synchronized void setCharacterCount(int characterCount)
 {
  this.characterCount = characterCount;
 }

 private boolean ignoreSpaces;
```

```
 public synchronized boolean isIgnoreSpaces()
 {
  return ignoreSpaces;
 }

 public synchronized void setIgnoreSpaces(boolean ignoreSpaces)
 {
  boolean old=isIgnoreSpaces();
  this.ignoreSpaces = ignoreSpaces;
  send("ignoreSpaces",old,isIgnoreSpaces());
 }

 @Override
 protected CubbyHole createCubbyHole()
 {
  return new SimpleBufferingCubbyHole();
 }

 @Override
 protected void doWork(Object ref) throws InterruptedException
 {
  int old=getCharacterCount();
  java.lang.String data=getLastInput();
  int cc=0;
  if(isIgnoreSpaces())
  {
   //Algorithmus-Implementierung ohne Leerzeichen hier
  }
  else
  {
   cc=data.length();
  }
  setCharacterCount(cc);
  send("characterCount",old,getCharacterCount());
 }

}
```

# Parallele Verarbeitung innerhalb eines Moduls

```
import de.netsysit.dataflowframework.modules.BeanContextChildModuleBase;
import de.netsysit.util.beans.context.service.BackgroundExecutor;
import java.beans.beancontext.BeanContextServiceAvailableEvent;
import java.beans.beancontext.BeanContextServiceRevokedEvent;
import de.elbosso.util.threads.Workload;
import de.elbosso.util.threads.ParallelSequentialManager;

public class ParallelSequential extends BeanContextChildModuleBase
{
 private BackgroundExecutor executor;
```

```
private ParallelSequentialManager psm;

public void serviceAvailable(BeanContextServiceAvailableEvent bcsae)
{
 super.serviceAvailable(bcsae);
 if(executor==null)
 {
  if (bcsae.getServiceClass()== BackgroundExecutor.class)
  {
   try
   {
    executor =
      (BackgroundExecutor) (bcsae.getSourceAsBeanContextServices()).getService(
      this, this, BackgroundExecutor.class, this, this);
   }
   catch (Exception e)
   {
    executor = null;
   }
   if(executor!=null)
    psm=new ParallelSequentialManager(executor);
  }
 }
}
@Override
public void serviceRevoked(BeanContextServiceRevokedEvent bcsre)
{
 super.serviceRevoked(bcsre);
 if(executor!=null)
 {
  if(bcsre.getServiceClass()==BackgroundExecutor.class)
  {
   executor=null;
  }
 }
}
private static int runningWorkloadNumber;

class MyWorkload extends Workload
{
 private int sleep;
 private int id=runningWorkloadNumber++;

 MyWorkload(int input)
 {
  super();
  sleep=input;
 }
 public void run()
 {
  send("msg",null,id+" sleeping for "+sleep+"ms");
  try
  {
   java.lang.Thread.currentThread().sleep(sleep);
```

```
   }
   catch (InterruptedException ex)
   {
    ex.printStackTrace();
   }
   send("msg",null,id+" awoke!");
  }
  public Runnable createSequential()
  {
   return new Runnable(){
    public void run()
    {
     send("msg",null,("sequential "+id));
    }
   };
  }
 }
 public void input(Number input)
 {
  if((psm!=null)&&(input!=null))
  {
   psm.enqueue(new MyWorkload(input.intValue()));
  }
 }
 private String msg;
 public String getMsg()
 {
  return msg;
 }
}
```

# Benutzung eines Dienstes aus dem BeanContext

```
import de.netsysit.dataflowframework.modules.BeanContextChildModuleBase;
import de.netsysit.util.beans.context.service.Notification;
import java.beans.PropertyVetoException;
import java.beans.beancontext.BeanContext;
import java.beans.beancontext.BeanContextServices;
import java.beans.beancontext.BeanContextServiceAvailableEvent;
import java.beans.beancontext.BeanContextServiceRevokedEvent;

/**
 *
 * @author elbosso
 */
public class BeanContextAwareCharacterCounter extends
 BeanContextChildModuleBase
{
 private Notification notificationService;
```

```
private String lastInput;

public void input(String in)
{
 lastInput=in;
 if(lastInput!=null)
  countCharacters();
 else
 {
  if(notificationService!=null)
   notificationService.notifyInfo(
    this.getClass().getSimpleName(), "NULL input empfangen!");
 }
}

private int characterCount;

public int getCharacterCount()
{
 return characterCount;
}

private boolean ignoreSpaces;

public boolean isIgnoreSpaces()
{
 return ignoreSpaces;
}

public void setIgnoreSpaces(boolean ignoreSpaces)
{
 boolean old=isIgnoreSpaces();
 this.ignoreSpaces = ignoreSpaces;
 send("ignoreSpaces",old,isIgnoreSpaces());
}

private void countCharacters()
{
 int old=getCharacterCount();
 //Algorithmus-Implementierung hier
 send("characterCount",old,getCharacterCount());
}


@Override
public void serviceAvailable(BeanContextServiceAvailableEvent bcsae)
{
 super.serviceAvailable(bcsae);
 if(notificationService==null)
 {
  if(bcsae.getServiceClass()==
   de.netsysit.util.beans.context.service.Notification.class)
  try
```

```
   {
    notificationService =
     (Notification)(bcsae.getSourceAsBeanContextServices()).getService(
      this, this,Notification.class, null, this);
   }
   catch(Exception e)
   {
    notificationService=null;
   }
  }
 }

 @Override
 public void serviceRevoked(BeanContextServiceRevokedEvent bcsre)
 {
  super.serviceRevoked(bcsre);
  if(bcsre.getServiceClass()==Notification.class)
  {
   notificationService=null;
  }
 }

 @Override
 public void setBeanContext(BeanContext bc) throws PropertyVetoException
 {
  if(notificationService!=null)
  {
   if(getBeanContext()!=null)
   {
    if(BeanContextServices.class.isAssignableFrom(
     getBeanContext().getClass()))
    {
     BeanContextServices bcs=(BeanContextServices)getBeanContext();
     bcs.releaseService(this, this, notificationService);
    }
   }
  }
  super.setBeanContext(bc);
 }

}
```

# Bereitstellung eines Dienstes für einen BeanContext

## Service

```
public interface Service
{
 int calculate(java.lang.String in);
```

```
}
```

# Implementierung

```
class HashImpl extends Implementation
{
 public int calculate(String in)
 {
  return in!=null?in.hashCode():-1;
 }
}
```

# ServiceProvider

```
import de.netsysit.dataflowframework.modules.BeanContextChildModuleBase;
import java.beans.PropertyVetoException;
import java.beans.beancontext.BeanContext;
import java.beans.beancontext.BeanContextServiceAvailableEvent;
import java.beans.beancontext.BeanContextServiceProvider;
import java.beans.beancontext.BeanContextServiceRevokedEvent;
import java.beans.beancontext.BeanContextServices;
import java.util.Iterator;

public class ServiceProvider extends BeanContextChildModuleBase implements
  BeanContextServiceProvider
{

 @Override
 public void setBeanContext(BeanContext bc) throws PropertyVetoException
 {
  BeanContext former=getBeanContext();
  super.setBeanContext(bc);
  if(bc==null)
  {
   if(former!=null)
   {
    if(BeanContextServices.class.isAssignableFrom(former.getClass()))
    {
     ((BeanContextServices)former).revokeService(
      Service.class, this, true);
    }
   }
  }
  if(getBeanContext()!=null)
  {
   if(BeanContextServices.class.isAssignableFrom(
    getBeanContext().getClass()))
   {
    ((BeanContextServices)getBeanContext()).addService(
```

```
      Service.class, this);
    }
   }
  }
  @Override
  public void serviceAvailable(BeanContextServiceAvailableEvent bcsae)
  {
   super.serviceAvailable(bcsae);
  }
  @Override
  public void serviceRevoked(BeanContextServiceRevokedEvent bcsre)
  {
   super.serviceRevoked(bcsre);
   if(getBeanContext()!=null)
   {
    if(BeanContextServices.class.isAssignableFrom(
     getBeanContext().getClass()))
    {
     BeanContextServices bcs=(BeanContextServices)getBeanContext();
     if(bcsre.isServiceClass(Service.class))
     {
      try
      {
       bcs.addService(Service.class, this);
      }
      catch (Exception e)
      {
      }
     }
    }
   }
  }

  public Object getService(BeanContextServices bcs, Object requestor,
   Class serviceClass, Object serviceSelector)
  {
   return new HashImpl();
  }

  public void releaseService(BeanContextServices bcs, Object requestor,
   Object service)
  {
  }

  public Iterator getCurrentServiceSelectors(BeanContextServices bcs,
   Class serviceClass)
  {
   return java.util.Collections.emptyIterator();
  }
 }
```

# Generics

```
import de.netsysit.dataflowframework.logic.Generic;
import de.netsysit.dataflowframework.modules.ModuleBase;

public class Generics extends ModuleBase implements
  Generic
{
 private int counter;

 private Object lastInput;

 public void input(Object in)
 {
  lastInput=in;
  ++counter;
  process(in);
 }
 private void process(Object in)
 {
  if(counter%2==0)
  {
   Object old=getProcessed();
   send("processed", old, getProcessed());
  }
 }

 private Object processed;

 public Object getProcessed()
 {
  return processed;
 }


}
```

# Variable Anzahl von Inputs

```
import de.netsysit.dataflowframework.modules.ModuleBase;

public class Adder extends ModuleBase
{
 Number[] numbers;

 public Adder()
 {
  super();
  numbers=new Number[0];
 }

 public void input (Number in, String spec)
```

```
 {
  java.lang.String remainder=spec.substring("input".length());
  int i=remainder.length();
  if(i>0)
   i=java.lang.Integer.parseInt(remainder);
  while(i>=numbers.length)
  {
   Number[] nt=new Number[i+1];
   System.arraycopy(numbers, 0, nt, 0, numbers.length);
   numbers=nt;
  }
  numbers[i]=in;
  process();
 }

 private double processed;

 public double getProcessed()
 {
  return processed;
 }

 private void process()
 {
  double old=getProcessed();
  double t=0.0;
  for (Number number : numbers)
  {
   if(number!=null)
   {
    t+=number.doubleValue();
   }
  }
  processed=t;
  send("processed", old, getProcessed());
 }
}
```

# Variable Module zur Visualisierung

```
import de.elbosso.dataflowframework.modules.VariableNumberofInputsVisualization;
import java.awt.Component;
import javax.swing.JLabel;

public class MultiLabelVis extends
  VariableNumberofInputsVisualization<Number,Number>
{
 public MultiLabelVis()
 {
  super();
 }
 public void addNumber(Number newNumber,String spec)
 {
```

```
 super.addInput(newNumber,spec,"addNumber");
}
protected Component addComponent(Number model)
{
 JLabel label=new JLabel();
 label.setMinimumSize(new java.awt.Dimension(10,10));
 label.setPreferredSize(new java.awt.Dimension(10,10));
 return label;
}
protected void handleAddition(Component comp,Number model,
 Number newdata, boolean isDataSlot)
{
 if(isDataSlot)
 {
  if(newdata==null)
  {
   ((JLabel)comp).setText("");
  }
  else
  {
   ((JLabel)comp).setText(newdata.toString());
  }
 }
}
protected Number createInstanceToBeStoredAtBackingStore()
{
 return new Double(0);
}
}
```

# Variable Module

```
import de.netsysit.dataflowframework.logic.ConnectionEndPointDescription;
import de.netsysit.dataflowframework.logic.ConnectionEndPointDescriptionCollection
import de.netsysit.dataflowframework.modules.ModuleBase;
import de.netsysit.dataflowframework.ui.variable.VariableBean;
import java.io.File;

public class VariableDemo extends ModuleBase implements VariableBean
{
 private File definition;

 public File getDefinition()
 {
  return definition;
 }

 public void setDefinition(File definition)
 {
  this.definition = definition;
  reReadDocument();
 }
 private ConnectionEndPointDescriptionCollections inAndOuts;
```

```
 private void reReadDocument()
 {
  ConnectionEndPointDescriptionCollections old=getInAndOuts();
  inAndOuts=null;
  java.util.Properties props=new java.util.Properties();
  java.io.FileInputStream fis=null;
  try
  {
   fis=new java.io.FileInputStream(definition);
   props.load(fis);
   java.util.List<ConnectionEndPointDescription> l=
    new java.util.LinkedList();
   for (java.lang.String key : props.stringPropertyNames())
   {
    ConnectionEndPointDescription cepd=
     new ConnectionEndPointDescription();
    cepd.setPortName(key);
    cepd.setTypename(props.getProperty(key));
    l.add(cepd);
   }
   inAndOuts=new ConnectionEndPointDescriptionCollections(null, l);
   send("inAndOuts",old,getInAndOuts());
   fis.close();
  }
  catch(java.io.IOException exp){}
 }
 public ConnectionEndPointDescriptionCollections getInAndOuts()
 {
  return inAndOuts;
 }
 public void genericOperation(Object input, String name)
 {
 }
}
```

# Variable Module für User Eingaben

```
import de.elbosso.dataflowframework.modules.VariableNumberofParameters;
import java.awt.event.ActionEvent;
import java.awt.Component;
import java.awt.event.ActionListener;
import javax.swing.JCheckBox;
import de.netsysit.dataflowframework.modules.ModuleBase;

public class VariableNumberOfBooleans extends VariableNumberofParameters<Boolean>
{

 public VariableNumberOfBooleans()
 {
  super(java.lang.Boolean.class, false);
 }
```

```
 protected VariableNumberofParameters<Boolean>.Glue<Boolean>
   addComponent(String name)
 {
  return new Glue(name, this);
 }

 protected VariableNumberofParameters<Boolean>.Glue<Boolean>
   addComponent(Component comp, String string)
 {
  return new Glue(comp, string, this);
 }

 private class Glue extends VariableNumberofParameters<Boolean>.Glue<Boolean>
   implements ActionListener
 {

  private JCheckBox comp;

  Glue(Component comp, String propertyName, ModuleBase module)
  {
   super(propertyName, module);
   this.comp = (JCheckBox) comp;
   this.comp.addActionListener(this);
  }

  Glue(String propertyName, ModuleBase module)
  {
   super(propertyName, module);
  }

  public Component getComponent()
  {
   if (comp == null)
   {
    comp = new JCheckBox();
    comp.addActionListener(this);
   }
   return comp;
  }

  public Boolean performDataChange()
  {
   return comp.isSelected();
  }

  public void actionPerformed(ActionEvent e)
  {
   dataChange();
  }
 }
}
```

# Module zur Filterung

```
import de.elbosso.dataflowframework.modules.filter.rules.RuleBase;
import de.netsysit.util.validator.rules.FloatingPointMinMaxRule;
import de.netsysit.util.beans.InterfaceFactory;
import de.netsysit.dataflowframework.modules.BeanContextChildModuleBase;

public class FloatingPointMinMaxFilter extends
  RuleBase<Number, FloatingPointMinMaxRule>
{

 static
 {
  InterfaceFactory.setSuperclassAssociationForEventDispatchThread
   (FloatingPointMinMaxFilter.class, BeanContextChildModuleBase.class);
 }

 public FloatingPointMinMaxFilter()
 {
  super(Number.class, new FloatingPointMinMaxRule());
 }
}
```

# Implementierung einer alternativen Bedienoberfläche

## Modul

```
import de.netsysit.dataflowframework.modules.ModuleBase;
import de.netsysit.dataflowframework.ui.VisualComponentProvider;
import java.awt.BorderLayout;
import java.awt.Container;
import javax.swing.JPanel;

public class VCPCharacterCounter extends ModuleBase implements
  VisualComponentProvider
{
 private String lastInput;

 public void input(String in)
 {
  lastInput=in;
  countCharacters();
 }

 private int characterCount;

 public int getCharacterCount()
 {
```

```
 return characterCount;
}

private boolean ignoreSpaces;

public boolean isIgnoreSpaces()
{
 return ignoreSpaces;
}

public void setIgnoreSpaces(boolean ignoreSpaces)
{
 boolean old=isIgnoreSpaces();
 this.ignoreSpaces = ignoreSpaces;
 send("ignoreSpaces",old,isIgnoreSpaces());
}

private void countCharacters()
{
 int old=getCharacterCount();
 //Algorithmus-Implementierung hier
 send("characterCount",old,getCharacterCount());
}

public Container getVisualComponent()
{
 IgnoreSpaceToggle ignoreSpaceToggle=new IgnoreSpaceToggle(this);
 JPanel p=new JPanel(new BorderLayout());
 p.add(ignoreSpaceToggle);
 return p;
}

}
```

# Angepasster JToggleButton

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.beans.PropertyChangeEvent;
import java.beans.PropertyChangeListener;
import javax.swing.JToggleButton;

public class IgnoreSpaceToggle extends JToggleButton implements
  PropertyChangeListener,
  ActionListener
{
 private VCPCharacterCounter bean;

 public IgnoreSpaceToggle(VCPCharacterCounter bean)
 {
  super("Ignore Spaces");
```

```
  this.bean = bean;
  this.setSelected(bean.isIgnoreSpaces());
  bean.addPropertyChangeListener(this);
  addActionListener(this);
 }

 public void propertyChange(PropertyChangeEvent evt)
 {
  this.setSelected(bean.isIgnoreSpaces());
 }

 public void actionPerformed(ActionEvent e)
 {
  bean.setIgnoreSpaces(this.isSelected());
 }


}
```

# Actions für Module

```
import de.netsysit.dataflowframework.modules.ModuleBase;
import de.netsysit.dataflowframework.ui.ActionsProvider;
import java.awt.event.ActionEvent;
import java.util.Date;
import javax.swing.AbstractAction;
import javax.swing.Action;
import javax.swing.JOptionPane;

public class ActionsCharacterCounter extends ModuleBase implements
  ActionsProvider
{
 private String lastInput;

 public void input(String in)
 {
  lastInput=in;
  countCharacters();
 }

 private int characterCount;

 public int getCharacterCount()
 {
  return characterCount;
 }

 private boolean ignoreSpaces;

 public boolean isIgnoreSpaces()
 {
```

```
 return ignoreSpaces;
}

public void setIgnoreSpaces(boolean ignoreSpaces)
{
 boolean old=isIgnoreSpaces();
 this.ignoreSpaces = ignoreSpaces;
 send("ignoreSpaces",old,isIgnoreSpaces());
}

private void countCharacters()
{
 int old=getCharacterCount();
 //Algorithmus-Implementierung hier
 send("characterCount",old,getCharacterCount());
}

public Action[] provideCustomActions()
{
 return new Action[]{
  new AbstractAction("action") {

   public void actionPerformed(ActionEvent e)
   {
    JOptionPane.showMessageDialog(null, new Date().toString());
   }
  }
 };
}

public void preparePopupShow()
{

}

}
```

# SocketOut

```
import java.io.IOException;
import java.net.Socket;
import java.io.PrintWriter;
import java.net.InetAddress;

public class SocketOut extends
 de.netsysit.dataflowframework.modules.CommunicationTemplate
{
 public void sendData(java.lang.String data)
 {
  if(isConnectionEstablished())
  {
```

```
  pw.println(data);
 }
}
private String host;

public String getHost()
{
 return host;
}

public void setHost(String adr)
{
 String oldAdr = getHost();
 this.host = adr;
 send("host", oldAdr, getHost());
}
private int port;

public void setPort(int p)
{
 int oldPort = getPort();
 this.port = p;
 send("port", oldPort, getPort());
}

public int getPort()
{
 return port;
}
private Socket socket;
private PrintWriter pw;

protected void manageConnectionImpl()
{
 boolean old=isConnectionEstablished();
 try
 {
  if(port>-1)
  {
   socket = new Socket(InetAddress.getByName(host), port);
   pw=new PrintWriter(socket.getOutputStream());
  }
 }
 catch (IOException ex)
 {
  //warn für die Meldung eines Problems
  warn(null,ex.getMessage());
 }
 manageConnectionEstablished(old,socket!=null);
}
protected void closeDown()
{
 boolean old=isConnectionEstablished();
 if(socket!=null)
```

```
  {
   try
   {
    pw.close();
    socket.close();
   }
   catch (IOException ex)
   {
    //error für die Meldung eines kritischen Fehlers
    error(null,ex.getMessage());
   }
   socket=null;
  }
  manageConnectionEstablished(old,socket!=null);
 }
}
```

# Start/Stop

```
import de.elbosso.dataflowframework.modules.StartStopModule;
import java.util.Random;

public class FastRandomSkalar extends StartStopModule
{
 public FastRandomSkalar()
 {
  super(FastRandomSkalar.class.getName());
 }
 private double value;

 public double getValue()
 {
  return value;
 }
 public void setRunning(boolean newrunning)
 {
  super.setRunning(newrunning);
  if(isRunning()==true)
  {
   processData(java.lang.Boolean.TRUE);
  }
  else
  {
   processData(null);
  }
 }
 private final static Random rand=new Random(System.nanoTime());

 protected void doWork(java.lang.Object ref)
 {
  if(disposed==false)
```

```
  {
   double old=getValue();
   value=rand.nextDouble();
   send("value",old,getValue());
  }
 }
}
```

# Start/Stop mit Einzelschritt

```
import de.elbosso.dataflowframework.modules.StartStopModuleWithDoOnce;
import java.util.Random;

public class FastRandomSkalarWithDoOnce extends StartStopModuleWithDoOnce
{
 public FastRandomSkalarWithDoOnce()
 {
  super(FastRandomSkalarWithDoOnce.class.getName());
 }
 private double value;

 public double getValue()
 {
  return value;
 }
 public void setRunning(boolean newrunning)
 {
  super.setRunning(newrunning);
  if(isRunning()==true)
  {
   processData(java.lang.Boolean.TRUE);
  }
  else
  {
   processData(null);
  }
 }
 private final static Random rand=new Random(System.nanoTime());

 protected void doWork(java.lang.Object ref)
 {
  if(disposed==false)
  {
   performWork();
  }
 }
 private void performWork()
 {
  double old=getValue();
  value=rand.nextDouble();
  send("value",old,getValue());
```

```
 }
 public void doOnce()
 {
  performWork();
 }
}
```

# Enterprise JavaBeans als Module

```
import de.elbosso.dataflowframework.modules.EJBModuleBase;
import java.util.Properties;
import de.netsysit.util.threads.CubbyHole;
import de.netsysit.util.threads.SimpleBufferingCubbyHole;

public class EJBWrapper extends EJBModuleBase<RemoteCalculator>
{

 public EJBWrapper()
 {
  super(RemoteCalculator.class, EJBWrapper.class.getName());
 }

 protected CubbyHole createCubbyHole()
 {
  return new SimpleBufferingCubbyHole();
 }

 protected Properties getEnvironment()
 {
  java.util.Properties env = new java.util.Properties();
  env.put(javax.naming.Context.INITIAL_CONTEXT_FACTORY,
    "org.jboss.naming.remote.client.InitialContextFactory");
  env.put(javax.naming.Context.PROVIDER_URL,
    "http-remoting://da_host:da_port");
  env.put(javax.naming.Context.SECURITY_PRINCIPAL,
    "da_user");
  env.put(javax.naming.Context.SECURITY_CREDENTIALS,
    "da_password");
  env.put(javax.naming.Context.URL_PKG_PREFIXES,
    "org.jboss.ejb.client.naming");
  env.put("jboss.naming.client.ejb.context",
    true);
  return env;
 }

 protected String getJndiName()
 {
  return "ejb:/wildfly-ejb-remote-server-side/CalculatorBean!";
 }
 private int result;
```

```
 public int getResult()
 {
  return result;
 }
 private Number a;
 private Number b;

 public void inputA(Number in)
 {
  a = in;
  processData(in);
 }

 public void inputB(Number in)
 {
  b = in;
  processData(in);
 }

 protected void doWork(Object ref)
 {
  if ((a != null) && (b != null))
  {
   int old = getResult();
   result = ejb.add(a.intValue(), b.intValue());
   send("result", old, getResult());
  }
 }
}
```

# Module mit geänderter Kommunikationsmetapher

```
import de.netsysit.util.beans.InterfaceFactory;
import de.elbosso.dataflowframework.modules.MapMessageModule;
import de.netsysit.dataflowframework.modules.ThreadingBeanContextChildModuleBase;
import de.netsysit.util.threads.CubbyHole;
import de.netsysit.util.threads.SimpleNonBlockingCubbyHole;

public class MappedAdder extends MapMessageModule
{

 static
 {
  InterfaceFactory.setSuperclassAssociationForEventDispatchThread(
    MappedAdder.class, ThreadingBeanContextChildModuleBase.class);
 }
 private double result;

 public MappedAdder()
```

```
 {
  super(MappedAdder.class.getName());
  java.util.Properties props = new java.util.Properties();
  props.setProperty("a", "a");
  props.setProperty("b", "b");
  setProps(props);
 }

 @Override
 protected CubbyHole createCubbyHole()
 {
  return new SimpleNonBlockingCubbyHole();
 }

 @Override
 protected void doWork(Object ref) throws InterruptedException
 {
//Kopie der eingehenden Map
  java.util.Map map = (java.util.Map) ref;
  Number a = null;
  Number b = null;
//Versuch, die erwarteten Daten aus der Map zu holen
  a = (Number) getData(map, "a");
  b = (Number) getData(map, "b");
  if ((a != null) && (b != null))
  {
   result = a.doubleValue() + b.doubleValue();
//Hinzufügen der Ergebnisses zur Map und Versenden
   java.util.Map old = null;
   map.put("result", result);
   send("output", old, map);
  }
 }
}
```

# Verteiltes Arbeiten (Remoting)

## Modul

```
import de.elbosso.dataflowframework.modules.RemoteModule;
import de.elbosso.dataflowframework.modules.helper.rmi.Hello;
import de.elbosso.dataflowframework.modules.helper.rmi.HelloImpl;

public class RemotingExample extends RemoteModule<Hello>
{

 public RemotingExample()
 {
  super(RemotingExample.class.getName());
 }
```

```
 protected void installWorker(java.lang.String newLocation)
 {
  installWorker(newLocation, HelloImpl.class, new Class[]
  {
   Hello.class
  });
 }

 protected Hello createLocalInstance()
 {
  return new HelloImpl();
 }

 public void input(Object in)
 {
  processData(in);//Hierdurch Start des Algorithmus
 }
 private Object data;

 public synchronized Object getData()
 {
  return data;
 }

 public synchronized void setData(Object data)
 {
  java.lang.Object old = getData();
  this.data = data;
  send("data", old, getData());
 }

 @Override
 protected void doWork(Object ref) throws InterruptedException
 {
  Hello h = getRemoteObject();
  if (h != null)
  {
   setData(h.hello());
  }
 }
}
```

# Interface

```
import java.io.Serializable;

public interface Hello extends Serializable
{
 public String hello();
}
```

## Implementierung

```
import java.net.UnknownHostException;
import java.net.Inet4Address;

public class HelloImpl implements Hello
{
 public String hello()
 {
  String rv="Hi there! - dont know where I am! ";
  try
  {
   rv="Hi there! - from "+
   Inet4Address.getLocalHost().getHostName()+
   " ";
  }
  catch (UnknownHostException ex)
  {
  }
  return rv+Thread.currentThread().getName()+" "+toString();
 }
}
```

# Getaktete Module

```
import de.netsysit.dataflowframework.modules.ModuleBase;
import de.elbosso.util.Latch;

public class ClockedAnd extends ModuleBase
{
 private Latch latch;

 public ClockedAnd()
 {
  super();
  latch=new Latch();
 }
 private boolean result;

 public boolean getResult()
 {
  return result;
 }
 public void inputA(boolean in)
 {
  latch.latch("inputA",in);
 }
 public void inputB(boolean in)
 {
```

```
  latch.latch("inputB",in);
 }
 public void clock(java.lang.Object in)
 {
  compute();
 }
 private void compute()
 {
  boolean old=getResult();
  //inputA holen
  boolean a = latch.fetchBoolean("inputA");
  //inputb holen
  boolean b = latch.fetchBoolean("inputB");
  result=a&&b;
  //Events versenden!
  send("result",old,getResult());
 }
}
```

# StateUpdater

```
import java.awt.GridLayout;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JList;
import de.netsysit.dataflowframework.ui.beans.PopupStateUpdater;
import de.elbosso.ui.moduleworkspace.connected.Slot;

public class NumberStateUpdaterDemo extends PopupStateUpdater
{
 private JLabel last;
 private JLabel current;
 private Object old;

 public NumberStateUpdaterDemo(Slot slot, JList list)
 {
  super(slot,list);
  JPanel p=new JPanel(new GridLayout(0, 2));
  JLabel l=new JLabel("letztes:");
  l.setOpaque(false);
  p.add(l);
  last=new javax.swing.JLabel("");
  p.add(last);
  l=new JLabel("aktuelles:");
  l.setOpaque(false);
  p.add(l);
  current=new javax.swing.JLabel("");
  p.add(current);
  toplevel.add(p);
  last.setOpaque(false);
  current.setOpaque(false);
```

```
  p.setOpaque(false);
 }
 protected void update(Object object)
 {
  last.setText(old!=null?old.toString():"--");
  current.setText(object!=null?object.toString():"--");
  old=object;
 }
}
```

# DemoWorkspaceExporter

```
import de.netsysit.dataflowframework.logic.services.workspace.WorkspaceExporter.Su
import de.netsysit.dataflowframework.ui.LinkDescription;
import de.netsysit.dataflowframework.ui.ModuleWidgetDescription;
import de.netsysit.dataflowframework.ui.WorkspaceDescription;
import java.io.OutputStream;
import de.netsysit.dataflowframework.logic.services.workspace.WorkspaceExporter;

public class DemoWorkspaceExporter extends java.lang.Object implements
 WorkspaceExporter
{

 public DemoWorkspaceExporter()
 {
  super();
 }

 public boolean save(Support support,
  WorkspaceDescription[] wda, OutputStream os)
 {
  boolean rv=false;
  java.io.PrintWriter pw=null;
  pw=new java.io.PrintWriter(os);
  if(wda!=null)
  {
   for (WorkspaceDescription workspaceDescription : wda)
   {
    ModuleWidgetDescription mwd[] =
     workspaceDescription.getModuleWidgetDescription();
    if(mwd!=null)
    {
     for (ModuleWidgetDescription moduleWidgetDescription : mwd)
     {
      pw.print(moduleWidgetDescription.getTitle());
      pw.print("\t");
      Object module=moduleWidgetDescription.getModule();
      pw.println(module.getClass().getName());
     }
    }
    LinkDescription ld[]=workspaceDescription.getLinkDescription();
```

```
    if(ld!=null)
    {
     for (LinkDescription linkDescription : ld)
     {
      pw.println(linkDescription);
     }
    }
   }
  }
  rv=true;
  if(pw!=null)
   pw.close();
  return rv;
 }

 public String getSuffix()
 {
  return "exdemo";
 }

}
```

# Anhang B. BeanContext Services

# LoggingConfig

## Einsatz

Dieser Service dient dazu, die Konfiguration von Log4J bezüglich bestimmter Logging-Kategorien aus einer Konfigurationsdatei zu lesen und daraus ein TableModel zu erzeugen.

## Methoden

### getConfig

```
public de.netsysit.model.table.Log4JConfig getConfig(Class[] logger,
                                                     java.net.URL url);
```

Diese Methode dient dazu, die Konfiguration von Log4J bezüglich bestimmter Logging-Kategorien aus einer Konfigurationsdatei zu lesen und daraus ein TableModel zu erzeugen.

#### Parameter

logger — Dieser Parameter enthält als Elemente des Arrays die Klassen, deren Log4J-Konfiguration ausgelesen und in das zu erzeugende Tabellenmodell integriert werden sollen. Die Klassen (beziehungsweise deren voll qualifizierte Namen) übernehmen dabei die Funktion der Log4J-Logging-Kategorien.

url — Dieser Parameter verweist auf die Konfigurationsdatei, aus der die Konfigurationen für die übergebenen Kategorien ausgelesen werden sollen.

#### Rückgabewert

Das TableModel enthält die Informationen über die aktuelle Konfiguration von Log4j aus der übergebenen Konfigurationsdatei bezüglich der übergebenen Kategorien.

# DialogParentFrame

## Einsatz

Dieser Service dient dazu, visuellen Komponenten (Dialogen,...) Zugriff auf das Hauptfenster einzuräumen, so dass Module Dialoge und Fenster im Kontext des Hauptfensters öffnen können.

## Methoden

### getParentFrame

```
public java.awt.Frame getParentFrame();
```

Diese Methode dient dazu, eine Referenz auf das Hauptfenster der Anwendung zur Verfügung zu stellen.

### Rückgabewert

Referenz auf das Hauptfenster der Anwendung.

## getParentComponent

```
public java.awt.Component getParentComponent();
```

Diese Methode dient dazu, eine Referenz auf eine Komponente der Anwendung zur Verfügung zu stellen, die als übergeordnete Komponente benutzt werden kann, um neue Komponenten zu erzeugen. Dabei bedeutet übergeordnet in diesem Falle logisch wie in der Beziehung Hauptfenster der Anwendung und deren Dialoge.

### Rückgabewert

Referenz auf eine Komponente.

# IDProvider

# Einsatz

Dieser Service hat einen sehr speziellen Zweck: er bietet die Möglichkeit, zu einer bestimmten Modulinstanz den Titel des ModuleWidgets zu ermitteln.

# Methoden

## getTitle

```
public String getTitle(Object client);
```

Diese Methode bietet die Möglichkeit, zu einer bestimmten Modulinstanz den Titel des ModuleWidgets zu ermitteln.

### Parameter

client Referenz auf die Modulinstanz, zu der der Name des ModuleWidgets gesucht wird.

### Rückgabewert

Der Titel des für die übergebene Modulinstanz erzeugten ModuleWidgets.

# Notification

# Einsatz

Dieser Service bietet Zugriff auf verschiedene Aspekte des System-Tray.

# Methoden

## notifyInfo

```
public void notifyInfo(String title,
```

```
                                    String message);
```

Diese Methode bietet die Möglichkeit, den Anwender mittels eines System-Tray-Popups zu informieren.

### Parameter

title Überschrift für das Popup.

message Anzuzeigende Information.

## addAction

```
public void addAction(javax.swing.Action action);
```

Diese Methode bietet die Möglichkeit, eine Action zum Popupmenü des System-Tray hinzuzufügen.

### Parameter

action Hinzuzufügende Action.

## removeAction

```
public void removeAction(javax.swing.Action action);
```

Diese Methode bietet die Möglichkeit, eine Action aus dem Popupmenü des System-Tray zu entfernen.

### Parameter

action Zu entfernende Action.

# ReportingEngine

# Einsatz

Dieser Service stellt Methoden zur Berichtsdefinition zur Verfügung

# Methoden

## getReportingItemSpecifications

```
public java.util.List<ReportingItemSpecification> getReportingItemSpecifications
```

Durch diese Methode wird der Service angewiesen, eine Collection von ReportingItemSpecifications - Elemente eines Berichts - zu liefern.

### Rückgabewert

Eine Liste mit den entsprechenden ReportingItemSpecifications

## produceReportingStatement

```
public String produceReportingStatement(java.util.List<ReportingItemSpecificatio
                                        java.sql.Timestamp start,
                                        java.sql.Timestamp end);
```

Diese Methode erstellt unter Berücksichtigung der Elemente des Berichts und des Start- und Endzeitpunktes eine SQL-Abfrage, die die Inhalte des zu erzeugenden Berichtes in der Datenbank selektiert.

### Parameter

specs Teile des zu erzeugenden Berichtes

start Startzeitpunkt des Berichtes - Ereignisse, die vor diesem Zeitpunkt liegen, werden nicht berücksichtigt

end Startzeitpunkt des Berichtes - Ereignisse, die nach diesem Zeitpunkt liegen, werden nicht berücksichtigt

### Rückgabewert

Die generierte SQL-Abfrage.

# WorkspaceAPI

# Einsatz

Dieser Service erlaubt den Zugriff auf ModuleWidgets

# Methoden

## getWidget

```
public de.netsysit.ui.moduleworkspace.ModuleWidget getWidget(Object bean);
```

Diese Methode erlaubt den Zugriff auf ein ModulWidget, das zu der übergebenen Modulinstanz gehört.

### Parameter

bean Referenz auf die Modulinstanz, deren ModuleWidget gesucht wird.

### Rückgabewert

Eine Referenz auf das gesuchte ModuleWidget

## getWidgetByUniqueId

```
public de.netsysit.ui.moduleworkspace.ModuleWidget getWidgetByUniqueId(String ui
```

Diese Methode erlaubt den Zugriff auf das ModulWidget, das zu der Modulinstanz gehört, die die übergebene ID trägt.

### Parameter

uid Unique ID der Modulinstanz, deren ModuleWidget gesucht wird.

### Rückgabewert

Eine Referenz auf das gesuchte ModuleWidget

# Bonjour

## Einsatz

Dieser Dienst erlaubt die Interaktion mit der Technik, die durch Apple unter dem Namen Bonjour bekannt wurde - andere Namen sind zum Beispiel Zeroconf,...

Der Dienst dient dazu, im Netzwerk Dienste ausfindig zu machen, die eine bestimmte, über den Namen des Dienstes identifizierte Funktionalität anbieten. Der Name wird dabei bei der Instantiierung der Service-Instanz als ServiceSelektor angegeben.

## Methoden

### getSocketAddresses

```
public de.netsysit.util.beans.context.service.impl.bonjour.SocketAddress[] getSo
```

Diese Methode dient dazu, für den gegebenen Namen alle Endpunkte Im Netzwerk zu ermitteln.

#### Rückgabewert

Ein Array mit im Netzwerk publizierten Endpunkten, die einen Dienst laut dem übergebenen ServiceSelektor implementieren.

### addBonjourListener

```
public void addBonjourListener(de.netsysit.util.beans.context.service.impl.bonjo
```

Diese Methode erlaubt es, über Änderungen im Bonjour-Netzwerk informiert zu werden. Zu solchen Änderungen oder Ereignissen gehören zum Beispiel das Anmelden oder Publizieren eines neuen Dienstes im Netzwerk oder auch das Abmelden oder Verschwinden eines solchen Dienstes oder Endpunktes.

#### Parameter

listener Instanz, die auf die verschiedenen Ereignisse reagieren soll

### removeBonjourListener

```
public void removeBonjourListener(de.netsysit.util.beans.context.service.impl.bo
```

Diese Methode erlaubt es, die Benachrichtigung über Ereignisse im Bonjour-Netzwerk - wie etwa Publizieren oder Entfernen eines Endpunktes wieder abzustellen.

#### Parameter

listener Instanz, die die Ereignisse abonniert hatte und nun nicht mehr darüber informiert werden soll.

# JSMandantManager

## Einsatz

Diese Methode dient dazu, Workspaces mandentenfähig zu machen.

Darunter versteht man, dass alle Daten, die im Workspace zirkulieren, noch einen gedachten Namen erhalten, der dann den sogenannten Mandanten symbolisiert. Damit ist es möglich, zum Beispiel die Verarbeitung folgendermaßen zu steuern: ein Modul, das die zu verarbeitenden Daten in Rohform liefert, wird dadurch informiert, dass es Daten anliefern soll, die von Mandant A stammen. Dieses Modul ändert seine Datenbeschaffungsstrategie daraufhin entsprechend ab. Mehrere nachgeordnete Module verarbeiten die Daten - ihnen ist es egal, von welchem Mandanten sie stammen, ja sie müssen gar nicht wissen, dass eine Mandantenabhängige Verarbeitung durchgeführt wird. Am Ende der Verarbeitungskette existiert dann noch ein Modul, das die Ergebnisse nach Mandanten sortiert wieder speichert. Nur das erste und letzte Modul in der Kette müssen über den aktuellen Mandanten informiert sein - dafür brauchen sie Zugriff auf diesen Dienst.

Im Moment werden die Mandanten einfach zyklisch weitergeschaltet - es existiert keine Wahlfreiheit, für welchen Mandanten als nächstes gearbeitet werden soll.

# Methoden

## getCurrentMandant

```
public String getCurrentMandant();
```

Diese Methode erlaubt den Zugriff auf den aktuell bearbeiteten Mandanten

### Rückgabewert

Name des aktuell bearbeiteten Mandanten

## switchMandant

```
public void switchMandant();
```

Diese Methode ersetzt den letzten aktuellen Mandanten durch den nächsten in der Liste.

## addMandant

```
public void addMandant(String m);
```

Diese Methode erlaubt das Hinzufügen eines neuen Mandanten

### Parameter

m Der Name des neuen Mandanten - dieser muss für den Dienst einzigartig sein.

# BackgroundExecutor

# Einsatz

Dieser Dienst erlaubt es, Instanzen von Klassen, die das Interface Runnable implementieren, in Hintergrund-Threads auszuführen.

# Methoden

## execute

```
public de.netsysit.util.threads.Synchronizer execute(Runnable r);
```

Diese Methode fordert den Dienst auf, die run-Methoden der übergebenen Instanz in einem eigenen Thread auszuführen. Diese Methode startet die Ausführung in einem eigenen Thread und kehrt sofort zurück

### Parameter

r Eine Instanz einer Klasse, die das Interface Runnable implementiert. Der Code der Methode run wird im Hintergrund ausgeführt.

### Rückgabewert

Instanz, die es erlaubt, auf das Ende der Ausführung zu warten.

## execute

```
public de.netsysit.util.threads.Synchronizer execute(Runnable r,
                                                     int priority);
```

Diese Methode fordert den Dienst auf, die run-Methoden der übergebenen Instanz in einem eigenen Thread mit der angegebenen Priorität auszuführen. Diese Methode startet die Ausführung in einem eigenen Thread und kehrt sofort zurück

### Parameter

r Eine Instanz einer Klasse, die das Interface Runnable implementiert. Der Code der Methode run wird im Hintergrund ausgeführt.

priority Die gewünschte Priorität, mit der der übergebene Code ausgeführt werden soll.

### Rückgabewert

Instanz, die es erlaubt, auf das Ende der Ausführung zu warten.

## execute

```
public void execute(Runnable[] r);
```

Diese Methode fordert den Dienst auf, die run-Methoden der übergebenen Instanz in einem eigenen Thread mit der angegebenen Priorität auszuführen. Diese Methode startet die Ausführung in einem eigenen Thread und kehrt sofort zurück

### Parameter

r Ein Array mit einer beliebigen Anzahl von Instanz von Klassen, die das Interface Runnable implementieren. Der Code der Methoden namens run wird jeweils im Hintergrund ausgeführt.

# ApplicationServer

# Einsatz

Dieser Dienst erlaubt es, Servlets zu publizieren. Die Implementierung im dWb + benutzt einen Jetty-Container, der dann gestartet wird, wenn die System-Property `dWb.service.ApplicationServer.port` einen gültigen Wert für einen TCP-Port aufweist. In diesem Fall wird versucht, diesen für eingehende Verbindungen zu öffnen. Ist das erfolgreich, steht dieser Dienst in dWb+ zur Verfügung.

# Methoden

## installServlet

```
public void installServlet(javax.servlet.http.HttpServlet servlet,
                           String path);
```

Diese Methode fordert den Dienst auf, das übergebene Servlet unter dem angegebenen Pfad zu publizieren

### Parameter

servlet Das Servlet, das unter dem angegebenen Pfad publiziert werden soll

path Der Pfad, der zum Start der Ausführung des Servlets führt.

## removeServlet

```
public void removeServlet(javax.servlet.http.HttpServlet servlet);
```

Diese Methode fordert den Dienst auf, das übergebene Servlet zu entfernen (nicht länger zu publizieren)

### Parameter

servlet Das Servlet, das entfernt werden soll

## getPort

```
public int execute();
```

Diese Methode liefert den TCP-Port, an dem Jetty eingehende Verbindungen erwartet.

### Rückgabewert

Nummer des TCP-Ports, an dem Jetty eingehende Verbindungen erwartet.

# Environment

# Einsatz

Dieser Dienst erlaubt es, Eigenschaften abzufragen und zu setzen, die für die jeweilige Modul-Instanz gelten - vergleiche dazu auch Abschnitt „Eigenschaften“.

# Methoden

## getProperty

```
public java.lang.String getProperty(java.lang.String name);
```

Diese Methode fragt den Wert einer Eigenschaft ab. Existiert die Eigenschaft nicht, liefert die Methode null zurück.

### Parameter

name Der Name der Eigenschaft, die abgefragt werden soll

### getProperty

```
public java.lang.String getProperty(java.lang.String name,
                                     java.lang.String defaultValue);
```

Diese Methode fragt den Wert einer Eigenschaft ab. Existiert die Eigenschaft nicht, liefert die Methode den zweiten Parameter zurück.

#### Parameter

| | |
|---|---|
| name | Der Name der Eigenschaft, die abgefragt werden soll |
| defaultValue | Der Wert, der zurückgegeben werden soll, falls eine Eigenschaft unter dem angegebenen Namen nicht definiert ist |

### setProperty

```
public void setProperty(java.lang.String name,
                        java.lang.String value);
```

Diese Methode setzt den Wert einer Eigenschaft.

#### Parameter

| | |
|---|---|
| name | Der Name der Eigenschaft, deren Wert gesetzt werden soll |
| value | Der Wert, der der Eigenscahft mit dem angegebenen Namen zugewiesen werden soll |

### getProperties

```
public java.util.Enumeration getProperties();
```

Diese Methode liefert die Namen aller für diese Modulinstanz definierten Eigenschaftenn.

# BeanContextServices

## Einsatz

Dieser Service dient dazu, Zugriff auf den BeanContext zu bekommen, damit man zur Laufzeit Services registrieren und deregistrieren kann.

Das kann zum Beispiel dann nützlich sein, wenn man einen BeanContext-Service in einem Modul implementiert, allerdings nicht sofort bei Instantiierung diesen Dienst registrieren kann oder will. Da ist zum Beispiel immer dann der Fall, wenn der Dienst externe Ressourcen wie eine Netwerk- oder Datenbankverbindung benötigt, diese jedoch erst noch konfiguriert werden muss. Durch Benutzung dieses Service kann man den neu geschaffenen dann registrieren, wenn die benötigte Ressource zur Verfgügung steht und deregistrieren, wenn der Zugriff auf die Ressource -aus welchen Gründen auch immer - nicht mehr möglich ist.
Die API entspricht der von `java.beans.beancontext.BeanContextServicesSupport`.

# Anhang C. Erstellen von Komponenten für dWb+ mittels Maven

## Einleitung

In allen Bereichen des vorliegenden Buches geht es um die Erstellung von Softwarekomponenten für das System dWb+ - seien es Module, StateUpdaters (Kapitel 27, *StateUpdaters* im Pogrammierhandbuch dWb +), Implementierungen von Interfaces als Teile der Service Provider Infrastructure (Kapitel 31, *Service Provider Infrastructure* im Pogrammierhandbuch dWb+) oder andere.

Diese Komponenten werden für dWb+ nutzbar, indem sie in Jar-Dateien eingepackt und in dafür vorgesehene Verzeichnisse platziert werden. Wie diese Jar-Dateien entstehen, war bisher noch nicht Thema - und aus gutem Grund: Der Autor wollte nicht vorschreiben, wie man programmiert oder welche IDE oder welches Build-System eingesetzt werden soll. Es gibt für solche Festlegungen keinerlei technische Gründe.

Nichtsdestotrotz wird in diesem Abschnitt der Fokus auf das Buildsystem namens Maven gelegt werden. Es werden POM-Dateien gezeigt, mit denen sich Modularchive und SPI-Komponenten erstellen lassen. Davon ausgehend kann man solche für State-Updater oder andere Komponenten ableiten.

## Module

Für Module existiert ein entsprechendes Repository unter der Adresse https://github.com/elbosso/dWb_custom_modules als Ausgangspunkt für eigene Implementierungen.

Alternativ ist es auch möglich Maven zu beauftragen, ein entsprechendes Projekt aus dem ebenfalls auf Github bereitgestellten Archetypen [https://github.com/elbosso/dWb_custom_modules_archetype] zu erzeugen.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <groupId>de.yourcompany</groupId>
 <artifactId>custom_maven_modules</artifactId>
 <packaging>jar</packaging>
 <version>1.0.0-SNAPSHOT</version>
 <name>Custom Maven Modules for dWb+</name>
 <url>http://maven.apache.org</url>
 <properties>
  <project.custom.encoding>UTF-8</project.custom.encoding>
  <project.custom.java.version>1.6</project.custom.java.version>
  <maven.compiler.source>
   ${project.custom.java.version}
  </maven.compiler.source>
  <maven.compiler.target>
   ${project.custom.java.version}
```

```
 </maven.compiler.target>
 <project.build.sourceEncoding>
  ${project.custom.encoding}
 </project.build.sourceEncoding>
 <project.build.outputEncoding>
  ${project.custom.encoding}
 </project.build.outputEncoding>
 <project.reporting.outputEncoding>
  ${project.custom.encoding}
 </project.reporting.outputEncoding>
 <project.scm.id>
  <!-- insert yours here! -->
 </project.scm.id>
 <elbosso.artifacts.version>
  1.8.0
 </elbosso.artifacts.version>
</properties>
<build>
 <pluginManagement>
  <plugins>
   <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
    <version>2.5.1</version>
   </plugin>
  </plugins>
 </pluginManagement>
 <plugins>
  <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-release-plugin</artifactId>
   <version>3.0.0-M1</version>
   <configuration>
    <!--preparationGoals>package assembly:single</preparationGoals-->
   </configuration>
  </plugin>
 </plugins>
</build>
<repositories>
 <repository>
  <id>repsy</id>
  <name>EL BOSSOs Maven Repository on Repsy</name>
  <url>https://repo.repsy.io/mvn/elbosso/default</url>
 </repository>
 <repository>
  <id>central</id>
  <url>https://repo1.maven.org/maven2</url>
  <releases>
   <enabled>true</enabled>
  </releases>
 </repository>
 <!--BeanShell-->
 <repository>
  <id>Boundless Repository</id>
```

```
    <url> https://repo.boundlessgeo.com/main/</url>
   </repository>
  </repositories>
  <dependencyManagement>
   <dependencies>
    <dependency>
     <groupId>de.elbosso</groupId>
     <artifactId>bom</artifactId>
     <version>${elbosso.artifacts.version}</version>
     <type>pom</type>
     <scope>import</scope>
    </dependency>
   </dependencies>
  </dependencyManagement>

  <dependencies>
   <dependency>
    <groupId>de.elbosso</groupId>
    <artifactId>dataflowframework_api</artifactId>
    <version>${elbosso.artifacts.version}</version>
   </dependency>
   <dependency>
    <groupId>de.elbosso</groupId>
    <artifactId>util_annotations</artifactId>

    <scope>compile</scope>
   </dependency>
   <dependency>
    <groupId>de.elbosso.annotation.processors</groupId>
    <artifactId>beaninfo</artifactId>

    <scope>compile</scope>
   </dependency>
   <dependency>
    <groupId>de.elbosso</groupId>
    <artifactId>algorithms</artifactId>

    <scope>compile</scope>
   </dependency>
   <dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
   </dependency>

  </dependencies>
  <distributionManagement>
   <repository>
    <id>internal.repo</id>
    <name>Internal repo</name>
    <url>file:///tmp/</url>
   </repository>
  </distributionManagement>
 </project>
```

Mittels Aufruf des Kommandos **mvn** `package` wird dann aus den Quelltexten eine Jar-Datei, die die Module enthält.

# Service Provider Interface

Für Imnplementierungen von Service Provider Interfaces existiert ein Repository unter der Adresse https://gitlab.com/elbosso/dwb_custom_linklifecyclehandler als Ausgangspunkt für eigene Implementierungen. Es demonstriert exemplarisch die Implementierung des Interface `de.elbosso.ui.moduleworkspace.LinkLifeCycleHandler`.

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
   http://maven.apache.org/maven-v4_0_0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <groupId>de.yourcompanyname</groupId>
 <artifactId>ArtifactId</artifactId>
 <packaging>jar</packaging>
 <version>0.1-SNAPSHOT</version>
 <name>Project name</name>
 <url>http://maven.apache.org</url>
 <properties>
  <project.custom.encoding>UTF-8</project.custom.encoding>
  <project.custom.java.version>1.6</project.custom.java.version>
  <maven.compiler.source>
   ${project.custom.java.version}
  </maven.compiler.source>
  <maven.compiler.target>
   ${project.custom.java.version}
  </maven.compiler.target>
  <project.build.sourceEncoding>
   ${project.custom.encoding}
  </project.build.sourceEncoding>
  <project.build.outputEncoding>
   ${project.custom.encoding}
  </project.build.outputEncoding>
  <project.reporting.outputEncoding>
   ${project.custom.encoding}
  </project.reporting.outputEncoding>
  <project.scm.id>
   <!-- insert yours here! -->
  </project.scm.id>
  <elbosso.artifacts.version>
   1.8.0
  </elbosso.artifacts.version>
 </properties>
 <build>
  <pluginManagement>
   <plugins>
    <plugin>
     <groupId>org.apache.maven.plugins</groupId>
```

```
    <artifactId>maven-compiler-plugin</artifactId>
    <version>2.5.1</version>
   </plugin>
  </plugins>
 </pluginManagement>
 <plugins>
  <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-release-plugin</artifactId>
   <version>3.0.0-M1</version>
   <configuration>
    <!--preparationGoals>package assembly:single</preparationGoals-->
   </configuration>
  </plugin>
 </plugins>
</build>
<repositories>
 <repository>
  <id>repsy</id>
  <name>EL BOSSOs Maven Repository on Repsy</name>
  <url>https://repo.repsy.io/mvn/elbosso/default</url>
 </repository>
 <repository>
  <id>central</id>
  <url>https://repo1.maven.org/maven2</url>
  <releases>
   <enabled>true</enabled>
  </releases>
 </repository>
 <!--BeanShell-->
 <repository>
  <id>Boundless Repository</id>
  <url> https://repo.boundlessgeo.com/main/</url>
 </repository>
</repositories>
<dependencyManagement>
 <dependencies>
  <dependency>
   <groupId>de.elbosso</groupId>
   <artifactId>bom</artifactId>
   <version>${elbosso.artifacts.version}</version>
   <type>pom</type>
   <scope>import</scope>
  </dependency>
 </dependencies>
</dependencyManagement>

<dependencies>
 <dependency>
  <groupId>de.elbosso</groupId>
  <artifactId>dataflowframework_api</artifactId>
  <version>${elbosso.artifacts.version}</version>
 </dependency>
 <dependency>
```

```
    <groupId>de.elbosso</groupId>
    <artifactId>util_annotations</artifactId>

    <scope>compile</scope>
   </dependency>
   <dependency>
    <groupId>de.elbosso.annotation.processors</groupId>
    <artifactId>beaninfo</artifactId>

    <scope>compile</scope>
   </dependency>
   <dependency>
    <groupId>de.elbosso</groupId>
    <artifactId>algorithms</artifactId>

    <scope>compile</scope>
   </dependency>
   <dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
   </dependency>

  </dependencies>
  <distributionManagement>
   <repository>
    <id>internal.repo</id>
    <name>Internal repo</name>
    <url>file:///tmp/</url>
   </repository>
  </distributionManagement>
</project>
```

Mittels Aufruf des Kommandos **mvn** `package` wird dann aus den Quelltexten eine Jar-Datei, die die Implementierung des Service-Provider-Interface enthält.

# OSGI-Bundles

Zur Erstellung von OSGI-Bundles (bei Einsatz des entsprechenden Plugins `osgimodules`) als Sammlung für Module geht man genauso vor, wie in „Module“ beschrieben, sorgt aber dafür, dass das Manifest die für OSGI benötigten Informationen enthält:

```
Manifest-Version: 1.0
Export-Package: the.actual.module.packages, separated.by.comma
Bundle-Vendor: Put your name here!
Bundle-Version: 1.0.0
Bundle-Localization: plugin
Bundle-Name: Some name or other...
Created-By: Put your name here!
Bundle-Activator: only.needed.if.there.actually.is.an.Activator
Bundle-ManifestVersion: 2
Import-Package: org.osgi.framework, de.netsysit.dataflowframework.modu
 les,de.elbosso.dataflowframework.modules,de.elbosso.util.beans,org.ap
```

```
 ache.log4j
Bundle-SymbolicName: symbolic.Name
```

# Anhang D. Fragen und Antworten

**D.1.** Ich habe ein Modul als abgeleitete Klasse einer JavaBean erzeugt. Die automatisch generierte GUI zeigt aber nur die Properties, die direkt in meinem Modul definiert sind - nicht die der Basisklasse. Was kann ich tun?

Das ist zunächst erst einmal völlig korrekt: die automatische Erstellung der Bedienoberflächen oder Parameter-Dialoge in dWb+ findet die Properties des jeweiligen Moduls über Introspektion heraus. Dabei werden nur die Properties der jeweiligen Klasse betrachtet und alle Properties eventuell vorhandener Basisklassen ignoriert.

Dieses Verhalten kann auf verschiedene Art und Weise geändert werden. Eine Möglichkeit wäre es, die entsprechende Property in der BeanInfo als zur Modulklasse gehörend zu markieren. Das ist aber unkomfortabel und fehleranfällig.

Einfacher ist es, dem Framework, das für die Generierung der Bedienoberfläche verantwortlich ist, mitzuteilen, bei welcher Klasse die Introspektion stoppen soll - Voreinstellung ist die direkte Basisklasse. Diese Angabe macht man am besten im static-Bereich der Klasse wie folgt:

```
import de.netsysit.util.beans.InterfaceFactory
import de.netsysit.dataflowframework.modules.ModuleBase

 static
 {
  InterfaceFactory.setSuperclassAssociationForEventDispatchThread(
   MainModul.class,ModuleBase.class);
 }
```

Diese Anweisung sorgt dafür, dass die Properties aller Basisklassen der Klasse MeinModul in der generierten Bedienoberfläche berücksichtigt werden. Diese Berücksichtigung endet jedoch mit der Klasse ModuleBase - deren Properties finden ebenso wie die aller darüber noch zu findenden Basisklassen nicht mehr zur generierten Bedienoberfläche.

**D.2.** Ich habe ein Modul erzeugt. Dieses hat zwar Properties, ich möchte aber nicht, dass diese zur Manipulation über die automatisch erzeugte GUI zur Verfügung stehen. Was muss ich tun?

Dafür gibt es zwei Möglichkeiten

1. Markieren aller Properties in der Beaninfo mit der Eigenschaft hidden.

2. Implementieren des Interface VisualComponentProvider im Paket de.netsysit.dataflowframework.ui und Rückgabe eines leeren Containers als visuelle Komponente

Beide Vorgehen sorgen dafür, dass das Modul über keinen Parameter-Dialog verfügt.

**D.3.** Ich habe ein Modul erzeugt. Dieses hat Methoden, die als Properties erkannt werden. Wie kann ich dafür sorgen, dass sie nicht in der automatisch erzeugten Bedienoberfläche auftauchen?

Dazu muss die Eigenschaft hidden in der BeanInfo für die entsprechende Property auf true gesetzt werden. Eine alternative Möglichkeit wäre es, die Property gar nicht erst in die BeanInfo-Klasse aufzunehmen.