

Integrierte Dienstentwicklung mit AUTOFOCUS² und OSGI
(Integrated Development of Service Based Systems with
AUTOFOCUS² and OSGI)
- SEP -

Michael Aichner

Florian Huber

14. Juli 2005

Vorwort

Diese Arbeit basiert auf anderen Arbeiten und Grundlagen, die im Rahmen des Projekts MEWADIS entstanden und entwickelt wurden. Daher empfiehlt es sich folgende Arbeiten zu lesen bzw. zum Nachschlagen bei Unklarheiten zu Rate ziehen:

- Diplomarbeit von Jürgen Steurer [Ste04]
- Ausarbeitung des SEP01 [MPS04]
- Ausarbeitung des SEP03 [Dem04]

Weitere Grundlagen sind das Metamodell von AUTOFOCUS² und die Arbeitsergebnisse aus dem Projekt MEWADIS.

Zuerst wollen wir uns bei unserem Betreuer Guido Wimmel bedanken. Des weiteren wollen wir dem gesamten MEWADIS-Team danken, das uns mit Ideen und Antworten auf Fragen bei den regelmäßigen Team-Meetings unterstützt hat. Abschließend möchten wir noch dem kompletten AUTOFOCUS-Team und insbesondere Tobias Hain und Oscar Slotosch (Validas AG) danken, die Fragen bzgl. AUTOFOCUS² umfangreich beantwortet und Probleme schnell gelöst haben.

Inhaltsverzeichnis

1	Einführung	3
2	Grundlagen	5
2.1	Adaptiver Dienst	5
2.1.1	Dienst	5
2.1.2	Adaptivität	6
2.2	OSGi	6
2.2.1	OSGi-Framework	6
2.2.2	Bundles	6
2.3	AUTOFOCUS ²	7
2.4	Service Management Framework	10
2.5	OSGiTool	10
2.6	ESG	10
3	Modell	13
3.1	DisplayScenario	14
3.2	Konventionen für das Modell	14
3.2.1	Aufbau der Komponenten	14
3.2.2	Anwendungsdienst	15
3.2.3	Erstellen der DTDs	16
3.3	Optimiertes Modell	16
3.3.1	Aufbau des Modells in AUTOFOCUS ²	16
3.3.2	Flache Hierarchien	16
3.3.3	Typdefinitionen	18
4	Konzept	19
4.1	Modellierungsprozess	19
4.2	Deploymentvariante	19
4.2.1	Gewonnene Vorteile	21
4.2.2	Kriterien für die Erstellung von Deploymentvarianten	21
4.3	Dynamisches Einbinden von Diensten	21
4.3.1	Ansatz zur Realisierung	22
4.3.2	Probleme	23

5	OSGiBundleEditor	25
5.1	Vorarbeit	25
5.2	Installation des Editors	25
5.3	Start	26
5.4	Grundlegendes zum Editor	26
5.5	Erstellung einer Deploymentvariante	27
5.6	Informationsangaben im Eigenschaften-Fenster	29
5.6.1	Deploymentvariante	29
5.6.2	Bundle	31
5.6.3	Service	33
5.7	Informationen im Modell	36
5.8	Installation auf ESG/SMF-Simulator	37
6	Codegenerierung	39
6.1	Analyse	39
6.2	Ablauf - Überblick - Anpassungen	39
6.3	Codeanpassungen	43
6.4	Fertige Bundles	44
6.4.1	Packagename	44
6.4.2	Aufbau eines Bundles	44
7	Ausblick	47

Zusammenfassung

Im Rahmen dieses SEPs wurde für AUTOFOCUS² ein umfangreiches Plugin, der OSGi-BundleEditor, entwickelt. Dieser Editor ermöglicht dem Entwickler adaptiver Dienste¹, Java-Code für das OSGi-Framework (Open Services Gateway Initiative) aus einem AUTOFOCUS²-Modell zu erstellen.

Mit Hilfe des OSGiBundleEditors werden die Komponenten eines dienstbasierten Modells um Deployment-relevante Informationen angereichert und flexibel auf verschiedene Bundles verteilt. Dadurch können z.B. Verkaufspakete für spezielle Zusatzdienste im Automobil realisiert werden. Somit wurde die Erstellung von OSGi-konformen Bundles zunehmend automatisiert und ein weiterer Schritt zur werkzeugunterstützten, modellbasierten Entwicklung von Diensten gemacht.

Außerdem wurden durch das SEP der neu eingeführte Plugin-Mechanismus von AUTOFOCUS² und die Import-Funktion von Modellen aus der Vorgängerversion ausführlich getestet und Verbesserungen angestoßen.

¹In dieser Arbeit werden die Begriffe Dienst und *Service* synonym verwendet.

Kapitel 1

Einführung

Der Einsatz von Software im Fahrzeug ist seit Jahren üblich. Allerdings fand sich diese bisher vornehmlich in den klassischen eingebetteten System (z.B. zur Motor- oder Airbagsteuerung). Mittlerweile setzen die Hersteller jedoch zunehmend auch auf Softwareunterstützung bei hoch komplexen und stark vernetzten Systemen, wie der Mensch-Maschine-Schnittstelle (Man Machine Interface, MMI). Das MMI ist ein Softwaresystem, durch das der Fahrer auf verschiedene Funktionen, u. a. Klimaanlage, Navigationssystem oder Telefon, zugreifen kann. Wir sprechen hierbei von *multifunktionalen Systemen* [DGPW04], deren Funktionen im Unterschied zu den klassischen Anwendungen in engen Beziehungen zu einander stehen. Darüber hinaus treten die Funktionen auch in unterschiedlichen Kontexten auf und müssen der Situation entsprechend agieren (Adaptivität). In dieser Art erweiterte Funktionen bezeichnen wir als *Dienst* [DGPW04].

Aktuelle Entwicklungsmethoden bieten allerdings keine ausreichenden Vorgehensweisen, um derart komplexe Dienste zu modellieren. Das Projekt MEWADIS verfolgt deshalb die Erweiterung von bestehenden Methoden, wie beispielsweise dem klassischen Wasserfallmodell oder dem Unified Software Development Process zur Analyse, Modellierung und Validierung von zuverlässigen, adaptiven und kontextbezogenen Diensten. Dabei fokussieren wir uns auf dynamisch austauschbare Dienste in Fahrzeugen und insbesondere des MMI.

Das Systementwicklungsprojekt „Integrierte Entwicklung von Fahrzeugdiensten mit AUTOFOCUS und OSGi“ entwickelt für das verwendete CASE Tool AUTOFOCUS² ein Plugin. Dieses ermöglicht dem Entwickler eines Dienstes, aus dem AUTOFOCUS²-Modell Java-Code für das OSGi-Framework zu erstellen. Somit wird die Generierung von OSGi-konformen Code weitgehend automatisiert und ein weiterer Meilenstein zur automatisierten modellbasierten Entwicklung von Diensten erreicht. Dabei wurden die erarbeiteten Konzepte und methodischen Grundlagen aus dem MEWADIS-Projekt benutzt und erweitert.

Diese Arbeit ist in mehrere Teile untergliedert. Anfangs wird auf die Aufgabenstellung und Ziele des SEPs eingegangen. Anschließend werden grundlegende Punkte noch einmal zusammengefasst und erläutert, speziell das Modell DisplayScenario (Kap. 3). Daraufhin wird das erarbeitete Konzept vorgestellt (Kap. 4), welches den Begriff der Deploymentvarianten einführt. Die Umsetzung des Konzepts wird in den darauf folgende Kapiteln 5 und 6 genauer betrachtet. Nachdem in Kapitel 6.4 ausführlich auf das Resultat der Bundlege-

nerierung, dem OSGi-Bundle, eingegangen wird, schließt ein Ausblick (Kap. 7) mit Ideen für kommende Arbeiten dieses Ausarbeitung ab.

Aufgaben und Ziele

Zu Beginn der Arbeit wird kurz erläutert, welche Ziele und Aufgaben durch dieses Systementwicklungsprojekts (SEP) realisiert werden sollten ¹. Zusammenfassend ergeben sich durch die Aufgabenstellung ergeben sich folgende Ziele ².

- Weiterentwicklung der Werkzeugunterstützung (Kap. 5)
- Für das Deployment relevante Informationen in die dienstbasierten Modelle integrieren (Kap. 5.7)
- flexible Deploymentvarianten entwickeln - besonders:
 - beispielsweise die Abbildung mehrerer Dienste auf einzelne Deploymentseinheiten in OSGi
 - und die Unterstützung des Austauschs von Diensten

(Kap. 4, Kap. 5)

¹Die Ausschreibung der Arbeit lässt sich im mewadis-Ordner des CVS unter `Studienarbeiten/SEP-Ausschreibung/SEP_Erweiterung_Codegenerierung.pdf` einsehen.

²In Klammern sind die Kapitel angegeben unter denen man die Bearbeitung nachlesen kann

Kapitel 2

Grundlagen

Dieses Kapitel liefert zunächst eine Zusammenfassung von Begriffen, die für das Verständnis dieser Ausarbeitung notwendig sind. Da einige grundlegende Begriffe, z.B. der des Dienstes, auch in anderen Kontexten vorkommen können, werden hier Definitionen festgelegt wie sie im Rahmen von MEWADIS bzw. dieses Systementwicklungsprojekts verstanden werden. Nachdem die Begrifflichkeiten geklärt sind, werden die in diesem Systementwicklungsprojekt eingesetzten Werkzeuge kurz vorgestellt.

2.1 Adaptiver Dienst

Um diesen Begriff zu klären, wird zunächst das Verständnis eines Dienstes und anschließend die Adaptivität erläutert. Dabei wird sich auf die Darstellungen bezogen, wie sie detailliert in [DGPW04] zu finden sind.

2.1.1 Dienst

Das Projekt MEWADIS beschäftigt sich mit der Modellierung von Funktionen und speziell von multifunktionale Systemen im Automobil. Die Besonderheit dieser Funktionen liegt in ihren Interaktionen, Abhängigkeiten und den unterschiedlichen Kontexten, in denen sie verwendet werden können. Anschaulich kann das an der Funktion der Lautstärkenregelung des Radios gemacht werden: diese kann nicht nur durch den Fahrer, sondern auch durch eingehende Telefonanrufe oder den Navigationsdienst lauter bzw. leiser gestellt werden.

Ein Dienst bezeichnet nun die Einheit einer Funktion - also sein Verhalten - mit Ihrer

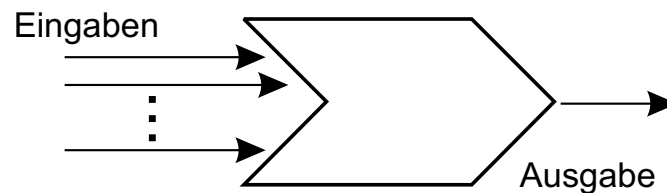


Abbildung 2.1: Schematische Darstellung eines Dienstes

Kapselung, Abhängigkeiten und Kombination. Dienste werden in `AUTOFOCUS`² als Komponenten ohne Unterkomponenten modelliert. Dabei stellt deren Automat das Verhalten des Dienstes dar.

2.1.2 Adaptivität

Adaptivität lässt Dienste in ihrer Funktion „den Umständen entsprechend agieren“. „Umstände“ werden hierbei als der Kontext bezeichnet, in dem sich ein Dienst befindet:

- vorhandene Diensterbringer: Ein Dienst zeigt unterschiedliches Verhalten, je nachdem, welche Diensterbringer er nutzt.
- Dienstanwender: Gegenüber Nutzern seiner Funktionalität soll ein Dienst auch ein adaptives Verhalten zeigen (z.B. liefert ein Navigationsdienst nur grobe Richtungsanweisungen an die Sprachausgabe, eine Karte mit detaillierten Informationen an das Multi-Informations-Display).
- Umgebungseigenschaften: Diese erhält ein Dienst als Eingabewerte über Sensoren und muss ihnen entsprechend reagieren (z.B. Einschalten der Scheinwerfer, wenn in einen Tunnel gefahren wird).

Neben der Adaptivität durch Kontext existiert noch die Adaptivität durch Lernen, wie es in [DGP05] und [Hol05] teilweise umgesetzt wurde (z.B. durch die Erweiterung der Dienste um lernende Rule Engines).

2.2 OSGi

Die (Open Services Gateway Initiative) Alliance ist ein Konsortium aus Unternehmen, Regierungsorganisationen und Universitäten. Sie hat sich die Spezifikation und Verbreitung einer Hardware-unabhängigen Plattform zum Ziel gesetzt, auf der Anwendungen und Dienste ausgeführt werden können. Diese Plattform kann auf allen Arten vernetzter Geräte eingesetzt werden und wird in MEWADIS als Framework für die Entwicklung der Prototypen im Automobil genutzt.

2.2.1 OSGi-Framework

Ein OSGi-Framework ist eine offene, modulare und skalierbare Plattform auf Java-Basis, welche die Auslieferung und Installation von Diensten, von Informationen und multimedialen Unterhaltungsinhalten ermöglicht. Ein bedeutendes Merkmal der Service Platform ist die Möglichkeit, dynamisch und kontrolliert Service-Anwendungen (sog. Bundles, siehe 2.2.2) zur Laufzeit einspielen und wieder entfernen zu können. Die Installation von Bundles im Framework bezeichnet man auch als *Deployment*, ein Begriff, der in dieser Arbeit häufig Verwendung findet.

2.2.2 Bundles

Die Grundeinheit des OSGi-Frameworks bilden die Bundles. In diesen Jar-Archiven werden die Anwendungen und Dienste einem definierten Aufbau entsprechend [OSGi] verpackt und

können dann auf einer Implementierung des Frameworks installiert werden.

Jedes Bundle kann einen oder mehrere Dienste beinhalten, die über Schnittstellen ihre Funktionalität anderen Bundles zur Verfügung stellen. Man kann Bundles im Grunde als Komponenten ansehen, die in sich wieder verschiedene Dienste kapseln. In Abbildung 2.2 wurden beispielsweise die Dienste des DisplayScenarios in die drei Bundles *Basisdienste*, *Gui* und *Multimedia* aufgeteilt.

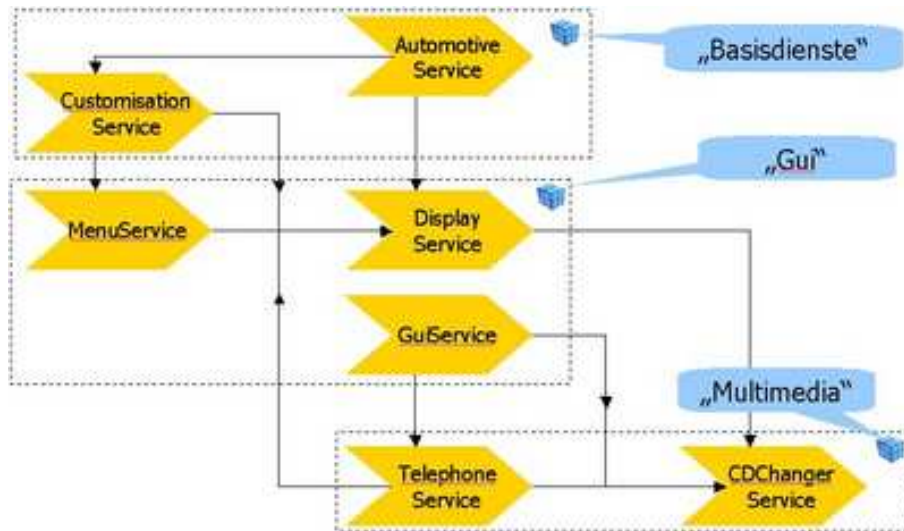


Abbildung 2.2: Bundles über dem DisplayScenario (durch gestrichelte Linie dargestellt)

2.3 AUTOFOCUS²

AUTOFOCUS² [AF2] ist ein CASE-Tool, welches am Lehrstuhl Broy entwickelt wurde. Es stellt eine Weiterentwicklung von AUTOFOCUS dar. Dieses graphische Werkzeug (s. Abb. 2.3) basiert auf formalen mathematischen Methoden (Focus) und unterstützt den Entwurf bzw. die Spezifikation verteilter und eingebetteter Systeme. Durch die Beschreibung mittels formaler Modelle ermöglicht AUTOFOCUS² die Systemeigenschaften von Diensten zu beweisen und bereits während des Entwicklungsprozess deren Verhalten zu simulieren. Dadurch unterstützt es den Entwickler bei der Lokalisierung von Fehlern in der Spezifikation. Durch den Codegenerator der Validas AG kann der Entwickler aus einem AUTOFOCUS²-Modell Java Code zu erzeugen, der später die Grundlage für die OSGi-Bundles bilden wird. Für die Modellierung adaptiver Dienste werden vor allem folgende Diagramme benutzt:

Systemstrukturdiagramm (SSD). Ein SSD besteht aus Komponenten und beschreibt zum einen, welche Ein- und Ausgabeports eine Komponente besitzt und zum anderen, über welche Kanäle diese miteinander Nachrichten austauschen können. In Abbildung 2.4 ist ein Diagramm für den *GuiService* und den *MenuService* dargestellt. In unserem Kontext wird jede Komponente in AUTOFOCUS² als Dienst angesehen. Zusätzlich gibt es in jedem Modell eine Hauptkomponente, die das Gesamtsystem spezifiziert, also das Zusammenspiel der Dienste innerhalb diesen Modells beschreibt.

Zustandsübergangsdiagramm (STD). Jeder Dienst wird durch sein dynamisches Verhalten charakterisiert, welches in den Zustandsübergangsdiagrammen (State Transi-

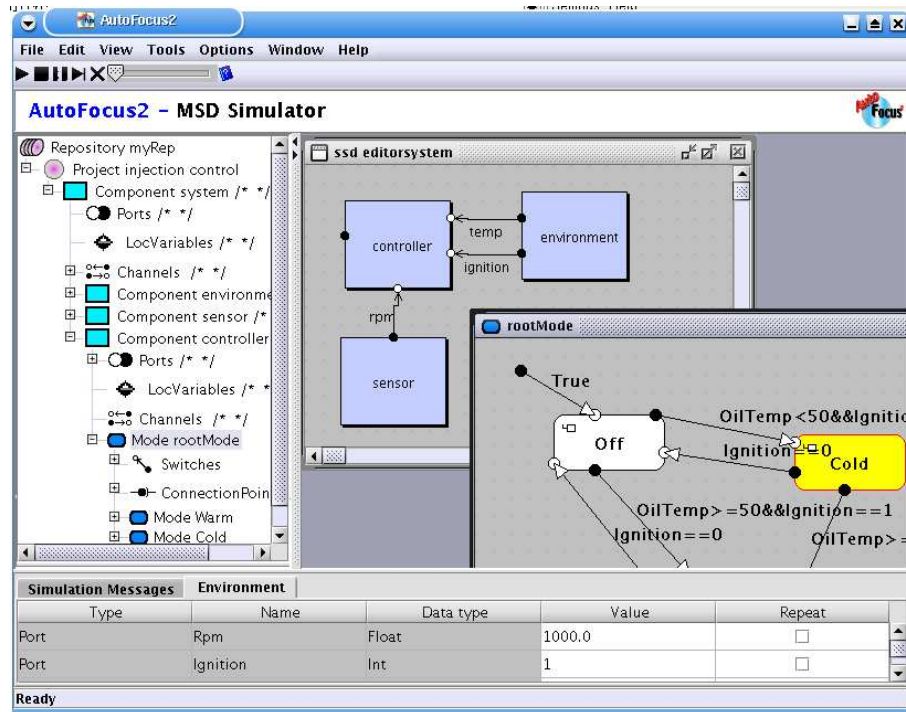
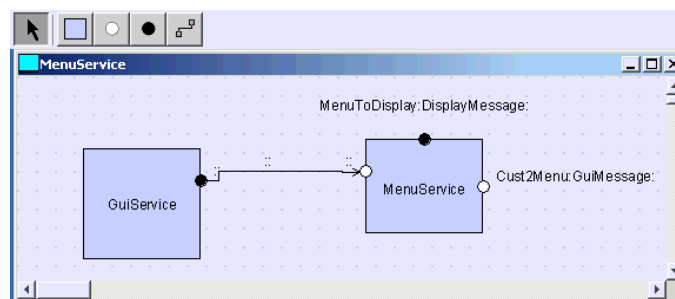
Abbildung 2.3: Oberfläche AUTOFocus²

Abbildung 2.4: Systemstrukturdiagramm des MenuService (modifiziert)

tion Diagrams, kurz: STD, siehe Abb. 2.5) als endlicher Automat dargestellt wird. Ein STD erzeugt abhängig vom momentanen Zustand und einer Eingabe aus der Umgebung (z.B. durch angebundene Kanäle) eine resultierende Ausgabe, die wiederum an die Umgebung gegeben wird.

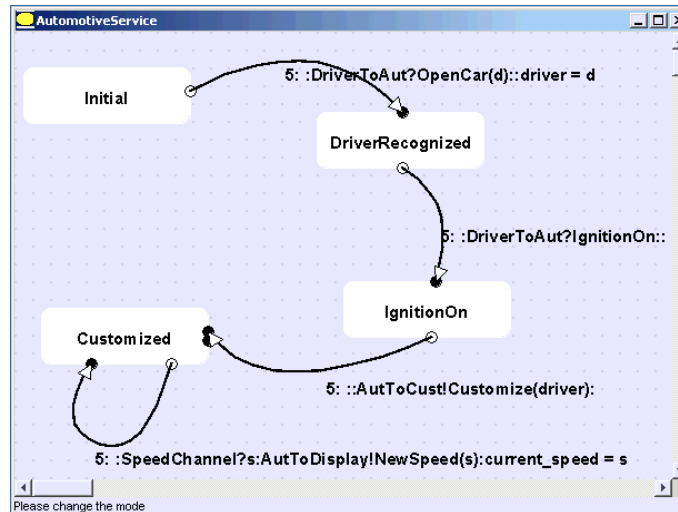


Abbildung 2.5: Zustandsübergangsdiagramm des AutomotiveService

Datentypdefinition (DTD). Mit diesen Dokumenten können Datentypen definiert werden, die den lokalen Variablen der Komponenten, den Ein- und den Ausgabeports zugrunde liegen. Diese Dokumente können auch Funktionen definieren, um den Wert eines Typs abhängig von bestimmten Parametern festzulegen. Der Editor für DTDs ist in Abb. 2.6 zu sehen.

```

import AutomotiveServiceDTD;

data TSignal = Present deriving EQ ;
data TPrio = None | Low | Med | High deriving EQ ;
data DisplayMessage = DisplayRequest(String, String, getPrio: TPrio) deriving EQ ;

fun override(High,x) = True
  | override(Med,Low) = True
  | override(Med,None) = True
  | override(Low,None) = True
  | override(x,y) = False;

fun midmsg(DisplayRequest(s,l,p),Speed150) = s | midmsg(DisplayRequest(s,l,p),curspd) = 1;
fun cdsmsg(DisplayRequest(s,l,p),curspd:TSpeed) = 1;
  
```

Abbildung 2.6: Editor für DTDs (hier: DisplayServiceDTD)

Metamodell von AUTOFOCUS². Ein wichtiger Unterschied zu AUTOFOCUS liegt in dem Aufbau des Metamodells und dementsprechend der Repräsentation im Navigationsbaum der Modelle: die frühere Version war Diagramm-orientiert, das heisst, dass jeweils

die SSDs, die STDs und die DTDs in einem gemeinsamen Ordner lagen. Dadurch war es insbesondere nicht möglich, eine DTD eindeutig einem Dienst zuzuweisen. [Dem04] löste dieses Dilemma durch die Einführung der Konvention, dass sich der Name einer DTD immer aus dem ihr zugewiesenen Service und dem Postfix *DTD* zusammensetzt (z.B. die DTD des Service *MenuService: MenuServiceDTD*).

Das Metamodell von AUTOFOCUS² ist komponentenorientiert, was für die Modellierung in unserem Fall den Vorteil bietet, dass nun die DTDs einem Dienst eindeutig zugewiesen werden können.

Anmerkung: Allerdings befand sich AUTOFOCUS² während dieses SEPs noch in der Entwicklungsphase und stellt noch nicht alle notwendigen Features zur Erstellung eines Modells zur Verfügung. Deswegen wird das Modell momentan noch in AUTOFOCUS erstellt und nach AUTOFOCUS² exportiert. In diesem Schritt wird allerdings nicht berücksichtigt, ob ein Service eine eigene DTD hat und die DTDs somit jeder Komponente unterstellt werden. Aus diesem Grund besteht die Konvention bzgl. der DTDs auch innerhalb dieses SEPs.

2.4 Service Management Framework

Das Service Management Framework (SMF) ist eine Implementation von IBM, welche auf der Spezifikationsversion 3 von OSGi basiert. Das SMF kommt momentan im Automotive Systems Lab (ASL) von MEWADIS auf der ESG (siehe Abschnitt 2.6) zum Einsatz und läuft hier auf der J2ME Implementation J9 von IBM. Zu Simulationszwecken kann das SMF auch lokal installiert werden, wobei es hierbei auf einer Standard JVM läuft. Ein genaue Anleitung zu Installation, sowohl auf der ESG, als auch lokal ist dem Anhang von [MPS04] zu entnehmen.

Das SMF stellt über die Kommandozeile u.a. die Befehle bereit, um Bundles zu installieren, zu starten, zu stoppen und zu deinstallieren.

2.5 OSGiTool

Das OSGiTool der BMW CarIT bietet eine komfortable, graphische Oberfläche, um die Kommandozeilenbefehle eines SMF bequem auszuführen. Dabei kann es auch auf SMF Installationen zugreifen, die nicht lokal installiert sind.

Die genaue Bedienung und Installation kann in [MPS04] nachgeschlagen werden.

2.6 ESG

Die ESG ist die zentrale Recheneinheit des Aufbaus im Automotive Systems Lab (siehe Bild ??). Auf ihr laufen die OSGi-Implementierung SMF, die Java Runtime (J9) und die Services. An der ESG sind zusätzliche Geräte, z.B. ein CD-Wechsler, ein Multi-Funktions-Display und ein Ergocommander angeschlossen. Durch diesen Aufbau kann das multimediale MMI (Mensch-Maschine-Interface) eines Automobils simuliert werden, auf dem prototypisch die entwickelten Services laufen.

Da die ESG über eine Netzwerkschnittstelle verfügt, kann man sie über eine statische IP erreichen. Nachdem das SMF auf der ESG über eine Telnet-Verbindung gestartet wurde,

lassen sich über das OSGiTool die gewünschten Services deployen. Es ist allerdings zu beachten, dass in dem SMF der ESG z.T. andere Basisdienste als auf der lokalen Implementierung installiert und gestartet werden müssen (Unterschiede sind im Anhang von [MPS04] erläutert).

Kapitel 3

Modell

Im Rahmen von MEWADIS wurde ein dienstbasiertes Modell erstellt, das im folgenden als *DisplayScenario* bezeichnet wird. Die Modellierung erfolgte mit Hilfe des Tools AUTOFOCUS.

An diesem Modell sollen die Konzepte für die Modellierung und Vorgehensmethoden umgesetzt und erprobt werden, die während der Projektlaufzeit erarbeitet wurden und werden. Ziel ist es, mit Hilfe von AUTOFOCUS bzw. AUTOFOCUS² ein *Rapid Prototyping* zu erreichen, welches ein schnelles Deployment der Dienste im OSGi-Framework ermöglicht.

Abbildung 3.1 stellt das Modell des DisplayScenarios dar. Die Services sind im Modell durch die Kästen dargestellt, welche über Kanäle miteinander kommunizieren. Ein Kanal setzt sich aus einem Ein- und einem Ausgabeport, die über einen gemeinsamen Datentypen (definiert in den DTDs) verfügen.

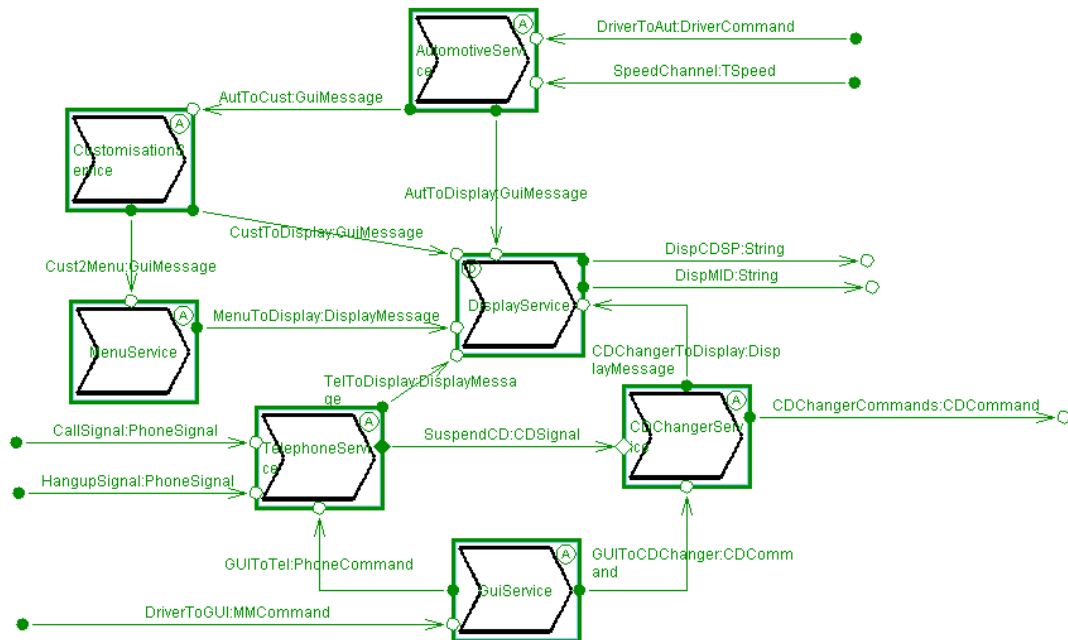


Abbildung 3.1: Das Modell DisplayScenario

Kapitelaufbau Dieses Kapitel ist in zwei Abschnitte aufgeteilt:

1. In Abschnitt 3.2 wird beschrieben, welche grundlegenden Prämissen bei der Erstellung eines Modells in AUTOFOCUS beachtet werden müssen, und wie das Modell in AUTOFOCUS² zum Zeitpunkt dieses Systementwicklungsprojekts aufgebaut war. Hier gestaltete sich der Modellierungsprozess noch derart, dass das Modell in AUTOFOCUS erstellt und anschließend in die neue Version des CASE-Tools importiert wurde. Dies ist damit begründet, dass die Arbeiten an der Benutzeroberfläche von AUTOFOCUS² noch nicht abgeschlossen waren.
2. In Abschnitt 3.3 werden die Punkte erläutert, die sich ändern werden, sobald AUTOFOCUS² eine reibungslose Modellierung anbietet und oben beschriebener Workaround entfallen kann. Das hieraus entstandene Modell bietet aufgrund des neuen Metamodells einige Optimierungspotentiale.

3.1 DisplayScenario

Das Modell erfüllt den Zweck, den im Display angezeigten Text adaptiv zu den Kontexten Geschwindigkeit und Telefonanruf zu verändern. So werden z.B. der Titel eines Lieds der aktuell abgespielten CD bei höheren Geschwindigkeiten auf die Tracknummer reduziert und bei einem eingehenden Anruf eine entsprechende Meldung auf dem Display ausgegeben.

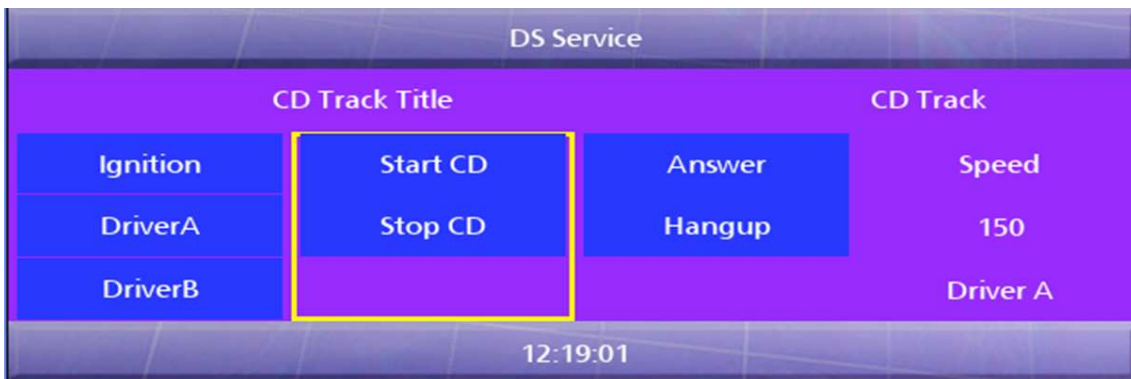


Abbildung 3.2: Ausgabe des DisplayScenario auf dem Multifunktionsdisplay

3.2 Konventionen für das Modell

Es werden nun einige Regeln für die Modellierung festgelegt, um die anschließende Codegenerierung (siehe Kapitel 6) zu ermöglichen. Die schon durch vorige SEPs aufgestellten Prämissen wurden in dieser Arbeit erweitert, um die automatisierte Erstellung von Java-Code zu erleichtern.

3.2.1 Aufbau der Komponenten

In AUTOFOCUS² wird ein Dienst durch eine Komponente beschrieben, die einen Automaten definiert, welcher das eigentliche Verhalten des Services bestimmt.

Die einzelnen Services eines Modells werden über einen sogenannten Anwendungsdienst (siehe Abschnitt 3.2.2) in Beziehung zueinander gesetzt. Durch den Import des Projekt aus AUTOFOCUS ergibt sich für ein Projekt der in Abbildung 3.3 erkennbare Aufbau.

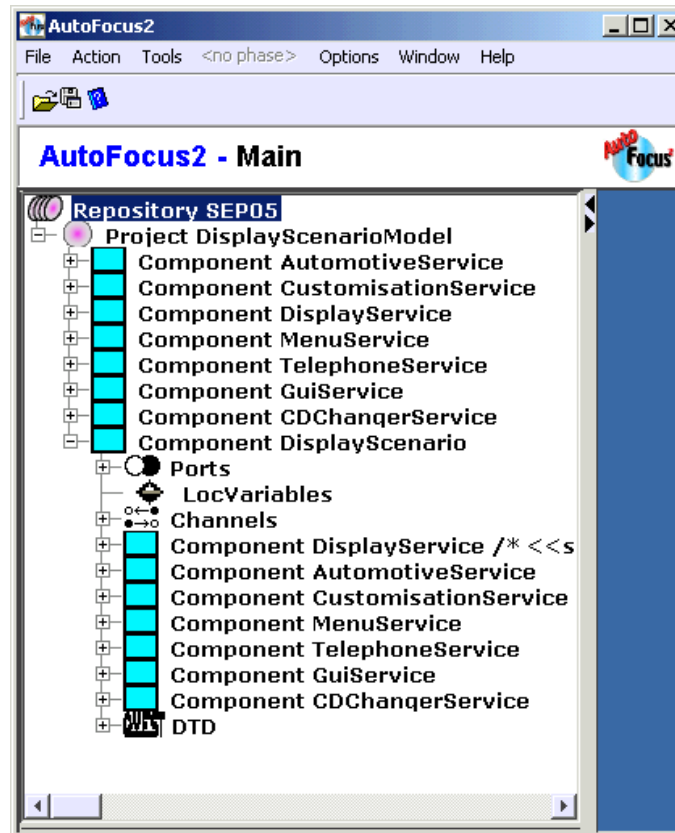


Abbildung 3.3: Aufbau eines Projekts, importiert aus AF

Die Komponenten, welche die Dienste beschreiben, sind dem Anwendungsdienst als Subkomponenten untergeordnet. Dieser könnte globale Typdefinitionen (DTDs) festlegen, die z.B. die Typen der Kommunikationskanäle festlegen. Von dieser Lösung wird allerdings abgeraten, da dadurch die einzelnen Dienste nicht mehr unabhängig sind.

Besser ist der Ansatz, dass jeder Dienst seine eigenen DTDs festlegt. Sollen zwei Dienste miteinander kommunizieren, muss der Kanal und die entsprechenden Ports an den Komponenten, auf demselben Typ arbeiten. Dazu muss ggf. eine Komponente die DTDs der anderen importieren.

3.2.2 Anwendungsdienst

Der Anwendungsdienst definiert die Kommunikationskanäle eines Modells. Er stellt also das Dach über den einzelnen Diensten dar. In obigem AUTOFOCUS²-Modell ist dies der Service mit der Bezeichnung *DisplayScenario*. Er stellt im Grunde keinen Dienst nach unserer Definition dar, sondern setzt lediglich die Dienste zueinander in Beziehung.

Dieser Umstand muss auch bei der Generierung des Java Codes des Anwendungsdienstes berücksichtigt werden. Da er als eine hierarchische Komponente die eigentlichen Dienste kapselt und deren Kommunikation darstellt, werden während der Codegenerierung spezielle

Methoden erstellt, die dies steuern. Um diesen speziellen Service zu markieren wurde deswegen die Konvention festgelegt, dass seine Bezeichnung immer mit dem Postfix *Scenario* enden muss.

3.2.3 Erstellen der DTDs

Diese werden nach der in [Dem04] bereits eingeführten Konvention benannt: So muss sich der Name für die DTD eines Services aus seiner Bezeichnung und dem Postfix *DTD* zusammensetzen (z.B. für den Service *CDChangerService* muss die DTD *CDChangerServiceDTD* lauten).

Diese Konvention begründet sich, wie bereits früher erklärt (siehe Abschnitt 2.3), in der Art und Weise der Modellierung von Diensten in AUTOFOCUS.

Typdefinitionen

Typdefinitionen werden innerhalb der DTDs erstellt. Hierbei ist dabei darauf zu achten, dass jede Typdefinition eine globale, also über das ganze Modell hinweg, eindeutige Bezeichnung hat. Der Grund hierfür liegt darin, dass nach der Codegenerierung und bei der Zusammenstellung der Bundles nicht eindeutig festgestellt werden kann, welcher Typ verwendet werden soll. In AUTOFOCUS² erfolgt keine qualifizierte Zuweisung eines bestimmten Typs zu einem Port oder einer lokalen Variablen. Abhilfe könnte hierzu eine qualifizierte Benennung der Typdefinitionen leisten.

3.3 Optimiertes Modell

Sobald eine gänzliche Loslösung von AUTOFOCUS erreicht wurde, können von einigen der obigen Prämissen Abstand genommen werden, da sich u.a. durch das neue Metamodell auch Vorteile für die Codegenerierung ergeben.

3.3.1 Aufbau des Modells in AUTOFOCUS²

Hier ist zu beachten, dass in AUTOFOCUS² für jedes Dienstszenario (auch Modell genannt) ein eigenes Projekt erstellt werden muss. Im Unterschied zum bisherigen Modell, bildet in diesem der Anwendungsdienst die einzige Root-Komponente, welche die anderen Dienste kapselt (schematisch in Abb. 3.4). Allerdings zeigten sich bis zum bisherigen Zeitpunkt Schwächen in der Bedienbarkeit der Modell-Editoren, womit die Erstellung eines konsistenten Modells nicht möglich war.

3.3.2 Flache Hierarchien

In der bisherigen Version des DisplayScenarios beinhaltet der Dienst *DisplayService* weitere Subdienste (vgl. Abb. 3.5). Dies entspricht jedoch nicht der Definition eines Dienstes, wie sie in [DGPW04] gegeben wird. Die Kapselung mehrerer Services innerhalb eines Dienstes entspricht einer Komponente.

Deswegen wird die Prämisse aufgestellt, dass ein Service keine weiteren Dienste in sich kapselt und somit jede Komponente in AUTOFOCUS² genau einem Dienst entspricht.

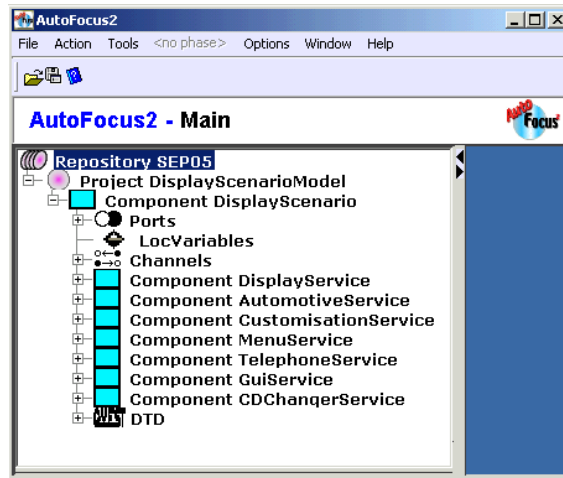


Abbildung 3.4: Sauberer, hierarchischer Aufbau eines Projekts

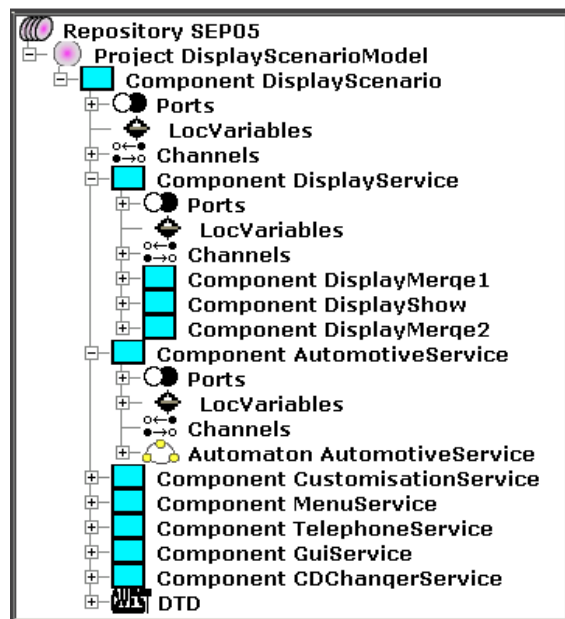


Abbildung 3.5: Dienst mit Substruktur

Anmerkung: Sollte sich trotzdem die Notwendigkeit ergeben, eine Art „Superdienst“ zu entwickeln, kann dies immer noch erreicht werden. Dazu müssen die zu bündelnden Dienste nur zusammen in ein Bundle gepackt werden. Dieses Vorgehen ist in Kapitel 5 beschrieben.

3.3.3 Typdefinitionen

Dank des neuen Metamodells kann nun jedem Service seine DTD (bzw. in `AUTOFOCUS2`: `DTDModul`) eindeutig zugeordnet werden. Dadurch wäre ist nicht mehr notwendig, dass die Bezeichnung des `DTDModul`s der Konvention aus dem Abschnitt 3.2.3 folgt. Dadurch wird auch die Schwäche des Imports vermieden, dass jedem Service alle `DTDModule` zugeordnet sind.

Anmerkung: Grundsätzlich hätte man dies auch schon in der momentanen Version des `OSGiBundleGenerators` umsetzen können. Allerdings wäre es dann notwendig gewesen, das Modell per Hand anzupassen und die während des Imports zuviel erstellten DTDs unter einem Service zu entfernen.

Kapitel 4

Konzept

Die SEPs im Rahmen von MEWADIS basieren auf dem eben erklärten Beispielszenario von *Diensten*, dem DisplayScenario. Dieses wurde mit Hilfe von AUTOFOCUS modelliert.

4.1 Modellierungsprozess

Bislang sah der Prozess vom Modell in AUTOFOCUS bis zum einsatzfähigen OSGi-Bundle so aus, wie in Abbildung 4.1 aufgezeigt.

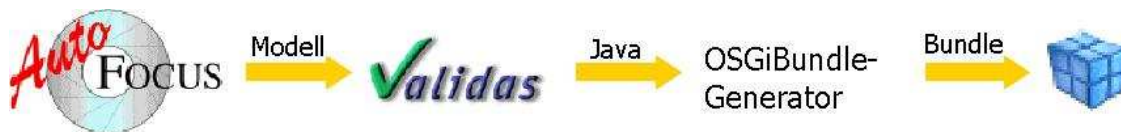


Abbildung 4.1: Deploymentprozess vor diesem SEP

Es wurde ein Modell in AUTOFOCUS erstellt. Dieses wurde in das Tool Validator der Validas AG exportiert, der einen Generator zur Erstellung von Java-Code anbietet. Unter Verwendung des OSGiBundleGenerators (siehe [Dem04]) wurden die generierten Java-Klassen aufbereitet, bereinigt und den Diensten entsprechend geordnet, d.h. dass die notwendigen Klassen eines Dienstes zusammengefasst wurden. Unter Verwendung von zusätzlichen Informationen aus separat erstellten Dateien wurde in einem letzten Schritt für jeden Dienst ein OSGi-Bundle erzeugt.

Gemäß der Aufgabenstellung, die wir bereits in Kapitel 1 besprochen haben, war es Ziel, die Toollandschaft zu verkleinern und den Deploymentprozess auf AUTOFOCUS² anzupassen. Anschaulich beschreibt das die Abbildung 4.2.

4.2 Deploymentvariante

Es gibt mehrere Möglichkeiten, die Zuordnung Dienst zu Bundle zu realisieren. Bislang waren die Möglichkeiten begrenzt, da nur alle Dienste in ein Bundle gepackt werden konnten, oder jeder Dienst einzeln in sein Bundle. Da dies verschiedene Varianten des Deployments sind, führten wir zum besseren Verständnis den Begriff der **Deploymentvariante** ein.



Abbildung 4.2: Deploymentprozess nach diesem SEP

Für ein Szenario, das n Dienste hat, besteht eine Deploymentvariante aus $1 \dots n$ Bundles, die wiederum aus $1 \dots n$ Diensten bestehen können, wobei die Bundles bezüglich der Dienste disjunkt sein müssen. Dies hat unter anderem den Hintergrund, dass dadurch auch nach der Codegenerierung und der Anpassung der generierten Dateien die Referenzen eindeutig sind.

In Abbildung 4.3 und Abbildung 4.4 sind die bisher vorhandenen Deploymentvarianten schematisch dargestellt. Abb. 4.5 zeigt eine mögliche, flexible Zusammenstellung von Bundles, die durch dieses SEP ermöglicht wird.

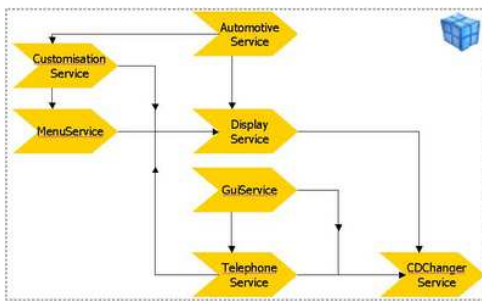


Abbildung 4.3: Deploymentvariante 1: „All in one“

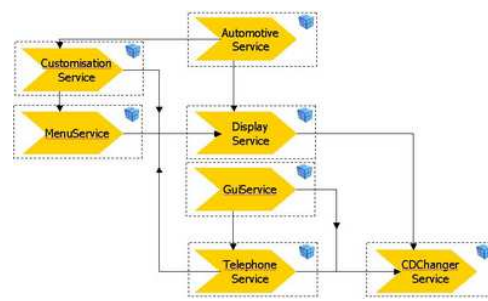


Abbildung 4.4: Deploymentvariante 2: „Ein Dienst - ein Bundle“

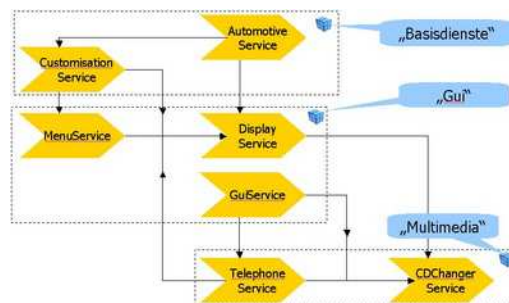


Abbildung 4.5: Deploymentvariante 3: „flexible Bundles“

Bemerkung: Zusätzlich zu diesen Diensten gibt es noch einen übergeordneten Dienst DisplayScenario, der die Kommunikation zwischen den anderen Diensten steuert. Dieser kann auch in ein beliebiges Bundle gepackt werden.

4.2.1 Gewonnene Vorteile

Nachdem jetzt nicht nur die Deploymentvarianten „All in one“ und „Ein Dienst - ein Bundle“ möglich sind, sondern auch weitere flexible Deploymentvarianten, können jetzt die Nachteile der vorherigen Deploymentvarianten behandelt werden und auch Vorteile für die flexiblen Deploymentvarianten abgeleitet werden.

Wollte man früher Dienste A bis N zusammenfassen, um diese dann beispielsweise auf dem OSGi-Framework mit einer Schnittstelle anzubieten, so musste man ein eigenes Modell erstellen. Bei diesem wurden dann die Dienste A bis N unter einer Superkomponente zusammengefasst, die dann die Dienste A bis N auf oberster Scenarioebene ersetzt. Somit musste also das Modell modifiziert werden.

Durch die flexible Zusammenstellung von Diensten in Bundles ist die Modellierung völlig unabhängig vom Deployment. Hierdurch muss man also zum Zeitpunkt der Modellierung noch nicht wissen, in welcher Form das Modell in Bundles aufgeteilt werden soll.

4.2.2 Kriterien für die Erstellung von Deploymentvarianten

Im Folgenden werden mögliche Kriterien behandelt:

Logische Bundles: Die Zusammenstellung kann, wie in Abb. 4.5 gezeigt, zum Beispiel nach funktionalen Gesichtspunkten erfolgen. Hierbei werden ähnliche und funktional zusammenhängende Dienste in ein Bundle gepackt. Im Fall von Deploymentvariante 3 „flexible Bundles“ sind dies die Bundles Gui, Multimedia und Basisdienste.

Verkaufspakete: Analog zu dem Bundle Multimedia aus Deploymentvariante 3 kann man bei größeren Szenarien, die sich auf die Dienste in einem Automobil beziehen, ähnliche Pakete erstellen. Dies könnten ein Hifi-Bundle, Navi-Bundle, Telephone-Bundle usw. sein. Das bedeutet, dass diese Bundles auch gleichzeitig als Verkaufspakete betrachtet werden können.

Hardwaregebundene Bundles: Vor allem im Automobil kann es sinnvoll sein, die Bundles nach der Hardwarezugehörigkeit der Dienste zusammenzustellen. Die Dienste, die die Funktionen des gleichen Steuergerätes benötigen, bilden somit ein Bundle.

Minimierte Schnittstellen: Zu den Überlegungen, nach welchen Gesichtspunkten man die Dienste gruppiert, können auch Entwicklungsaspekte kommen. Beispielsweise, dass ein Zulieferer ein bestimmtes Bundle liefern soll, das wiederum im gesamten Szenario als Zusammenfassung von Diensten fungieren soll. Auch im Zuge der verteilten Entwicklung von Diensten eines Szenarios könnte man nicht nur logische Dienste zusammenfassen, sondern auch versuchen die Schnittstellen möglichst gering zu halten.

4.3 Dynamisches Einbinden von Diensten

In [MPS04] wurde als Schwäche der momentanen Code- bzw. Bundlegenerierung angeführt, dass der erzeugte Code direkt auf den Implementierungen anderer Services arbeitet. Dies entspricht grundsätzlich nicht dem Schlüsselgedanken einer serviceorientierten Architektur, in der nur über Schnittstellen auf andere Services zugegriffen werden sollte. Dadurch würde erreicht werden, dass Dienste zur Laufzeit installiert, gestartet, gestoppt, deinstalliert und

ausgetauscht werden könnten. Man wäre also in der Lage, jederzeit Dienste bzw. deren Verhalten neu zu modellieren und beliebig erneut zu installieren.

In dieser Arbeit wurde untersucht, in wie weit der generierte Code um Schnittstellenklassen ergänzt werden müsste, um obiges Ziel umzusetzen.

4.3.1 Ansatz zur Realisierung

Grundsätzlich wird dem theoretischen Lösungsansatz von [MPS04] zugestimmt, dass durch eine Anpassung der Codegeneration die Interfaces automatisch erstellt werden könnten. Der *InstanceManager*, der Teil des Bundles *ServiceBinder* ist, bietet die Funktionalität, um Interfaces zu Instanzen eines Dienstes dynamisch zu binden (ausführlich wird dies in [CH04] und [Ste04] beschrieben). Vereinfacht funktioniert dieser Mechanismus wie in Abbildung 4.6 dargestellt:

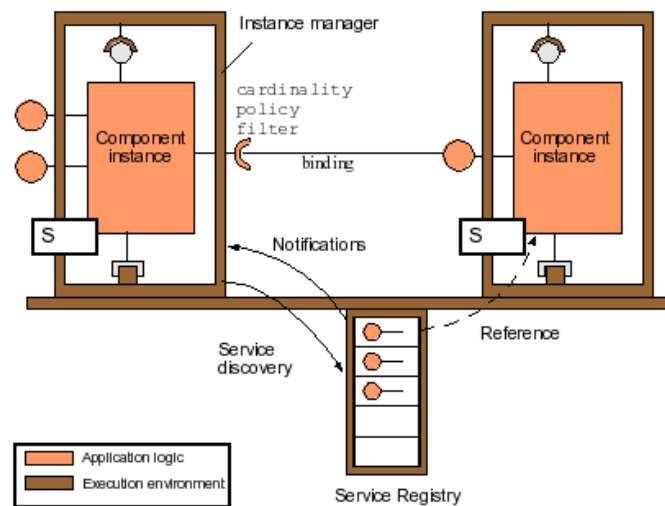


Abbildung 4.6: Dynamisches Binden von Diensten

Ein Bundle A (*Component instance* auf der rechten Seite der Abb. 4.6) meldet die von ihm angebotenen Schnittstellen beim *InstanceManager* an, welcher einen Eintrag in der zentralen *Service Registry* anlegt. Soll nun ein anderer Dienst gestartet werden, überprüft der *InstanceManager*, ob die benötigten Dienste in der *Service Registry* vorhanden sind und bindet diese, falls verfügbar, an den Dienstanbieter.

Anmerkung: Nachdem aber die Services aus dem AUTOFOCUS²-Modell nicht nur andere Dienste, sondern z.T. auch deren Typdefinitionen verwenden, müssen für diese auch Schnittstellen erstellt werden.

Änderungen für Dienstanbieter

Dies würde für die Anpassungen am generierten Code des Dienstanbieters zur Folge haben, dass

1. für die Klasse, welche den Automaten des Dienstes repräsentiert und

2. für jede Typdefinition, deren *Events*-, *Ports*-, *Interf*-Klassen ¹

jeweils ein eigenes Interface erstellt wird. Es ist nicht ausreichend, nur für die Klasse, die den Automaten darstellt ein Interface anzubieten. Da das Ziel eine völlige Unabhängigkeit von jeglichen Implementierungen ist, muss auf die benutzten Typdefinitionen auch über Schnittstellen zugegriffen werden.

Die angebotenen Schnittstellen müssen zusätzlich im Instance Descriptor aufgeführt werden.

Änderungen für Dienstanutzer

Jede Klasse des Dienstanutzers, welche eine externe Schnittstelle benötigt, muss nun um die notwendigen *bind*- und *unbind*-Methoden erweitert werden. Diese binden bzw. lösen die Implementierungen der fremden Dienste, Typdefinitionen usw. von den notwendigen Schnittstellen. Folgendes Beispiel zeigt eine *bind*-Methode, die ein Objekt der Klasse *CDCCommandListener* an die Variable *cdCommandListener* bindet.

```
public void setCDCCommandListener(CDCCommandListener ccl)
{
System.out.println("CDPlayer set " + ccl.toString());
cdCommandListener = ccl;
}
```

Dabei ist hervorzuheben, dass für jede notwendige Schnittstelle diese Methoden zu erstellen sind. Zusätzlich muss der Instance Descriptor des Dienstanutzer um diese Interfaces erweitert werden.

4.3.2 Probleme

In den angestellten Überlegungen ergaben sich nun zwei Hauptprobleme bei der Realisierung obiger Umsetzung. Deswegen wird empfohlen, dieses Thema in einer eigenen Arbeit zu bearbeiten und basierend auf den angegebenen Ansätzen, Lösungskonzepte zu erstellen. Dies wird auch eine Änderung an der momentanen Version des *ServiceBinders* erfordern, was den Rahmen dieses SEPs überschritten hätte.

Vielzahl an Schnittstellen

Ein Problem stellen die bereits erwähnten vielseitigen Abhängigkeiten zwischen den Diensten, speziell in Betracht auf die definierten Typen, über welche die Kommunikation erfolgt² dar. Somit würde es keinesfalls ausreichen, nur ein Interface für den Dienst an sich anzubieten. Es wären auch noch Interfaces für jede Typdefinition, Events usw. notwendig, was mit umfassenden Änderungen an der bestehenden Codegenerierung verbunden ist und die erstellten Klassen um ein Vielfaches vergrößert (pro notwendigem Interface zwei zusätzliche Methoden).

¹Es werden pro Typdefinition fünf weitere Klassen erzeugt

²Anzumerken ist, dass nicht nur eine Java Klasse für jeden definierten Typ erstellt wird, sondern zusätzlich noch weitere Klassen (z.B. Events auf diesen Typen) generiert werden, deren Implementierungen die anderen Dienste auch benutzen

Fehler im ServiceBinder

Es wurde während der Analyse auch festgestellt, dass die im ASL verwendete Version des *ServiceBinders* (Ver. 1.0) ein einwandfreies dynamische Binden und Nachinstallieren nicht unterstützt. Wie bereits beschrieben, wird jede angebotene Schnittstelle in der Service Registry abgelegt. Ein Dienst kann diese nun nutzen, indem der *InstanceManager* diese an den Service bindet.

Beispielsweise benötigt ein Dienst *ExtendedCDChanger* den Grunddienst *CDPlayer*, welcher über eine Schnittstelle zur Verfügung steht. Dazu muss zunächst der *CDPlayer* installiert und gestartet werden, womit sein Interface in der Service Registry vorliegt. Durch den Eintrag im Instance Descriptor des *ExtendedCDChanger* wird bei dessen Start der *CDPlayer* an ihn gebunden.

Stoppt man nun den Dienst *CDPlayer*, erfolgt automatisch ein Aufruf der unbind-Methode des *ExtendedCDChanger*. Ein erneuter Start feuert erneut die bind-Methode. Nun ergibt sich allerdings ein Problem, falls der Service *CDPlayer* im Framework deinstalliert wird, um z.B. eine Änderung an seinem Modell vorzunehmen. Dadurch wird der eindeutige Eintrag in der Service Registry entfernt. Bei einem erneuten Installieren und Starten des Services wird er zwar wieder in der Registry aufgenommen. Jedoch fehlt die zuvor bestehende Assoziation mit dem Dienstanbieter und dementsprechend wird die bind-Methode des *ExtendedCDChanger* nicht aufgerufen.

Zur Realisierung eines vollständigen dynamischen Deployments müssten also auch Erweiterungen am bestehenden *ServiceBinder* bzw. dessen *InstanceManager* vorgenommen werden (z.B. indem die aufgebauten Assoziationen zwischen Dienstanbieter und -nutzer zur Runtime persistent gespeichert werden).

Kapitel 5

OSGiBundleEditor

Mit dem OSGiBundleEditor lassen sich bequem Deploymentvarianten und OSGi-Bundles erstellen. Hierzu werden benötigte Daten in die entsprechenden Felder eingetragen und anschließend über die Codegenerierung die fertigen Bundles erstellt. Diese können dann auf dem Simulator oder der ESG ausgeführt werden. Die Verwendung des Editors und der Ablauf des Deploymentprozesse, sowie die notwendigen Informationen, die für die Bundlegenerierung benötigt werden, werden in diesem Kaptiel ausführlich beschrieben.

5.1 Vorarbeit

Um Deploymentvarianten und entsprechende lauffähige Bundles mit Hilfe des OSGiBundleEditors erstellen zu können, müssen folgende Vorarbeiten erbracht werden. Einerseits muss ein AUTOFOCUS²-Modell erstellt werden, andererseits muss ein sogenannter Glue Code¹ erstellt werden um die Verbindung zur Hardware und zur GUI herzustellen.

Nach den Vorarbeiten, ist nun das Erstellen der Deploymentvarianten mit Hilfe des OSGi-BundleEditors möglich ist.

5.2 Installation des Editors

Das beispielsweise mit dem Buildtool *Ant* erzeugte Jar-File `OSGiBundleGenerator.jar`² wird im AUTOFOCUS²-Verzeichnis `lib/plugins/` abgelegt.

Des weiteren ist das Jar-File, wie es zum Ende des SEPs vorlag, auch unter

`mewastud/SEP_05/fertige_Dateien/`

abgelegt und muss dann nur in den entsprechenden plugin-Ordner unter AUTOFOCUS² kopiert werden.

Des weiteren benötigt der OSGiBundleEditor die Library `TOOLS.JAR` (ab Java-Version 1.5 bzw. 5.0), die bei jedem installierten SDK im `lib`-Verzeichnis liegt. Diese muss in das `lib`-Verzeichnis von AUTOFOCUS² kopiert werden. Damit ist die Kompilierung der generierten Dateien aus dem Editor gewährleistet.

¹Die Funktionsweise und die Aufgaben des Glue Codes ist in [MPS04] ausführlich dargestellt.

²Hierzu kann das make-File `build_osgi.xml` genutzt werden, welches im Projektordner dieses SEPs zu finden ist

5.3 Start

Der OSGiBundleEditor wurde als Plugin für AUTOFOCUS² entwickelt. Daher muss zuerst AUTOFOCUS² gestartet und ein Szenario erstellt bzw. geladen werden. Wir haben hierfür das DisplaySzenario benutzt, das man im CVS unter

```
mewastud/SEP_05/fertige_Dateien/displayScenario_SEP05.xml
```

findet. Ist dieses nun in AUTOFOCUS² geladen, so kann man über das Kontext-Menü (linke Maustaste) des Punkts *Project* das Plugin starten. Hierzu wählt man den Punkt *OSGi-BundleEditor* (vgl. Abb. 5.1).

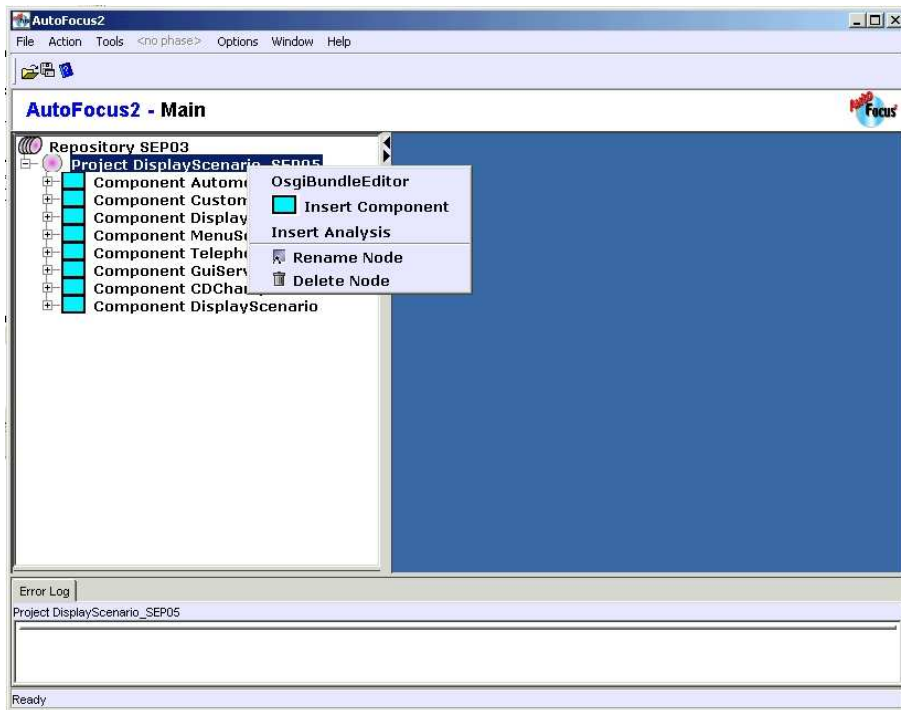


Abbildung 5.1: Aufruf des OSGiBundleEditors aus AUTOFOCUS²

5.4 Grundlegendes zum Editor

Der OSGiBundleEditor ist in drei große Bereiche eingeteilt:

- **Deploymentvarianten-Fenster.** Im linken Teil wird ein Baum entstehen, dessen Wurzel den Namen des AUTOFOCUS²-Projekts trägt. Darunter wird dann als Kind eine Deploymentvariante hinzugefügt, darunter ein Bundle, und darunter wiederum ein Dienst
- **Dienste-Fenster.** Im rechten Teil sind die Komponenten des AUTOFOCUS²-Modells aufgelistet, die dann als Dienste den Bundles im Deploymentvarianten-Fenster hinzugefügt werden können

- **Eigenschaften-Fenster.** Unten im OSGiBundleEditors findet sich ein Eigenschafts-Fenster, in dem dann je ausgewähltem Element im Baum die Informationen für die Bundle-Generierung eingetragen werden können



Abbildung 5.2: Startbild des OSGiBundleEditors

5.5 Erstellung einer Deploymentvariante

Zu beachten ist hierbei die in Kap. 4.2 getroffene Konvention der disjunkten Bundles. Um eine Deploymentvariante erstellen zu können, muss man die Vorarbeiten (siehe Kapitel 5.1) leisten. Bevor man mit dem Erstellen der Deploymentvarianten beginnt, sieht das Fenster des OSGiBundleEditors so wie in Abb. 5.2 aus.

Deploymentvariante hinzufügen

1. Im Deploymentvarianten-Fenster auf die Wurzel des (noch leeren) Baumes klicken
2. Im Menü *Action* den Punkt *Add Deploymentvariant* wählen

3. Dann den Namen der Deploymentvariante eingeben (dieser darf keine Leerzeichen besitzen und muss mit einem Buchstaben beginnen)

Dies lässt sich beliebig oft wiederholen, je nachdem wie viele Deploymentvarianten abgespeichert werden sollen.

Bundle hinzufügen

1. Im Deploymentvarianten-Fenster die entsprechende Deploymentvariante auswählen, der das Bundle zugeordnet werden soll.
2. Nun im Menü *Action* den Punkt *Add Bundle* auswählen.
3. Dann den Namen des Bundles eingeben. (Dieser darf keine Leerzeichen besitzen und muss mit einem Buchstaben beginnen)

Auch dieser Schritt lässt sich für jedes Bundle wiederholen. Allerdings macht es keinen Sinn, mehr Bundles als Dienste anzulegen, da in zwei Bundles einer Deploymentvariante nicht der gleiche Dienst enthalten sein darf. (4.2)

Dienst zuordnen

1. Im Deploymentvarianten-Fenster das Bundle auswählen, dem der Dienst hinzugefügt werden soll
2. Nun im Dienste-Fenster den Dienst auswählen, den man dem Bundle hinzufügen will
3. Anschließend im Menü *Action* den Punkt *Add Service* auswählen

Nun ist der Dienst dem Bundle zugeordnet. Auch dieser Schritt lässt sich wiederholen, wobei aber zu berücksichtigen ist, dass die Bundles innerhalb einer Deploymentvariante disjunkt sein müssen (siehe Konvention disjunkter Bundles oben)

5.6 Informationsangaben im Eigenschaften-Fenster

Hier werden Schritt für Schritt die einzelnen Eigenschaften der entsprechenden Bauelemente beschrieben. Die mit einem Stern * gekennzeichneten Elemente sind zwingend notwendig, um die Lauffähigkeit des Bundles auf einem OSGi-Framework zu gewährleisten.

Alle Informationen können mit einem Doppelklick in der rechten Tabellenspalte geändert werden. Zum Beenden des Bearbeitens bitte die ENTER-Taste drücken.

Im folgenden wird in Klammern der Wert angegeben, der beim DisplayScenario bei der einfachsten Deploymentvariante „All in one“ (alle Dienste werden in ein Bundle gepackt) in dem Feld steht.

5.6.1 Deploymentvariante

name * Name der Deploymentvarianten; der Name darf keine Leerzeichen enthalten.
(All_in_one)

author Name des Autors der Deploymentvariante (Michael Aichner, Florian Huber)

email E-mail-Adresse des Autors für eventuelle Rückfragen (mewadis@in.tum.de)

Die Abb. 5.3 zeigt den Editor mit den eingetragenen Informationen.

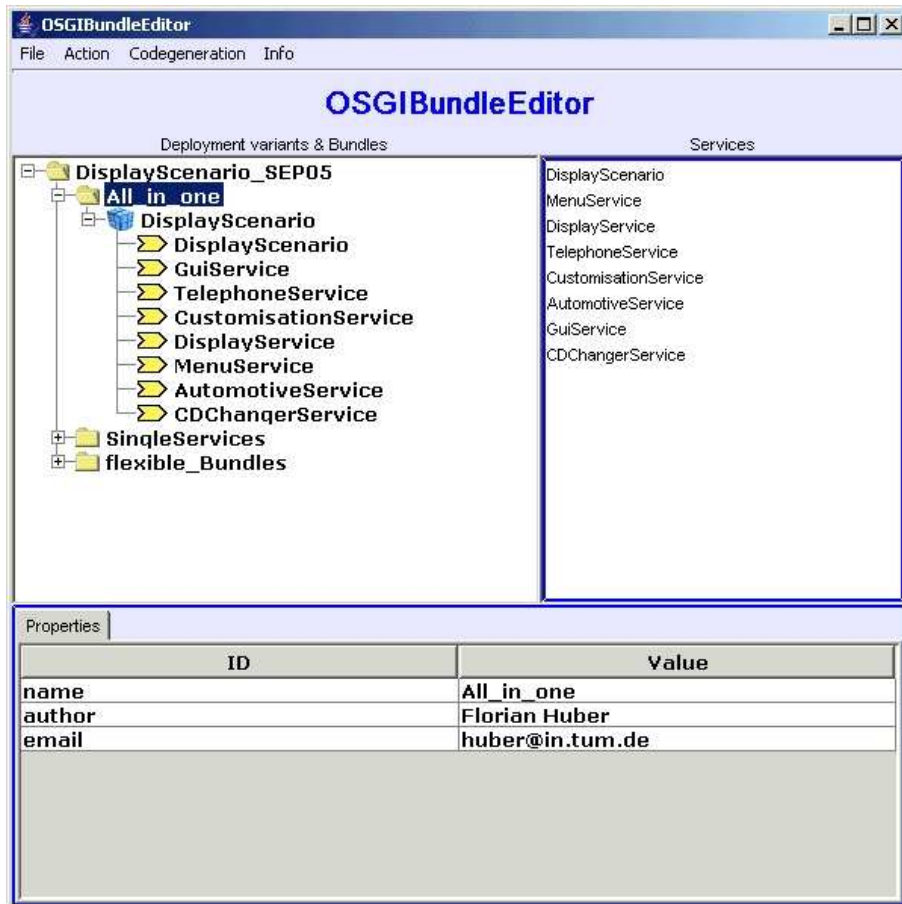


Abbildung 5.3: Eigenschaften-Fenster bei Deploymentvariante 1

5.6.2 Bundle

Registerkarte *Properties*

name * Name des Bundles (all_services)

classpath * Angabe externer Jar-Archive, die benötigt werden (absoluter Pfad - z.B. osgi-2.0.jar, serviceinterfaces-tum-0.1.jar, classicwt-appmanager-1.0.jar, classicwt-1.0.jar, tumInterfaces-0.1.jar, servicebinder-0.2.jar, ewt-1.0.jar
Hierbei ist zu beachten, dass nach dem ersten Generieren der Deploymentvariante (Kap. 6), die jeweils anderen Bundles (inkl. baseclasses.jar) dem classpath dieses Bundles hinzugefügt, also in dieses Feld eingetragen, werden. Somit kann ein einzelnes Bundle dann problemlos nachträglich generiert und mit dem alten Bundle ausgetauscht werden.

Registerkarte *Manifest*

Das Manifest beschreibt den Inhalt des Bundles und gibt Informationen über das Bundle an, sowie eine kurze Beschreibung der Funktionalität. Details lassen sich unter anderem in [Ste04] nachlesen.

Die Abb. 5.4 zeigt einen Screenshot des Editors mit dem Eigenschaften-Fenster für das Manifest eines Bundles.

Im Folgenden werden ein paar Eigenschaften des Manifests kurz erklärt. Für alle anderen Felder sollte man die OSGi-Beschreibung eines Manifests zu Rate ziehen.

priority * Priorität des Bundles (Wert zwischen 1 und 255) (250)

bundle-activator * „true“, wenn ein BundleActivator, sonst erhält dieses Feld den Wert *false* (true)

Bemerkung: Dieses Feld existiert nicht in einem Manifest, sondern wurde für den internen Ablauf des OSGiBundleEditor hinzugefügt.

bundle-classpath * Der Classpath des Bundles („“)

bundle-name * Der Name des Bundles; Dieser sollte identisch sein mit dem Wert der in der Registerkarte *Properties* unter *name* eingetragen wurde (DisplayScenario)

export-package Die Packages, die dieses Bundle anbietet (Achtung! Diese werden automatisch bei der Codegenerierung erzeugt!) (-)

import-package Die Packages, die aus anderen Bundles importiert werden müssen. („Diese Angaben dienen aber nur zur Information und werden vom OSGi-Framework nicht auf Erfüllbarkeit geprüft. So obliegt die Prüfung und Auflösung der Dienstabhängigkeiten den Bundles selbst und erfolgt nicht automatisiert durch das OSGi-Framework.“) [Ste04]

(de.bmw.crisp.services.audiodiskplayer.interfaces, de.bmw.ewt, de.bmw.kylie.classic.appmanager, de.bmw.kylie.classic, de.bmw.ewt.graphics, de.tum.mewadis.interfaces.telephoneservice, org.ungoverned.gravity.servicebinder, de.tum.mewadis.interfaces.speedService, de.bmw.ewt.widgets, org.ungoverned.gravity.common, de.tum.mewadis.baseclasses.events, de.tum.mewadis.baseclasses.defaulttypes,

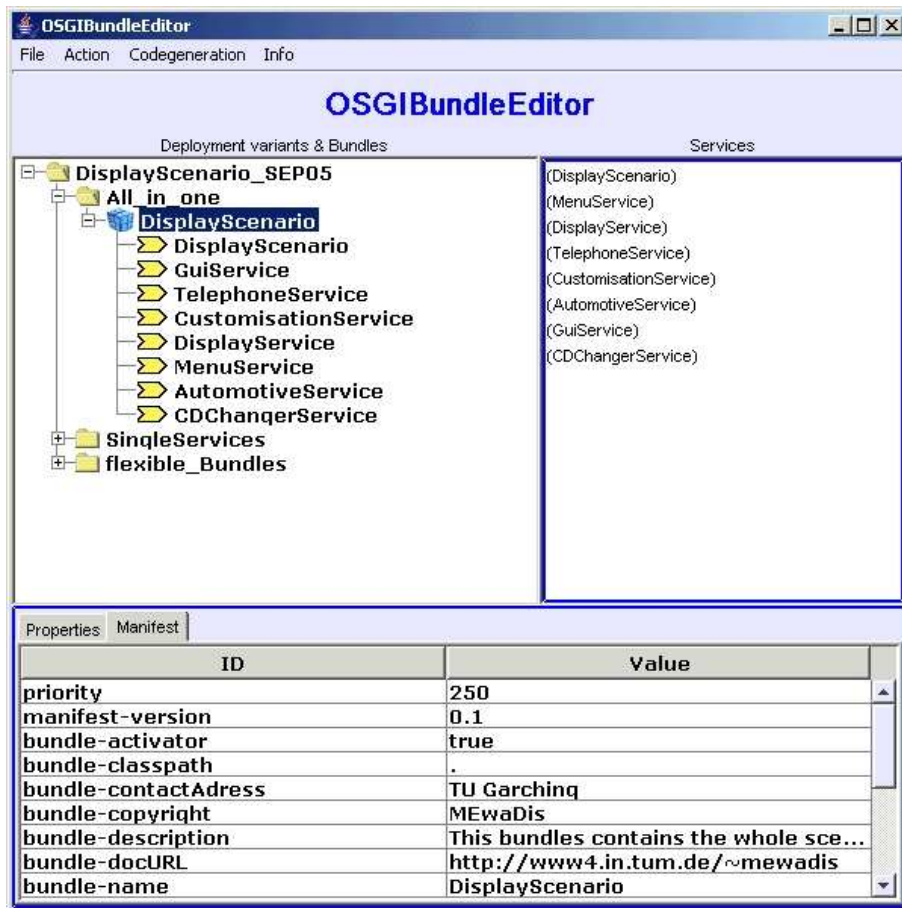


Abbildung 5.4: Eigenschaften-Fenster bei Bundle DisplayScenario in Deploymentvariante 1 - Manifest

de.tum.mewadis.baseclasses.data, de.tum.mewadis.baseclasses.common,
de.tum.mewadis.baseclasses.interf, de.tum.mewadis.baseclasses.typclasses)

metadata-location Existiert ein Bundle-Activator (und wurde dies im Feld *bundle-activator* mit „true“ vermerkt), so kommt hier der relative Pfad des Instance-Descriptors hinein. Der Instance-Descriptor (bzw. die metadata.xml) wird aus den Angaben aus der Registerkarte Instance-Descriptor eines Services automatisch erstellt und standardmäßig auf folgenden Pfad gelegt: *etc/metadata.xml*

5.6.3 Service

Die Angaben in Klammern beziehen sich auf die Angaben, die beim Dienst DisplayScenario gemacht werden.

Registerkarte *Properties*

name * Name des Dienstes; Dieser kann nicht bearbeitet werden und stimmt mit dem Namen der Komponente im AUTOFOCUS²-Modell überein. (DisplayScenario)

glue code * Absoluter Pfad zum Glue Code (KylieGUI.java)

Die Gluecodeklassen dienen später als Basis für die Erstellung der Instances im Instance-Descriptor. Deswegen muss hier für jede Gluecodeklasse ein Eintrag im Instance-Descriptor bei *Instance*, *instance class* erstellt werden. Das Problem hierbei ist, dass das Paket, in dem dieser Service letztlich landet, noch nicht bekannt ist. Deshalb werden beim Erstellen des Instance-Descriptors die Angaben zur Instanzklasse ergänzt. Der Benutzer muss also lediglich die zu instanzierende Klasse *ohne* Packageangabe eintragen (Standardmäßig lautet das Package für die Instanzklassen „de.tum.mewadis.*BundleName*.glue“).

Anmerkung: In unseren Modellen entspricht die Instanzklasse immer dem Glue Code, d.h. dass im Instance-Descriptor für den Wert des Attributes „instance class“ die Klasse des Glue Codes stehen muss.

Registerkarte *instance descriptor*

Der Instance-Descriptor ist wieder baumartig strukturiert. Die Wurzel bildet der Name des Dienstes. Die im folgenden erklärten Elemente eines Instance-Descriptors lassen sich hinzufügen, indem man einen entsprechenden Knoten markiert und im rechten Fenster des Eigenschaftens-Fensters dann den Button „add ...“ anklickt. Löschen kann man die Elemente einzeln, indem man das entsprechende Element im Baum markiert und dann im Menü *Action* und *remove from instanceDesc* auswählt (vgl. auch Abb. 5.5).

Zum Instance-Descriptor(metadata.xml): „Diese XML-Datei beschreibt die Implementierungsklassen (siehe 4.2.4) des Bundles, welche Dienste anbieten oder benötigen und somit vom ServiceBinder verwaltet werden.“ [Ste04]

Instance Die Instanz, die der Dienst implementiert; Hier ist das meistens der Glue Code. Anzugebene ist der Packagename, der sich wie in Kap. 6.4.1 zusammensetzt. (de.tum.mewadis.DisplayScenario.glue.KylieGUI)

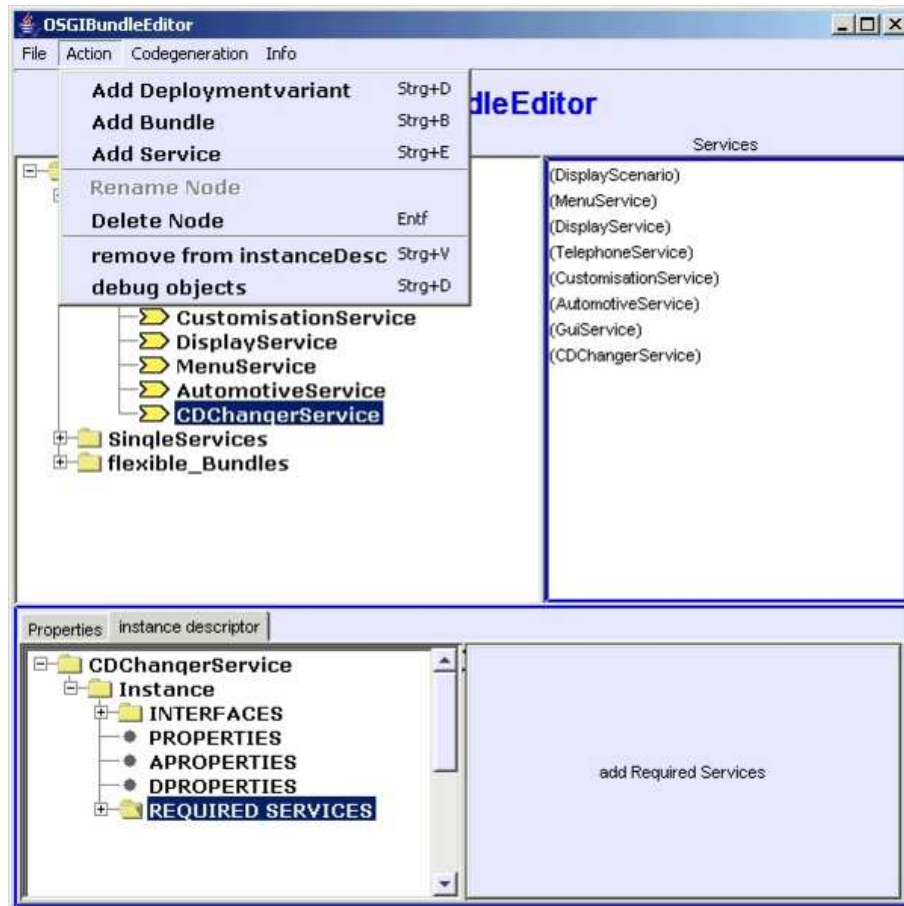


Abbildung 5.5: Aufbau des Instance-Descriptors im Editor

Interface Definiert die Schnittstelle, an der sich diese Implementierungsinstanz anmeldet. Dienste, die diese Instanz verwenden, müssen diese Schnittstelle und *Required Services* (s.u.) angeben. (-)

Properties Beschreibung der Diensteeigenschaften; nur für Implementierungsklassen, die eine Schnittstelle anbieten. (-)

AProperties „Array-Property“, das mehrere Werte besitzen kann. (-)

DProperties „Dynamic-Property“ ist eine Eigenschaft, die sich zur Laufzeit ändern kann (-)

Required Services Für jede Dienstabhängigkeit wird hier ein Eintrag benötigt.
(de.tum.mewadis.interfaces.speedService.SpeedSource,
de.tum.mewadis.interfaces.telephoneservice.Telephone,
de.tum.mewadis.DisplayScenario.events.CDCommandListener)

Für jeden benötigten Dienst gibt es noch folgende Eigenschaften

cardinality Die Kardinalität zu dem Dienst (jeweils 1..1)

rankingpolicy Der Pfad (hier immer: etc/policy.xml) zur Ranking-Policy, falls eine vorhanden sein soll, sonst einfach leer lassen (jeweils -)

bind-method Die Methode mit der man den benötigten Dienst startet (für die benötigten Dienste, in der gleichen Reihenfolge wie oben aufgelistet: setSpeedSource, setTelephone, setCDChanger)

unbind-method Die Methode mit der man den benötigten Dienst beendet. (Auch hier wieder wie bei bind-method: unsetSpeedSource, unsetTelephone, unsetCDChanger)

filter Hier kann noch ein Filter gesetzt werden (jeweils -)

policy Diese policy hat nichts mit der Ranking-policy zu tun. Sie kann entweder „static“ oder „dynamic“ sein (jeweils auf static)

Für jeden *required service* kann man eine Ranking Policy angeben.

Ranking Policies „XML-Beschreibung, welche für einen Dienstyp die Prioritäten zwischen den Eigenschaften einerseits, und zwischen den Eigenschaftsausprägungen andererseits, festlegt.“ [Ste04] (-)

Eine ausführliche Beschreibung der Elemente eines InstanceDescriptors findet sich im Kapitel 5.2.1 ServiceBinder in [Ste04]

Für einen noch besseren Einblick werden hier nocheinmal die Elemente des InstanceDescriptors des CDChangerService aufgelistet. Hierbei wird auch ein Interface implementiert und eine RankingPolicy definiert.

Name des Feldes	Eintrag
Instance	de.tum.mewadis.DisplayScenario.glue.XCDCGlue
Interface	de.tum.mewadis.DisplayScenario.events.CDCommandListener
Properties	-
AProperties	-
DProperties	-
Required Services	de.bmw.crisp.services.audioplayer.interfaces.AudioDiskPlayer
RANKING-POLICY	
cardinality	1..1
rankingPolicy	etc/policy.xml
bind-method	setCDChanger
unbind-method	unsetCDChanger
filter	-
policy	static
POLICY PROPERTY	
function	min
name	cdinfo_source
weight	1
PVALUE	
value	online
prio	0
PVALUE	
value	offline
prio	1

5.7 Informationen im Modell

Mit dem Abspeichern einer Konfigurationsdatei werden entsprechende Informationen in das AUTOFOCUS²-Modell gespeichert.

Im Kommentar des Projekts der Pfad zur aktuellen Konfigurationsdatei:

```
<<ConfigFileBundleGen=C:\temp\bundleConf_SEP03.xml>>
```

Zusätzlich werden in diesem Projektkommentar noch die vorhandenen Deploymentvarianten, die in diesem Config-File gespeichert wurden eingetragen:

```
{BundleGenRelation=All_in_one}
{BundleGenRelation=SingleServices}
\textttt{\{BundleGenRelation=flexible_Bundles}
```

Zu jeder Komponente im Modell wird noch die Zugehörigkeit zu dem Bundle gespeichert. Im Beispiel des CDChangerServices in unserem Modell sieht das folgendermaßen aus:

```
{BundleGenRelation.All_in_one=DisplayScenario}
{BundleGenRelation.SingleServices=CDChangerServiceSingle}
{BundleGenRelation.flexible_Bundles=Multimedia}
```

5.8 Installation auf ESG/SMF-Simulator

Für die Installation auf der ESG oder dem SMF-Simulator benötigt man die Konfigurationen im OSGi-Tool. Diese sind analog zu den Empfehlungen aus Anhang A aus [MPS04] zu erstellen. Konkret bedeutet das, dass die Konfigurationen *base*, *kylie* und *services* so übernommen werden können.

Die Konfiguration *apps* wird nicht übernommen, stattdessen eine eigene Konfiguration erstellt, z.B. *displayscen*

Folgende Jar-Archive müssen in die *apps*-Konfiguration des OSGi-Tools:

- extendedchanger-0.2.jar
- speedGenerator-0.1.jar
- telephoneSimulator-0.1.jar

Zusätzlich werden nun die aus dem OSGiBundleEditor entstandenen Bundles mit hineingenommen. Dies sind z.B. bei Deploymentvariante 1 „All in one“:

- baseclasses.jar
- DisplayScenario.jar

Alle müssen installiert und dann gestartet werden.

Kapitel 6

Codegenerierung

6.1 Analyse

In [MPS04] wurde bereits gezeigt, dass, basierend auf dem automatisch generierten Java-Code, ein lauffähiges Bundle zu einem Dienst erstellt werden kann. Dabei wurden noch externe Tools, z.B. der *Validator* für die Codegenerierung, benutzt. Dessen Methoden sind mittlerweile in das AUTOFOCUS²-Projekt integriert, so dass sie direkt aufgerufen werden können. Durch die Änderungen am Metamodell hat sich natürlich auch die Codegenerierung an einigen Stellen geändert.

Nach der Analyse und dem Vergleich der beiden Codes gab es zwei wesentliche Veränderungen, die Auswirkung auf unsere Arbeit hatten.

Eine Änderung ist, dass die Struktur der erstellten Dateien eine andere ist. Es kamen neue Dateien und Packages hinzu. Dies ist in den Tabellen 6.1 und 6.1 gegenüber gestellt.

Bemerkung: Die Tabellen sind folgendermaßen zu lesen:

1. Stehen in einer Zeile bei AUTOFOCUS und bei AUTOFOCUS² Dateien, so sind diese ähnlich, aber unterscheiden sich im Namen.
2. Steht nur in einer Spalte ein Dateiname, so fehlt dieser in der anderen AUTOFOCUS-Version!

Eine weitere Änderung ist, dass jetzt innerhalb der Klassen die Referenzen auf andere Klassen nicht nur mittels einer import-Anweisung aufgelöst, sondern die Klassenreferenzen mit dem voll-qualifizierten Klassennamen generiert werden. Dies machte die Anpassung des Codes (vgl. Kap. 6.3) schwieriger als ursprünglich angenommen.

6.2 Ablauf - Überblick - Anpassungen

Es gibt drei Varianten der Codegenerierung, die man aus dem OSGiBundleEditor aufrufen kann. Zum Einen kann die Codegenerierung auf einer ganzen Deploymentvariante angewandt werden (Menü *Codegeneration, Deploymentvariant*). Eine weitere Möglichkeit besteht darin nur den Code für ein Bundle zu generieren (Menü *Codegeneration, Selected Bundle*). Oder aber man erstellt nur das so genannte „baseclasses bundle“ (Menü *Codegeneration, Baseclasses Bundle*) (vgl. Kap. 6.4.2).

Tabelle 6.1: Tabellarischer Vergleich des Codes von AF und AF2(1)

Ordner	Unterschied	AUTOFOCUS	AUTOFOCUS ²
common	gleich		jede Datei um 4 - 29 byte kleiner
components	fast gleich		ASDISPLAYSCENARIO fehlt, DisplayScenario.java um ca. 200 Bytes kleiner Dateien bis zu 2000 Byte kleiner/größer
data			+ TypParser.java bei jeder DTD
defaulttypes		gibts hier nicht	AFArray.java AFChar.java AFSet.java AFString.java BoolEG.java FloatNUM.java Implementation.java IntNUM.java
events		(59 Dateien) + AFBooleanListener + AFByteListener + AFDoubleListener + AFFloatListener + AFIntegerListener + BooleanEvent + BooleanListener + ByteEvent + ByteListener + DoubleEvent + DoubleListener + FloatEvent + FloatListener + IntegerEvent + IntegerListener	(54 Dateien) + AFArrayListener + AFCharListener + AFSetListener + AFStringListener + AFArrayEvent + AFArrayListener + AFBoolEQListener + AFChatEvent + AFCharListener + AFFloatNUMListener + AFIntNUMListener + AFSetEvent + AFSetListener + AFStringEvent + BoolEQEvent + BoolEQListener + FloatNUMEvent + FloatNUMListener + IntNUMEvent + IntNUMListener

Tabelle 6.2: Tabellarischer Vergleich des Codes von AF und AF2(2)

Ordner	Unterschied	AUTOFOCUS	AUTOFOCUS ²
interf(aces)		(36 Dateien) + booleanLocVar + booleanPort + byteLocVar + bytePort + doubleLocVar + doublePort + floatLocVar + floatPort + intLocVar + intPort	(40 Dateien) + AFArrayLocVar + AFArrayPort + AFCharLocVar + AFCharPort + AFSetLocVar + AFSetPort + AFStringLocVar + AFStringPort + BoolEQLocVar + BoolEQPort + FloatNUMLocVar + FloatNUMPort + IntNUMLocVar + IntNUMPort
typclasses		gibts hier nicht!	+ DATA + EQ + NUM + ORD

Ein kurzer Überblick über den Ablauf der Codegenerierung soll Abb. 6.1 liefern.

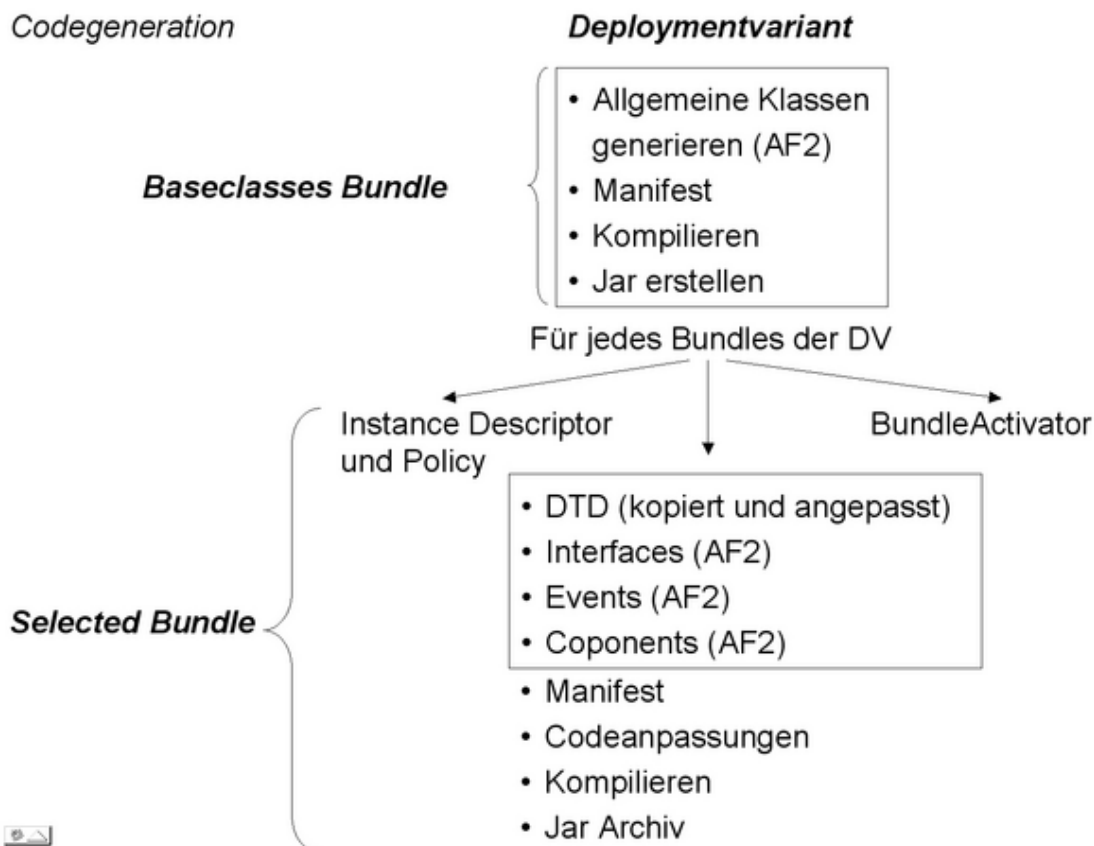


Abbildung 6.1: Schematischer Überblick zur Codegenerierung und Erstellung eines Bundles

Die Generierung einer Deploymentvariante setzt sich also aus der Generierung der Basis-klassen und der Generierung der einzelnen Bundles zusammen.

Ein einzelnes Bundle kann allerdings nicht generiert werden, ohne vorher die ganze Deploy-mentvariante generiert zu haben. Es fehlen sonst die bereits erstellten Klassen der anderen Bundles, die referenziert werden und die im Classpath zur Kompilierung liegen müssen.

Ausserdem hat dies auch konzeptionelle Hintergründe. Ein Bundle soll innerhalb einer Deploymentvariante nur dann nachgeneriert werden müssen, wenn sich etwas am Dienst verändert hat. Diese Änderung darf aber keine weiteren Dienste näher beeinflussen (z.B. Schnittstellenänderung), da sonst die Deploymentvariante mit dem Anwendungsdienst (hier: DisplayScenario) komplett neu generiert werden muss. Das heißt konkret, dass nur für ein Update eines Bundles, dieses alleine neu generiert werden darf.

Detaillierter Ablauf Ein OSGi-Bundle setzt sich aus

- den Klassen, welche die Funktionalität der Dienste darstellen (DTDs, Interfaces, Events, Components),
- ggf. den Klassen, die den Glue Code beinhalten (müssen manuell erstellt werden),

- dem Instance Descriptor (und ggf. Ranking Policies) und
- dem Manifest

zusammen (vgl. auch Kap. 6.4).

Diese Dateien werden, ausgehend von der Klasse `DeployVarGen`, in der in Abb. 6.1 angegebener Reihenfolge erstellt. Zuerst wird bei der Generierung einer Deploymentvariante das Bundle `baseclasses` erstellt. Dieses setzt sich aus den allgemeinen Klassen, die die Codegenerierung von `AUTOFOCUS2` erzeugen zusammen. Zusätzlich wird wie bei jedem Bundle ein Manifest erstellt. Anschliessend werden die Klassen kompiliert und das Jar-Archiv `baseclasses.jar` erstellt.

Nun werden für jedes Bundle der Deploymentvariante die gleichen Schritte unternommen. Besitzt ein Bundle einen Instance-Descriptor (und evtl. auch eine Policy), so werden diese erstellt. Ebenso verhält es sich mit dem `BundleActivator`.

Daraufhin werden die Klassen, die die Funktionalität beinhalten, erzeugt. Das sind DTDs, Interfaces, Events und die Components. Wie schon beim Erstellen der `Baseclasses` wird das Manifest generiert. Nun werden hier aber noch Codeanpassungen (Kap. 6.3) vorgenommen. Zuletzt, werden alle Klassen kompiliert und ein Jar-Archiv erstellt.

6.3 Codeanpassungen

Wie bereits in Abb. 6.1 ersichtlich, müssen nach der Generierung der Dateien Anpassungen am Code vorgenommen werden. Dies hat die Ursache, dass bei der standardmäßigen Codegenerierung von AF2 beispielsweise keine Packageangaben übergeben werden können. Ausserdem bilden wir ein Package mit den Standardtypen (`Baseclasses.jar`), die von der Codegenerierung zu jeder Komponente erstellt werden. Dies optimieren wir, indem wir diese nur einmal erzeugen lassen und uns dann an entsprechenden Stellen in der Codegenerierung von AF2 einhängen. Im Zuge dessen müssen dann in jeder Klasse, die diese Standardtypen verwenden, die `import`-Anweisungen angepasst werden und auch die voll qualifizierten Klassenreferenzen innerhalb der Dateien.

Die Anpassungen werden im folgenden einzeln aufgelistet und kurz erklärt.

- Anpassung am Glue Code
Hierbei wird die Package-Angabe des Glue Codes berichtigt, jenachdem in welchem Bundle (und dadurch auch Package) der Glue Code anschliessend liegen wird.
- `common`-Dateien
Standarddateien werden von der AF2-Codegenerierung in den Ordner `common` gelegt. Nachdem wir diese in das Package `baseclasses` packen, müssen die Referenzen auf diese Klassen angepasst werden.

Aus

```
„import de.tum.mewadis.componenten-name.common.Klasse“
```

wird

```
„import de.tum.mewadis.baseclasses.common.Klasse“
```

- PMException

Wie bei den common-Dateien verhält es sich mit der Klasse PMException, die ebenfalls im Package baseclasses liegen wird.

Aus

```
„de.tum.mewadis.componenten-name.data.PMException“
```

wird

```
„de.tum.mewadis.baseclasses.data.PMException“
```

- die übrigen baseclasses

Genauso verhält es sich auch mit den übrigen Referenzen auf die Klassen, die in das Package baseclasses kommen.

Aus

```
„de.tum.mewadis.componenten-name.Package-name.<Klasse aus baseclasses>“
```

wird

```
„de.tum.mewadis.baseclasses.Package-name.<Klasse aus baseclasses>“
```

wobei Package-name eines der folgenden Package-Teile ist: interf, events, defaulttypes, common, typclasses.

6.4 Fertige Bundles

In diesem Kapitel wird der Aufbau und die enthaltenen Dateien der automatisch erstellten Bundles zusammengefasst und näher erläutert.

6.4.1 Packagename

Die Bezeichnung eines Packages, welches mit dem *OSGiBundleEditor* erstellt wurde lautet standardmäßig

```
de.tum.mewadis.BundleName
```

6.4.2 Aufbau eines Bundles

baseclasses.jar Dieses Bundle enthält alle von der Codegenerierung erzeugten Standardklassen. Diese sind von keinen Komponenten abhängig und werden bei der Codegenerierung üblicherweise zu jeder Komponente erstellt. Dies wurde in unserer Codegenerierung aber optimiert und sie werden nun nur noch einfach erstellt

Es gibt in diesem Bundle 6 Packages:

de.tum.mewadis.baseclasses.

- common
- data
- defaulttypes
- events
- interf
- typclasses

BundleName.jar Alle weiteren Bundles besitzen auf höchster Ebene einen Ordner `META-INF`, in dem sich die Datei mit dem Manifest, der Bundlebeschreibung, befindet.

Besitzt ein Dienst in diesem Bundle einen Instance-Descriptor (und evtl. auch Abhängigkeitsregeln), so liegt dieser als `metadata.xml` (und `policy.xml`) im Verzeichnis `/etc`. Folgende Packages besitzt jedes Bundle: `de.tum.mewadis.BundleName`.

- `components`
- `events`
- `interf`

Besitzt ein Dienst in einem Bundle eine DTD, so liegt diese im Package

- `data`

Ein entsprechender Glue Code wird in das Package

- `glue`

gelegt.

Der BundleActivator, falls vorhanden, kommt in das Package

- `osgi`

weitere Dateien und ihre Pfade Die Pfade zum Instance-Descriptor, der Ranking-Policy, den Glue Code(s) und dem Bundle-Activator werden erst bei der Bundlegenerierung festgelegt. Es gelten jedoch folgende Konventionen:

Manifest Dieses liegt standardmäßig in dem Ordner `/META-INF` (dies ist durch den Java-Befehl `jar` festgelegt). Dateiname: `Manifest.mf`

Instance-Descriptor Der Instance Descriptor befindet sich immer im Ordner `/etc`. Diese Information muss so auch im Manifest, unter dem Attribute `Metadata-Location` gespeichert werden. Dateiname: `metadata.xml`

Ranking Policy Diese Datei oder Dateien liegen ebenfalls im Ordner `/etc` unter dem im Instance Descriptor angegebenen Namen. Dateiname: `policyName.xml`

Bundle-Activator Sollte das Bundle einen Bundle Activator haben (siehe Einstellungen in dem Eigenschaftsfenster des Bundles), wird dieser im Ordner `/PackageName/osgi` gespeichert. Da der Dateiname für alle Bundles gleich lautet, muss der Eintrag `Bundle-Activator` im Manifest auch immer `BundleActivator` lauten. Dateiname: `BundleActivator.class`

Glue Code Diese Klassen werden in dem Ordner `/PackageName/gluecode` gespeichert. Der Dateiname wird dabei so übernommen, wie ihn der Entwickler im OsgiBundle-Editor angegeben hat.

Kapitel 7

Ausblick

Während des SEPs haben sich offene Fragen ergeben und einige Probleme der momentanen Lösungen wurden entdeckt.

Wie bereits in Kapitel 4.3 „Dynamisches Einbinden von Diensten“ einige Schwachstellen des momentanen OSGi-Frameworks und des Servicebinders aufgezeigt wurden, so kann hier eine weitere Arbeit weitergehend das OSGi-Framework detailliert analysieren und einige Verbesserungen einarbeiten.

In diesem Zusammenhang kann dann auch die Codegenerierung weiter optimiert und an die Beschaffenheit des OSGi-Frameworks angepasst werden.

Was konkret die Erstellung der Deploymentvarianten angeht, so kann man über semantische Ansätze nachdenken, mit Hilfe derer man Deploymentvarianten nach bestimmten Kriterien automatisch erstellen lässt. Mögliche Kriterien sind die bereits in Abschnitt 4.2.2 vorgestellten.

Im Zuge dieser möglichen semantischen Analyse und der im Modell abgespeicherten Informationen, welcher Dienste welchem Bundle zugeordnet wurde, können vorhandene Modellverifikationen erweitert werden. Auch hier können die Kriterien berücksichtigt, evtl. genauer analysiert und bewertet werden, um aussagekräftige Argumente liefern zu können, ob gewisse Deploymentvarianten vorteilhaft bzgl. eines Kriteriums ist. Wie bereits in Abschnitt 4.2.2 erwähnt kann das Optimierungen auf dem Endgerät betreffen, wie z.B. die optimierte Platzierung der Bundles im realen Steuergeräteverbund.

Da sich AUTOFOCUS² noch in der Entwicklungsphase befindet und sich auch hier weiterhin Veränderungen und Verbesserungen abzeichnen, kann auch auf unsere Konventionen verzichtet werden. Die Benennung der Typdefinitionen wird wohl demnächst über den Type-Checker in AUTOFOCUS² abgefragt. Somit könnte eben unsere Konvention der global eindeutigen Typdefinitionen entfallen, da der Type-Checker dann einen mehrfach definierten Typ meldet.

Der OSGiBundleEditor hat auch noch Weiterentwicklungspotenzial. Hier könnte z.B. ein Einbinden des OSGiTools erfolgen.

Abbildungsverzeichnis

2.1	Dienst	5
2.2	Bundles des DisplayScenario (Beispiel)	7
2.3	Screenshot AF2	8
2.4	Systemstrukturdiagramm	8
2.5	Zustandsübergangdiagramm	9
2.6	Datentypdefinition	9
3.1	Das Modell DisplayScenario	13
3.2	Ausgabe des DisplayScenario auf dem Multifunktionsdisplay	14
3.3	Projekt-Struktur, importiert	15
3.4	Projekt-Struktur, ideal	17
3.5	Dienst mit Subdiensten	17
4.1	Deploymentprozess vor diesem SEP	19
4.2	Deploymentprozess nach diesem SEP	20
4.3	Deploymentvariante 1	20
4.4	Deploymentvariante 2	20
4.5	Deploymentvariante 3	20
4.6	Instance Management	22
5.1	Aufruf des OSGiBundleEditors	26
5.2	Startbild des OSGiBundleEditors	27
5.3	Deploymentvariante 1 - Properties	30
5.4	Deploymentvariante 1 - Manifest	32
5.5	Instance-Descriptor Screenshot	34
6.1	Codegenerierung	42

Literaturverzeichnis

- [Ste04] Steurer Jürgen, „Entwicklung eines Dienstekonzepts für das Mensch-Maschine-Interface (MMI) System eines Fahrzeugs“ Diplomarbeit, TU München, Lehrstuhl Broy, 2004
- [MPS04] Mas Y Parareda Benedikt, Schröder Gunnar, „Realisierung einer Dienst basierten Anwendung für die Mensch-Maschine-Schnittstelle im Fahrzeug“, Systementwicklungsprojekt, TU München, Lehrstuhl Broy, 2004
- [Dem04] Demir Ahmet, „Automatisierte Entwicklung dienstbasierter Anwendungen für die Mensch-Maschine Schnittstelle im Fahrzeug“, Systementwicklungsprojekt, TU München, Lehrstuhl Broy, 2004
- [DGPW04] Martin Deubler, Johannes Grünbauer, Gerhard Popp, Guido Wimmel, „Arbeitspapier Mewadis“, TU München, Lehrstuhl Broy, 2004
- [DGP05] Krasimir Dzhigovechki, Angel Gerdzhikov, Boris Pavlov, „Realisierung eines dienstbasierten, adaptiven und personalisierten Infotainment-Systems für Fahrzeuge“, Systementwicklungsprojekt, TU München, Lehrstuhl Broy, 2005
- [Hol05] Andreas Holzbach, „Entwicklung einer adaptiven Nutzerschnittstelle für den Bereich Automotive“, Systementwicklungsprojekt, TU München, Lehrstuhl Broy, 2005
- [OSGi] OSGi Alliance, 2005 – http://www.osgi.org/osgi_technology/download_specs.asp
- [AF2] AUTOFOCUS² - TU München, Lehrstuhl Broy, 2005 – <http://www4.in.tum.de/~af2>
- [CH04] Humberto Cervantes, Richard S. Hall, „Simplifying application development on the OSGi services platform“, 2004 – <http://gravity.sourceforge.net/servicebinder>