Institut für Informatik
Arbeitsgruppe Softwaretechnik
Warburger Straße 100
33098 Paderborn

Ein dynamischer Erweiterungsmechanismus für Fujaba

Studienarbeit
zur Erlangung des Grades
Bachelor of Computer Science
im Rahmen des Studiengangs Informatik

von
Sinan Alptekin
Kilianstr. 17
D-33142 Bueren

vorgelegt bei
Prof. Dr. Wilhelm Schäfer
und
Prof. Dr. Odej Kao

Paderborn, im April 2004

Erklärung

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegeben Quellen angefertigt habe, und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

--	------------------------------
Ort, Datum	Unterschrift

Inhaltsverzeichnis

1 Einleitung **1**

1.1 Motivation . . . 1

1.2 Zielsetzung . . . 3

1.3 Aufbau und Struktur der Arbeit . . . 3

2 Grundlagen und Begriffe **5**

2.1 Einführung in die Fujaba Tool Suite . . . 5

2.2 Erweiterbare Softwarewerkzeuge und Softwaresysteme . . . 6

2.3 Versions- und Konfigurationsmanagement . . . 7

3 Konzeption des dynamischen Erweiterungsmechanismus **11**

3.1 Die neue Fujaba Tool Suite System-Architektur . . . 11

3.2 Anwendungsunabhängigkeit des dynamischen Erweiterungsmechanismus . . . 13

3.3 Versionierung der Anwendung und ihrer Plug-Ins . . . 13

3.4 Lade-Mechanismus für Plug-Ins . . . 16

3.5 Update und Installation von Plug-Ins . . . 18

3.6 Fujaba GUI-Erweiterung . . . 18

4 Technische Umsetzung **21**

4.1 Schnittstellen des Erweiterungsmechanismus . . . 21

4.1.1 Tool-Schnittstelle . . . 22

4.1.2 Plug-In-Schnittstelle . . . 23

4.2 Kern des Erweiterungsmechanismus - Plug-In Manager . . . 25

4.3 Dateistruktur eines Fujaba Plug-Ins . . . 29

4.4 Ein Fujaba Plug-In-Generator - PluginCreator . . . 31

4.5 Anpassung und Erweiterung vorhandener Konzepte . . . 32

4.5.1 Erweiterung und Anpassungen an Klassenlader . . . 32

4.5.2 Anpassungen am Speichermechanismus und der Projektverwaltung . . . 32

4.6 Ein Hilfesystem für das Fujaba Plug-In . . . 33

5 Beispielsitzung 35
5.1 Plug-Ins Optionen und Sitzungsdaten 35
5.2 Informationen über geladene Plug-Ins 36
5.3 Update und Installation von Plug-Ins über das Internet 36
5.4 Speicherung der Plug-In Sitzungsdaten 38

6 Zusammenfassung und Ausblick 39
6.1 Zusammenfassung 39
6.2 Ausblick 40

A Fujaba Plug-in creator [English] 41
A.1 Plug-in description [Step 1] 41
A.1.1 1st Step 42
A.1.2 2nd Step 42
A.2 Plug-in requirement [Step 2] 43
A.3 Generating stable.xml document [Step 3] 44

Literatur 49

Index 51

Abbildungsverzeichnis

2.1 Fujaba Plattform-Architektur vor dem Erweiterungsmechanismus . . 5
2.2 Eclipse Plattform-Architektur 6
2.3 Horizontale und vertikale Versionierung von Softwaresystemen und Komponenten 8

3.1 Dynamisch-erweiterbare Fujaba Architektur 11
3.2 Dynamische Erweiterung von Fujaba mit Plug-Ins A und B 12
3.3 Anwendungsunabhängigkeit des Erweiterungsmechanismus 13
3.4 Fujaba und Plug-In Versionierung 14
3.5 Aus der Abbildung 3.4 erzeugte Abhängigkeitsgraph 15
3.6 Plug-Ins-Abhängigkeiten und Chain-of-responsibility 16
3.7 Beispielszenario zum Laden von Ressourcen mit dem Plug-In Klassenlader 17
3.8 Erweiterung der Fujaba GUI um Plug-In spezifische GUI-Elemente . 19

4.1 Schnittstellen des Erweiterungsmechanismus 21
4.2 KernelInterface Schnittstelle 22
4.3 Fujaba Plug-Ins Schnittstelle mit der dazugehörigen abstrakten Wrapper-Klasse 23
4.4 Ausschnitt aus dem Klassendiagramm des Erweiterungsmechanismus 25
4.5 Use-Case Diagramm über Plug-In Manager 26
4.6 Klassendiagramm Benutzerschnittstelle 27
4.7 Use-Case Diagramm über Plug-In Downloader 28
4.8 Interaktionsszenario von Fujaba und Plug-In Manager 28
4.9 Interaktionsszenario von Benutzer und Plug-In Downloader 29
4.10 Typische Dateistruktur von Fujaba Plug-Ins 29
4.11 Dateistruktur des Fujaba Referenz-Plug-Ins 30
4.12 Klassendiagramm vom Plugin.dtd 30

5.1 Dialog um Umgebungsoptionen für Plug-Ins zu setzen 35
5.2 Informationen über geladene Plug-Ins 36
5.3 Dialog um neue Fujaba Plug-Ins herunterzuladen bzw. einen Update vorzunehmen 37

5.4 Dialog über den Download- bzw. Installationsprozess 37
5.5 Dialog über plug-in-spezifische Sitzungsdaten 38

A.1 Dialog to generate plugin.xml document 41
A.2 Table of plug-in properties 42
A.3 Extended plug-in property dialog 43
A.4 Table of plug-in properties 44
A.5 Plug-in requirement dialog 45
A.6 Dialog to add required plug-ins 45
A.7 Table of required plug-ins attributes 46
A.8 Dialog to generating stable.xml document 46
A.9 Dialog to add actions 47
A.10 Table of action attributes 48

1 Einleitung

1.1 Motivation

Die Konzeption von Software als ein monolithisches System stößt heute an seine Grenzen. Solche Softwaresysteme werden sehr komplex und programmiertechnisch mit wachsender Größe immer schlechter beherrschbar. Diese Systeme sind zudem unflexibel, denn Sie lassen sich meistens nicht an die individuellen Bedürfnisse und Wünsche des Benutzers in der Hinsicht anpassen, dass bestimmte Funktionalitäten und Module des Gesamtsystems nicht ausgeschaltet/entfernt werden können.

Die *Fujaba Tool Suite* wurde ebenfalls als ein monolithisches System konzipiert und entwickelt, um eine Entwicklungsumgebung für Round-Trip Engineering mit UML, Java^{TM} und Design-Patterns zu realisieren. Desweiteren hatte die Entwicklung von Fujaba als ein monolithisches Softwaresystem als Ziel, ein breites Spektrum an Funktionalitäten, wie verschiedene UML-Diagramme, Reengineering, Realtime-Diagramme etc., anzubieten.

Die Konzeption von Fujaba als ein monolithisches Softwaresystem bringt neben dem nicht zu leugnenden Vorteil des Zusammenspiels der einzelnen Komponenten (Funktionalitäten) aber auch einiges an Nachteilen mit sich. Erhöhte Komplexität, Redundanz, extrem aufwändige Wartung und Pflege des Systems, lange Entwicklungszeiten und schlechte Wiederverwendbarkeit sind einige Nachteile, die besonders hervorzuheben sind.

Mit dem hier vorgestellten Konzept wird Fujaba zergliedert in einzelne Module - sogenannte Plug-Ins. Die Zergliederung von Fujaba in einzelne Plug-Ins erleichtert nicht nur die Projektierung, die Modellierung (Konstruktion) und Entwicklung, sondern auch die Handhabung, Pflege und Wartung (Änderung, Dokumentation). Die Modularisierung von Fujaba ermöglicht ein getrenntes Austesten der einzelnen Plug-Ins und günstige Arbeitsteilung zwischen Entwicklern.

Durch die Arbeitsteilung behindern sich die Entwickler nicht mehr gegenseitig. Fujaba wird dadurch transparenter und wartungsfreundlicher als bisher, da der Kern mit wenig Funktionalität sehr klein gehalten wird. Die Studierenden, die im Rahmen ihrer Studien- und/oder Diplomarbeit ein Werkzeug für Fujaba entwickeln,

können sich schneller und besser einarbeiten. Da die zu entwickelnden Werkzeuge ausschliesslich als Plug-Ins zur Vefügung gestellt werden, ist es ausgeschlossen, dass eine Abhängigkeit *Fujaba benötigt Werkzeug (Plug-In) XYZ* entsteht. Somit wird der Kern von Fujaba klein und stabil gehalten.

Ein Softwaresystem bestehend aus mehreren kleinen Modulen bietet Flexibilität in höchstem Maße. Der Anwender kann entscheiden, welche Module er benötigt und welche nicht. Somit werden nur die Module geladen, die er tatsächlich benötigt. Daher kann hier ein zusätzlicher Performanzgewinn erzielt werden. Zudem benutzt (und bezahlt) der Anwender nur die Funktionalitäten, die er benötigt. Ein Designer z.B. benötigt nicht alle Werkzeuge und Funktionalitäten, die einem Programmierer bereitgestellt werden und umgekehrt.

Modulare Softwaresysteme lassen sich wegen der Möglichkeit der erhöhten Wiederverwendung wesentlich schneller entwickeln als monolithische und führen damit zu verkürzten Entwicklungszeiten (Stichwort Time-To-Market). Ein in der Industrie und anderen Disziplinen selbstverständlicher Aspekt (Komponenten-/Modulbasierte Entwicklung) etabliert sich erst langsam in Informatik.

Ein weiterer Vorteil von modular aufgebauten Softwaresysteme ist die Testbarkeit. Tests können in einem Softwaresystem mit Softwarekomponenten, die in kleinen Modulen zerlegt sind, einfacher und effizienter durchgeführt werden als in einem monolithischen Softwaresystem. Daher können vorhandene Fehler im System schneller gefunden und behoben werden. Das spart Kosten und Entwicklungszeit.

Trotz bekannter ingenieurmäßiger Methoden wird Software häufig unstrukturiert konstruiert und implementiert. Die Folgen sind explodierende Wartungskosten, für die sowohl die notwendige Behebung von Fehlern als auch der große Aufwand, der im Erweiterungsprozeß entsteht, verantwortlich sind. Daher ist ein weiterer entscheidender Grund, warum immer mehr monolithische Softwaresysteme in modulare Systeme transformiert werden, die Wartbarkeit der Systeme.

Die Wartung eines monolithisch aufgebauten Systems ist um ein vielfaches aufwändiger als die Wartung eines modularen Softwaresystems, da die modularen Softwaresysteme häufig aus wiederverwendbaren Softwarebausteinen bestehen. Viele Software-Projekte scheitern, weil die Kosten für Wartung und Pflege explodieren. Man kann davon ausgehen, dass in der Softwarewartung und Pflege ein erhebliches Kostenreduzierungs- und Rationalisierungspotential steckt.

1.2 Zielsetzung

Diese Arbeit setzt sich die Ausarbeitung und Entwicklung eines *dynamischen Erweiterungsmechanismus* zum Ziel, welcher Fujaba modularisiert und die in Kapitel 1.1 aufgezeigten Probleme löst. Im Fujaba-Kern werden Funktionalitäten wie z.B. Speichermechanismus, Projektverwaltung, graphische Benutzungsschnittstelle realisiert. Weitere Funktionalitäten werden durch Plug-Ins bereitgestellt.

Diese Arbeit baut auf einen vorhandenen rudimentären Erweiterungsmechanismus auf. Dieser Erweiterungsmechanismus soll um Anwendungsunabhängigkeit erweitert werden, d.h. der im Rahmen dieser Arbeit entwickelte *Erweiterungsmechanismus* ist unabhängig von der Anwendung, in der er verwendet wird. Weiterhin wird er um ein Versions- und Konfigurationsmanagement ergänzt. Außerdem sollen Plug-Ins über das Internet installiert und aktualisiert werden.

1.3 Aufbau und Struktur der Arbeit

Die Arbeit ist unterteilt in folgende Kapitel:
In **Kapitel 2** wird die bisherige Architektur von Fujaba kurz vorgestellt. Anschließend werden Plug-In Konzepte und Versions- und Konfigurationsmanagement erläutert.

Die konzeptionelle Vorstellung der Fujaba Plug-Ins erfolgt in **Kapitel 3** gefolgt von der technischen Umsetzung in **Kapitel 4**. **Kapitel 5** beschäftigt sich mit einer Beispielsitzung des Benutzers mit Fujaba und Plug-Ins.

Den Abschluss bildet **Kapitel 6** mit einer Zusammenfassung und einem Ausblick auf mögliche anschliessende Arbeiten. Im Anhang befindet sich die Dokumentation und Benutzerhandbuch für ein Fujaba Plug-In, das einen Rahmen wie z.B. die Schnittstelle, die XML-Dokumente und Aktionen für Fujaba Plug-Ins generiert.

2 Grundlagen und Begriffe

2.1 Einführung in die Fujaba Tool Suite

Die *Fujaba Tool Suite* ist eine integrierte Entwicklungsumgebung (CASE Tool), welche an der Universität Paderborn seit 1997 entwickelt wird. Ursprünglich wollte man mit Fujaba eine visuelle Entwicklungsumgebung für UML zur Verfügung stellen. Mit der Zeit wurde Fujaba um zusätzliche Funktionalitäten, zum größten Teil von Studierenden im Rahmen von Studien- und Diplomarbeiten, erweitert. Einige wichtige Funktionalitäten und Erweiterungen, die hier erwähnt werden können, sind verschiedene UML-Diagramme wie Klassen- und Aktivitätsdiagramme, Statecharts, Realtime-Diagramme, sowie Re-Engineering.

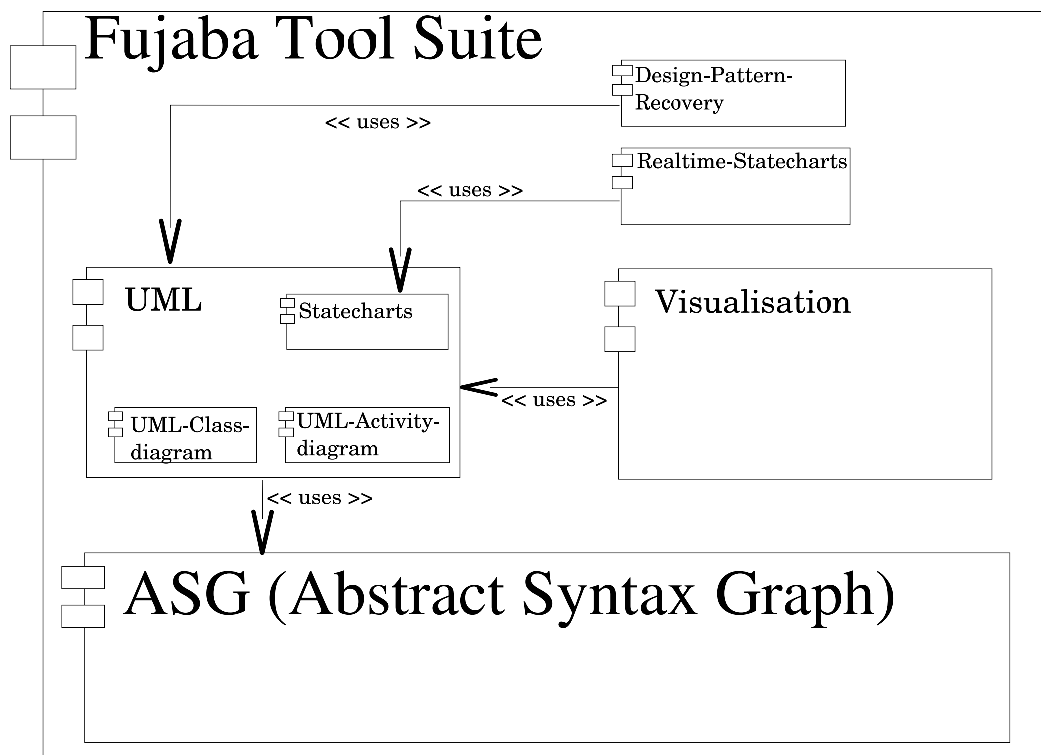

Abbildung 2.1: Fujaba Plattform-Architektur vor dem Erweiterungsmechanismus

In Abbildung 2.1 ist die *Fujaba Tool Suite* Plattform-Architektur abgebildet. Wie aus der Abbildung zu entnehmen ist, sind alle Komponenten ein fester Bestandteil der *Fujaba Tool Suite*. Um die Abbildung kompakt zu halten, wurden einige Komponenten nicht abgebildet. Die Abbildung stellt einen konzeptionellen IST-Zustand der *Fujaba Tool Suite* Architektur vor der Umstellung auf eine dynamisch-erweiterbare Software-Architektur dar.

2.2 Erweiterbare Softwarewerkzeuge und Softwaresysteme

Eclipse

Ein erweiterbares Softwaresystem bietet ein wohl-definiertes Application Programming Interface (API), um das Softwaresystem mit zusätzlichen Funktionalitäten zu erweitern. Ein sich in der Industrie mit diesem Konzept etabliertes Softwarewerkzeug ist *Eclipse*[1]. In Eclipse werden bis auf einen sehr klein gehaltenen Kern (*Platform Runtime*) alle Funktionalitäten in Plug-Ins verlagert [OTI04]. Die Abbildung 2.2

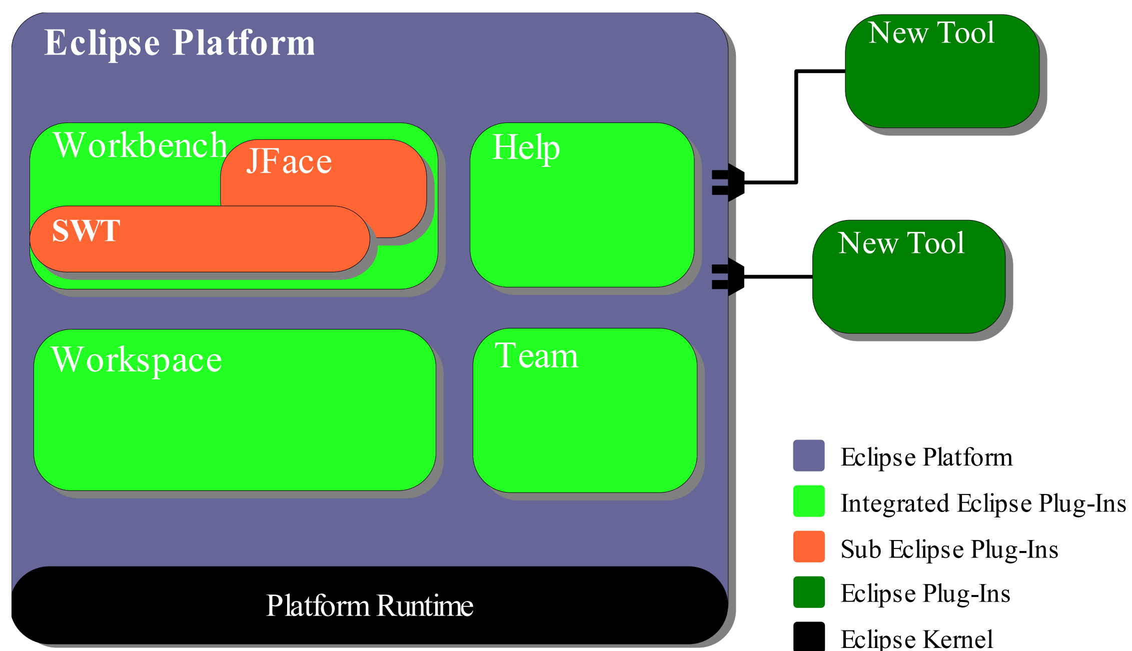

Abbildung 2.2: Eclipse Plattform-Architektur

zeigt die Plattform-Architektur von Eclipse, in der bis auf die *Platform Runtime*

[1] Eclipse ist eine dynamisch erweiterbare integrierte Entwicklungsumgebung

alle Funktionalitäten konsequent in Eclipse Plug-Ins ausgelagert sind. Der Erweiterungsmechanismus von Eclipse ist ein fester Bestandteil des Eclipse-Kerns. Der Eclipse-Kern kann zwar für eine eigene Anwendung verwendet werden, indem diese Anwendung auf dem Eclipse-Kern aufbaut. Eine bestehende Anwendung durch den Eclipse-Kern zu erweitern ist allerdings nicht ohne weiteres möglich.

Plug-In Architektur

Das Konzept der Plug-In Architektur und des dynamisch ladbaren Codes hat für die Entwicklung und Entwurf von Software, die mit dynamisch ladbaren Bibliotheken erweitert werden soll, an Bedeutung gewonnen. Es stellt für Entwickler eine attraktive Möglichkeit dar, eine Anwendung modular, anpassbar und einfach erweiterbar zu erstellen [ADC04]. In [LVM95] wird das Konzept der Plug-In Architektur sowie zwei weitere Konzepte wie *object connection architecture* und *interface connection architecture* für den Entwurf und die Implementierung von Software vorgestellt.

Was als eine einfache Möglichkeit begann, ein Softwaresystem mit Dritt-Anbieter Features zu erweitern, ohne den Zugriff auf Quell-Code zu geben, wurde für viele Entwickler ein methodischer Ansatz, um Software zu entwickeln. Schnelle Implementierung und Erweiterung mit neuen Features, einfache Isolierung von Modulen, um Fehler zu entdecken und zu lösen, benutzerdefinierte Versionen eines Softwaresystems ohne den Quell-Code zu ändern, Einbindung von Dritt-Anbieter Features, ohne Zugriff auf Quell-Code zu gewähren, sind nur einige erwähnenswerte Vorteile für die Entwickler.

Web-Applikationen auf der Basis einer Plug-In Architektur

Die Entwicklung von Software auf der Basis einer Plug-In Architektur kam erstmals in der Entwicklung von Web-Applikationen wie z.B. die bekannten Web-Browser Netscape Navigator, Mozilla sowie Microsoft Internet Explorer und Anwendungen wie Adobe Photoshop und Macromedia Director zum Einsatz [Eli00].

Sowohl Netscape Navigator und Mozilla als auch Microsoft Internet Explorer bieten die Möglichkeit, den Browser mit neuen Funktionalitäten zu erweitern, welche in Form von dynamisch ladbaren Bibliotheken zur Verfügung gestellt werden. Es zeigt sich, dass die Plug-In Architektur ein wichtiger Aspekt geworden ist, um Framework-ähnliche Umgebungen mit Komponenten dynamisch zu erweitern, die eine wohl-definierte Schnittstelle (Muster oder Protokoll) implementieren.

2.3 Versions- und Konfigurationsmanagement

Mit der ständig steigenden Größe und Komplexität von Softwaresystemen haben sich in den letzten Jahren auch die bei der Softwareentwicklung auftretenden

Probleme verschärft. Dazu zählen insbesondere die Erhaltung der Systemkonsistenz bei häufig stattfindenden Änderungen von Dokumenten und die Koordination der Teamarbeit [Sac99].

Bei der Entwicklung von komponenten- bzw. modulbasierter Software setzt man ein komponentenorientiertes Konfigurationsmanagement, das die Verwaltung versionierter Komponenten in den Vordergrund stellt, ein. In der Abbildung 2.3 ist

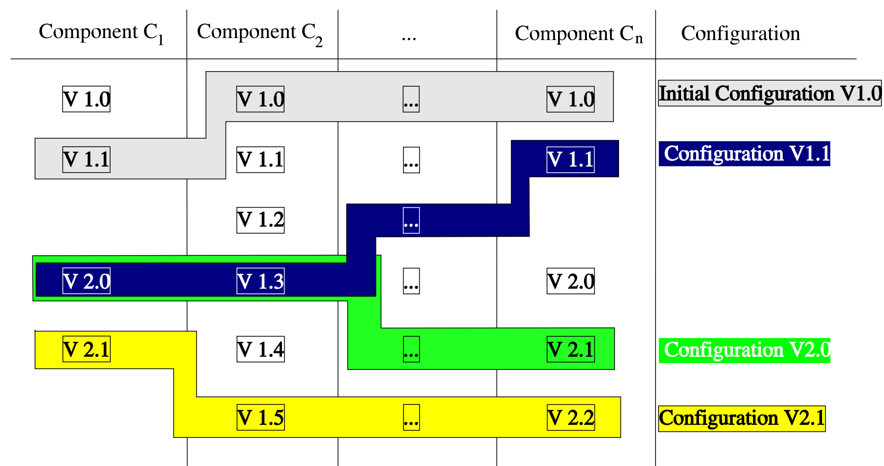

Abbildung 2.3: Horizontale und vertikale Versionierung von Softwaresystemen und Komponenten

ein typisches Versions- und Konfigurationsdiagramm eines komponentenbasierten Softwaresystems dargestellt. Die Versionierung der einzelnen Komponenten bzw. des Systems ist vertikal eingetragen und drückt die Entwicklungsgeschichte der Komponenten bzw. des Gesamtsystem aus. Die Versions-Nummer ist zweistellig mit Major und Minor Nummer, wobei die Major Nummer sich ändert, wenn die Schnittstellen der Komponenten geändert werden. Die Minor Nummer ändert sich, wenn die Komponente geändert wird, jedoch die Schnittstelle der Komponente von dieser Änderung nicht betroffen ist. Auf der horizontalen Ebene sind die Konfigurationen des Systems eingetragen.

Konfigurationen werden in einem zweistufigen Ansatz aus versionierten Komponenten zusammengesetzt. In der ersten Stufe werden die Komponenten eines Systems in Form eines Systemmodells festgelegt. So gibt das allgemeine Systemmodell für das in Abbildung 2.3 dargestellte Softwaresystem an, dass dieses System aus Komponenten $C_1, C_2, ..., C_n$ besteht. In der zweiten Stufe werden Versionsauswahlregeln definiert, mit Hilfe derer für jede im Systemmodell angegebene Komponente eine Version ausgewählt wird.

In Abbildung wurde die Version *v 1.1* der *Komponente* C_1, die Version *v 1.0* der *Komponente* C_2 usw. ausgewählt (in der Abbildung farblich hinterlegt). Diese ausgewählten Versionen bilden zusammen die initiale Systemkonfiguration bzw. *Systemversion 1.0*. Die Konfiguration des Gesamtsystems beginnt mit einer initialen Version und ändert sich, wenn die Versionierung einer Komponente im Gesamtsystem sich geändert hat.

3 Konzeption des dynamischen Erweiterungsmechanismus

3.1 Die neue Fujaba Tool Suite System-Architektur

Die *Fujaba Tool Suite* soll sich mit der dynamisch-erweiterbaren Software-Architektur sukzessive von der monolithisch-aufgebauten Software-Architektur ablösen, indem Komponenten, die bisher ein fester Bestandteil von Fujaba waren, in Plug-Ins verlagert werden. In der neuen Fujaba-Architektur bieten einige integrierte Komponenten eine Speicher- und Projektverwaltung sowie einen Abstrakten Syntaxgraphen (ASG) und UML-Diagramme, die über wohldefinierte Schnittstellen von Plug-In-Entwicklern benutzt werden können. Die Abbildung 3.1 zeigt, dass

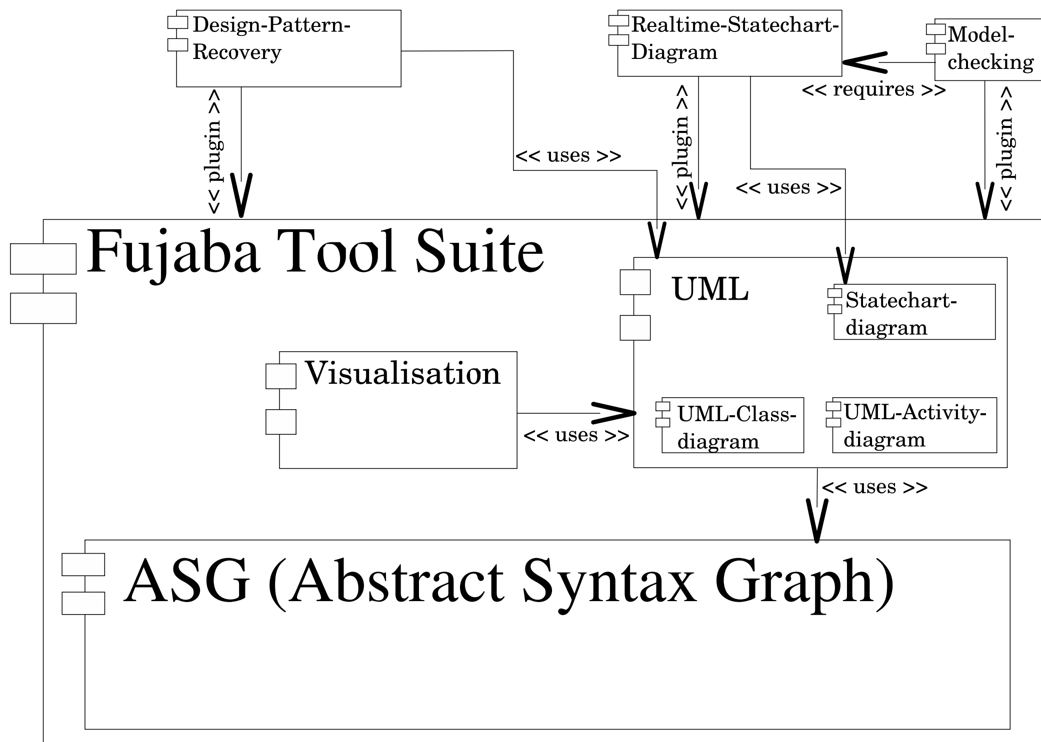

Abbildung 3.1: Dynamisch-erweiterbare Fujaba Architektur

neben den in Fujaba in Form von API und Komponenten bereitgestellten Funktionalitäten, Plug-Ins - auch von Dritt-Anbietern bereitgestellt - darauf aufbauende Dienste wie z.B. Realtime Statecharts anbieten aber auch Fujaba mit ganz neuen Funktionalitäten wie Topic Maps, Modelchecking usw. bereichern können.

Sowohl die *Fujaba Tool Suite* mit der neuen System-Architektur als auch Eclipse sind dynamisch erweiterbare Werkzeuge, die sich jedoch in technischen Details unterscheiden. In Fujaba ist der dynamische Erweiterungsmechanismus beispielsweise komplett unabhängig von Fujaba. Fujaba stellt lediglich eine Implementierung für die Plug-Ins Schnittstelle bereit.

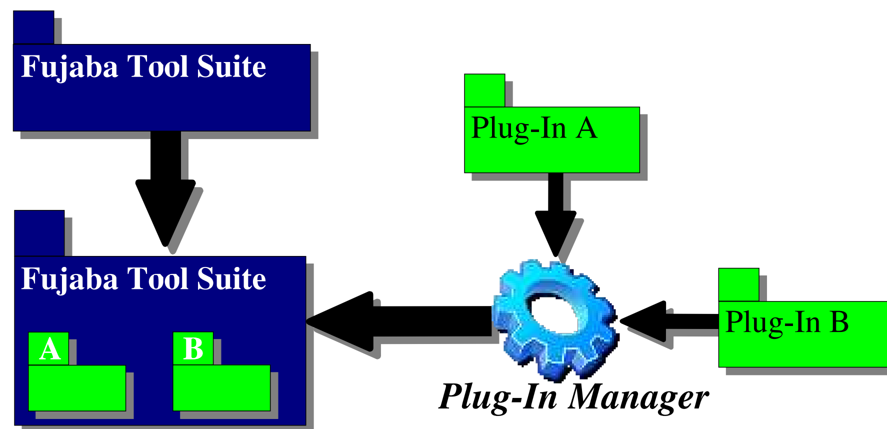

Abbildung 3.2: Dynamische Erweiterung von Fujaba mit Plug-Ins A und B

Ein weiterer entscheidender Unterschied sind die Laderoutinen für Plug-Ins. In Fujaba hat jedes Plug-In einen eigenen Klassenlader, so dass beispielsweise unterschiedliche Plug-Ins die gleiche Bibliothek in unterschiedlicher Versionen benutzen können.

Die Abbildung 3.2 zeigt, wie Fujaba mit dem Mechanismus der Plug-Ins dynamisch erweitert werden kann. Der *Plug-In Manager* parst und lädt die Plug-Ins A und B in Fujaba. Nach einem erfolgreichen Auflösen der Abhängigkeiten werden dem Anwender/Benutzer Funktionalitäten sowohl der Fujaba Entwicklungsumgebung als auch der Plug-Ins A und B bereitgestellt.

3.2 Anwendungsunabhängigkeit des dynamischen Erweiterungsmechanismus

Der dynamische Erweiterungsmechanismus ist unabhängig von Fujaba und kann daher in jeder Anwendung integriert werden, welche einen Erweiterungsmechanismus benötigt. Für die Integration des Erweiterungsmechanismus muss lediglich eine einfach gehaltene Schnittstelle implementiert werden.

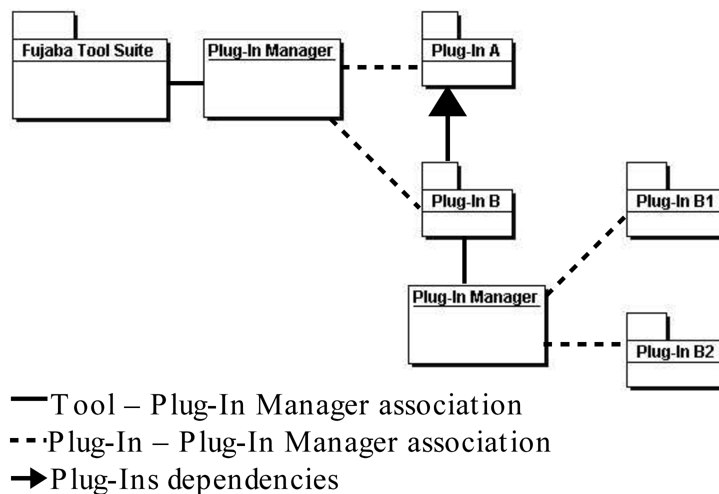

Abbildung 3.3: Anwendungsunabhängigkeit des Erweiterungsmechanismus

Wie in Abbildung 3.3 dargestellt, können auch Plug-Ins den dynamischen Erweiterungsmechanismus nutzen, um selbst Funktionalitäten in Plug-Ins zu verlagern. Somit kann eine Modularisierung bis auf kleinste Funktionseinheiten erreicht werden.

3.3 Versionierung der Anwendung und ihrer Plug-Ins

Der dynamische Erweiterungsmechanismus erlaubt Plug-Ins, vorhandene Plug-Ins zu nutzen und zu erweitern. Mit dieser Möglichkeit entstehen Plug-In-Abhängigkeiten, die aufgelöst werden müssen. Wenn ein Plug-In A ein Plug-In B voraussetzt, so muss das Plug-In B vorhanden sein (mit vorausgesetzter Version), damit Plug-In A geladen werden kann. Weiterhin kann ein Plug-In erst geladen werden, wenn alle benötigten (vorausgesetzten) Plug-Ins geladen worden sind.

Aus den Abhängigkeiten der Plug-Ins wird ein gerichteter Abhängigkeitsgraph erzeugt. Der Abhängigkeitsgraph kann unter Umständen auch Zyklen enthalten. Der Algorithmus für die Auflösung von Plug-In-Abhängigkeiten ist in der Lage, die Zyklen zu erkennen. Die erkannten Zyklen werden durchbrochen, um eine eindeutige Reihenfolge zu erzeugen, nach der die Plug-Ins geladen werden können. Dem Benutzer wird eine Warnung ausgegeben und er kann daraufhin entscheiden, ob er mit dem Ergebnis zu frieden ist oder nicht. Generell weist eine zyklische Abhängigkeit von Plug-Ins auf falsch beschriebene Abhängigkeiten im Plug-In-Beschreibungsdokument (plugin.xml) hin.

Zur Versionierung erhalten sowohl die Anwendung als auch ihre Plug-Ins Major, Minor und Build Nummern. Die Versionierung wurde eingeführt, um eine Schnittstellenkompatibilität der Plug-Ins untereinander (wegen Plug-In-Abhängigkeiten) aber auch die Schnittstellenkompatibilität zwischen der Anwendung und ihren Plug-Ins sicherzustellen.

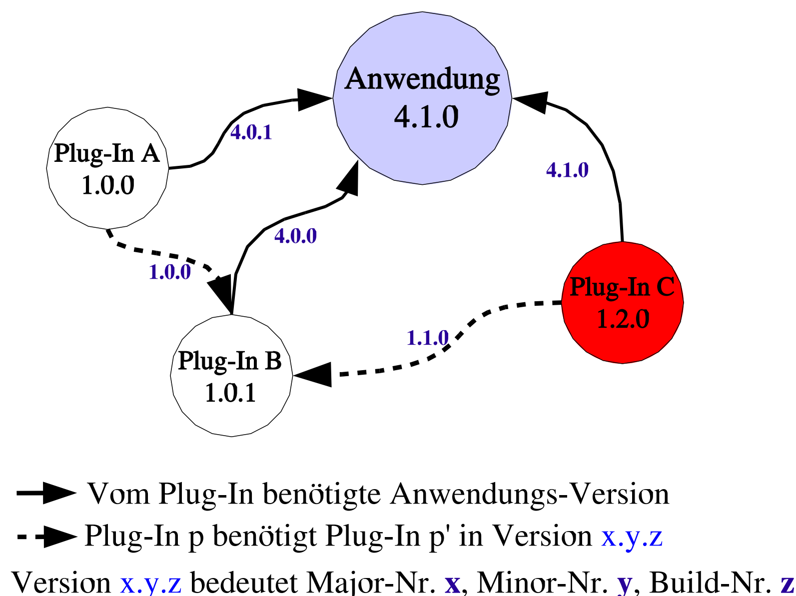

Abbildung 3.4: Fujaba und Plug-In Versionierung

Der Zusammenhang zwischen Versionsnummer und der Schnittstellenkompatibilität ist, dass durch die Änderung der Build Nummer die Schnittstelle nicht betroffen ist. Die Änderung beinhaltet vielmehr Bugfixes. Die Änderung der Minor Nummer weist auf eine Änderung, die auch die Schnittstelle betrifft. Die vorgenommene Änderung ist jedoch abwärts kompatibel, d.h. wenn ein Plug-In A ein Plug-In B in Version

1.0.0 benötigt, Plug-In B in Version 1.1.0 vorliegt, ist die Abhängigkeit erfüllt und beide Plug-Ins können geladen werden. Eine Änderung der Major Nummer weist auf eine Änderung hin, von der auch die Schnittstelle betroffen sein kann. Bei dieser Änderung kann nicht garantiert werden, dass die Schnittstellenkompatibilität erfüllt ist.

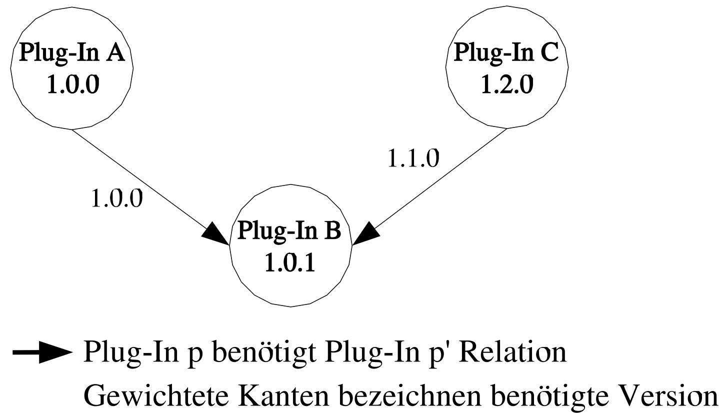

Abbildung 3.5: Aus der Abbildung 3.4 erzeugte Abhängigkeitsgraph

In Abbildung 3.4 wird deutlich, dass der Plug-In C nicht geladen werden kann, weil Plug-In C eine Version 1.2.0 vom Plug-In B benötigt, aber Plug-In B in Version 1.0.1 vorliegt. Plug-In A und B können geladen und instanziert werden, weil die Versionierungsanforderungen erfüllt sind.
Im Allgemeinen gilt:

Sei P_1 ein Plug-In in Version $x_1.y_1.z_1$ und sei $u_f^1.v_f^1.w_f^1$ die benötigte Anwendungs-Version. Die Anwendung sei in Version $x_f.y_f.z_f$ verfügbar. Plug-In P_1 sei von keinem Plug-In abhängig.
Sei P_2 ein weiteres Plug-In in Version $x_2.y_2.z_2$ und benötige Anwendungs-Version $u_f^2.v_f^2.w_f^2$. Plug-In P_2 sei vom P_1 abhängig und benötige Plug-In P_1 in Version $u_1^2.v_1^2.w_1^2$.
Plug-In P_1 kann genau dann geladen werden, wenn gilt:
$u_f^1 = x_f$ (benötigte Major-Nr. gleich vorhandene Major-Nr. der Anwendung)
$v_f^1 \geq y_f$ (benötigte Minor-Nr. größer gleich vorhandene Minor-Nr.)
$w_f^1 \geq z_f$ (benötigte Build-Nr. größer gleich vorhandene Build-Nr.)

Plug-In P_2 kann genau dann geladen werden, wenn gilt:
$u_f^2 = x_f$ (benötigte Major-Nr. gleich vorhandene Major-Nr. der Anwendung)
$v_f^2 \geq y_f$ (benötigte Minor-Nr. größer gleich vorhandene Minor-Nr.)
$w_f^2 \geq z_f$ (benötigte Build-Nr. größer gleich vorhandene Build-Nr.)

und es muss wegen der Abhängigkeit zum Plug-In P_1 gelten:
$u_1^2 = x_1$ (benötigte Major-Nr. gleich vorhandene Major-Nr. von Plug-In P_1)
$v_1^2 \geq y_1$ (benötigte Minor-Nr. größer gleich vorhandene Minor-Nr.)
$w_1^2 \geq z_1$ (benötigte Build-Nr. größer gleich vorhandene Build-Nr.)

Die Schnittstellenkompatibilität bezüglich Versionierung wird mit dem oben vorgestellten Verfahren erreicht. Dadurch kann sichergestellt werden, dass die Plug-Ins untereinander und mit der Anwendung konform bezüglich der Schnittstellendefinition sind.

3.4 Lade-Mechanismus für Plug-Ins

Beim Laden der Anwendung wird im Unter-Verzeichnis ***plugins*** des Installationsverzeichnisses und in benutzerdefinierten Verzeichnissen nach möglichen Plug-Ins gesucht. Wenn der Plug-In Manager ein Plug-In gefunden hat, wird dieses geladen und ein Klassenlader für diesen instanziiert. Das Plug-In kann dann alle Ressourcen, die es benötigt, über diesen Klassenlader laden bzw. über den Mechanismus der Chain-of-Responsibility den Ladevorgang für Ressourcen an andere Klassenlader delegieren.

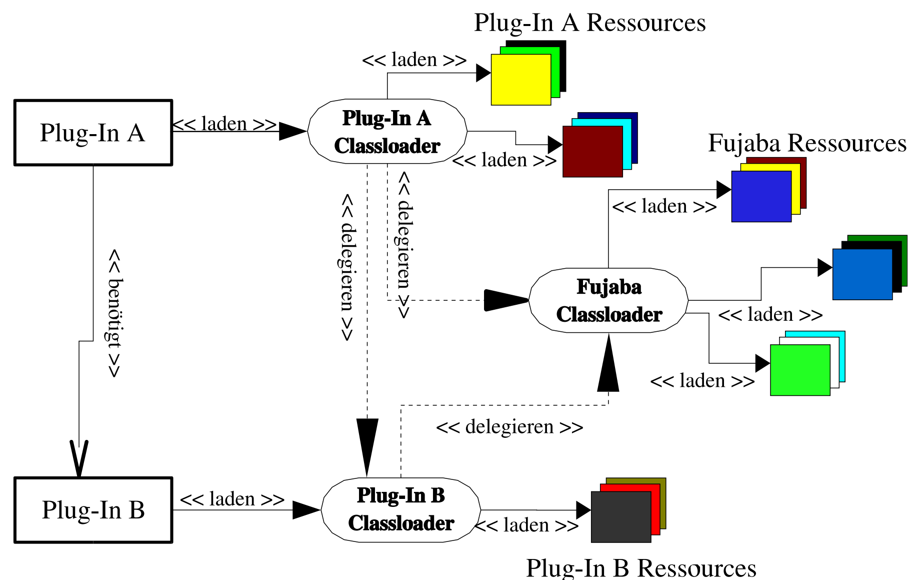

Abbildung 3.6: Plug-Ins-Abhängigkeiten und Chain-of-responsibility

Plug-In-Abhängigkeiten und Chain-of-responsibility
Für jedes Plug-In existiert eine Klassenlader-Instanz, die für das Laden von Ressourcen zuständig ist. Es können so Bibliotheken in verschiedenen Versionen durch Plug-Ins geladen und benutzt werden, d.h. ein Plug-In kann mit einer aktuellen Version einer Bibliothek (z.B. XML4Java-2-1-16.jar) arbeiten, während ein anderes Plug-In eine andere Version benutzt (z.B. XML4Java-1-1-0.jar).

Aus der Abbhängigkeitskette folgt, wie in Abbildung 3.6 dargestellt, eine Zuständigkeitskette für das Laden von Ressourcen. Wenn ein Plug-In eine Ressource laden will, wird zunächst versucht, diese über den eigenen Klassenlader zu laden. Falls die Ressource nicht geladen werden konnte, so werden gemäß der Abhängigkeitskette weitere Klassenlader durchlaufen. Falls auch dieser Ladevorgang fehlschlägt, wird das Laden der Ressource an den Klassenlader der Anwendung delegiert.

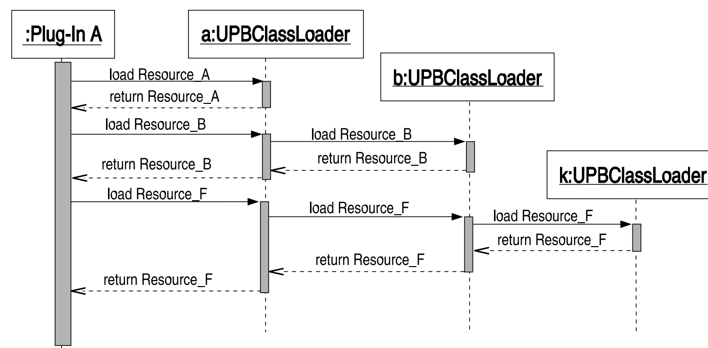

Abbildung 3.7: Beispielszenario zum Laden von Ressourcen mit dem Plug-In Klassenlader

Abbildung 3.7 stellt eine Beispielszenario über das Laden von Ressourcen mit dem Plug-In Klassenlader in Form eines Sequenzdiagramms dar. Wie in Abbildung 3.6 sei Plug-In A abhängig vom Plug-In B. In Abbildung 3.7 ist der Klassenlader a für das Laden von Plug-In A Ressourcen, Klassenlader b für Plug-In B Ressourcen und Klassenlader k für das Laden von Ressourcen der Kern-Anwendung zuständig.

Beim Laden von Ressourcen des Plug-Ins A (Ressource_A) liefert der Klassenlader A die Ressource. Beim Laden von Ressource Ressource_B wird der Ladevorgang an Klassenlader B für Plug-In B delegiert, weil zuvor der Ladevorgang von Klassenlader a des Plug-Ins A fehlschlägt. Der Ladevorgang einer Ressource wird nur dann weiterdelegiert, wenn er fehlsclägt. Der Klassenlader B lädt und liefert die Ressource zurück. Beim Laden von Ressourcen der Kern-Anwendung wird zunächst die Zuständigkeitskette der Klassenlader durchlaufen und zum Schluss der Klassenlader

von der Kern-Anwendung mit dem Ladevorgang beauftragt. Die Zuständigkeitskette wird nur dann durchlaufen und der Ladevorgang weiterdelegiert, wenn der Ladevorgang zuvor fehlschlägt. Falls der Ladevorgang durch den Klassenlader der Kern-Anwendung fehlschlägt, wird der Vorgang abgebrochen und eine Fehlermeldung ausgegeben, dass eine Ressource nicht gefunden und geladen werden kann.

3.5 Update und Installation von Plug-Ins

Im Rahmen dieser Arbeit wurde der dynamische Erweiterungsmechanismus um eine Funktionalität erweitert, die dem Benutzer die Verwendung und Installation von Plug-Ins vereinfacht. Es können anwendungsspezifische Plug-Ins von einem Server heruntergeladen und installiert werden. Um Plug-Ins über das Internet herunterzuladen und zu installieren, müssen diese vorher auf einem Web-Server bereitgestellt werden.

Über die Schnittstelle zum Erweiterungsmechanismus wird eine Liste von Web-Servern übergeben, die Plug-Ins bereitstellen. Auf den Web-Servern befinden sich XML-basierte Beschreibungen aller Plug-Ins. Die verfügbaren Plug-Ins können dem Benutzer angezeigt werden. Der Benutzer kann Plug-Ins nach seinem Wunsch herunterladen und installieren bzw. ein Update vornehmen. Dabei werden auch gemäß der Abhängigkeitsbeschreibung im XML-Dokument weitere benötigten Plug-Ins heruntergeladen, falls diese lokal nicht verfügbar sein sollten.

3.6 Fujaba GUI-Erweiterung

Die graphische Benutzungsschnittstelle (GUI) von Fujaba basiert auf mehreren XML-Dokumenten, welche beim Starten von Fujaba geladen und geparst werden. Bevor der Fujaba UI-Manager aus den Dokumenten die GUI-Elemente parst und extrahiert, fügt der Plug-In Manager die GUI-Definition von Plug-Ins - ebenfalls definiert in einem XML-Dokument dem UI-Manager hinzu. Das heißt, die Definition der Fujaba-Kern GUI und die Plug-In GUI-Elemente aus den XML-Dokumenten für jedes Plug-In werden zu einem einzigen großen XML-Dokument gemischt und durch den UI-Manager geparst. Die nötigen GUI-Elemente werden generiert und die dazugehörigen Aktionen entsprechend ihrer Definition geladen und instanziert.

Abbildung 3.8 zeigt die konzeptionelle Funktionsweise der Erweiterung der Fujaba GUI mit GUI-Einträgen, die in Plug-Ins definiert werden und dem Benutzer zur Vefügung gestellt werden. Grundsätzlich werden die Plug-Ins erst durch Ausführung einer im Plug-In definierten Aktion, welche beispielsweise einer

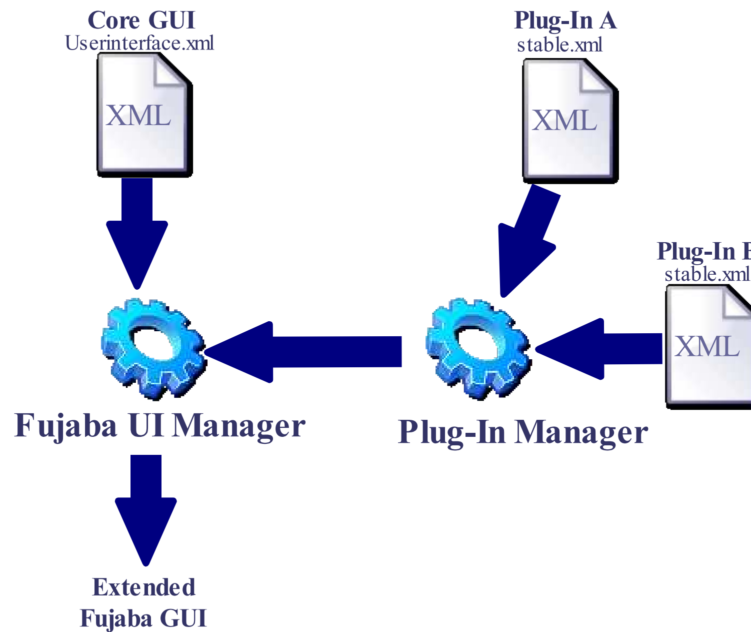

Abbildung 3.8: Erweiterung der Fujaba GUI um Plug-In spezifische GUI-Elemente

Schaltfläche zugeordnet ist, aktiv. Eine Ausnahme besteht beim erstmaligen Laden der Plug-Ins, indem für jedes Plug-In die initialize-Methode aufgerufen wird, welche die Plug-Ins in einen initialisierten Zustand versetzt.

Der UI-Manager von Fujaba wurde nicht im Rahmen dieser Arbeit entwickelt. Dieser wurde lediglich weiterentwickelt, um beispielsweise dafür zu sorgen, dass die definierten GUI-Elemente vom jeweils zuständigen Plug-In Klassenlader instanziert werden. In diesem Zusammenhang wurde ebenfalls der Klassenlader an den Erweiterungsmechanismus angepasst und erweitert.

4 Technische Umsetzung

In diesem Kapitel wird die technische Umsetzung der Konzepte und Ideen, die in den vorherigen Kapiteln konzeptionell vorgestellt wurden, anhand von Klassendiagrammen, Designdokumenten, etc. dokumentiert und die Realisierung vorgestellt.

4.1 Schnittstellen des Erweiterungsmechanismus

Der Erweiterungsmechanismus besteht aus zwei sehr einfach gehaltenen Schnittstellen **KernelInterface** und **PluginInterface**. Das **KernelInterface** dient zur Interaktion mit einer Anwendung - hier für die Interaktion mit der *Fujaba Tool Suite* - und das PluginInterface zur Interaktion mit Plug-Ins. Für die PluginInterface Schnittstelle existiert eine Wrapper-Klasse **AbstractPlugin**, die diese Schnittstelle implementiert. Um eine volle Konformität mit der Plug-In Schnittstelle zu erreichen und Konformität über zukünftige Schnittstellenerweiterungen hinaus zu erreichen, wird empfohlen, dass die Plug-Ins eine Klasse anbieten, die **AbstractPlugin** erweitert.

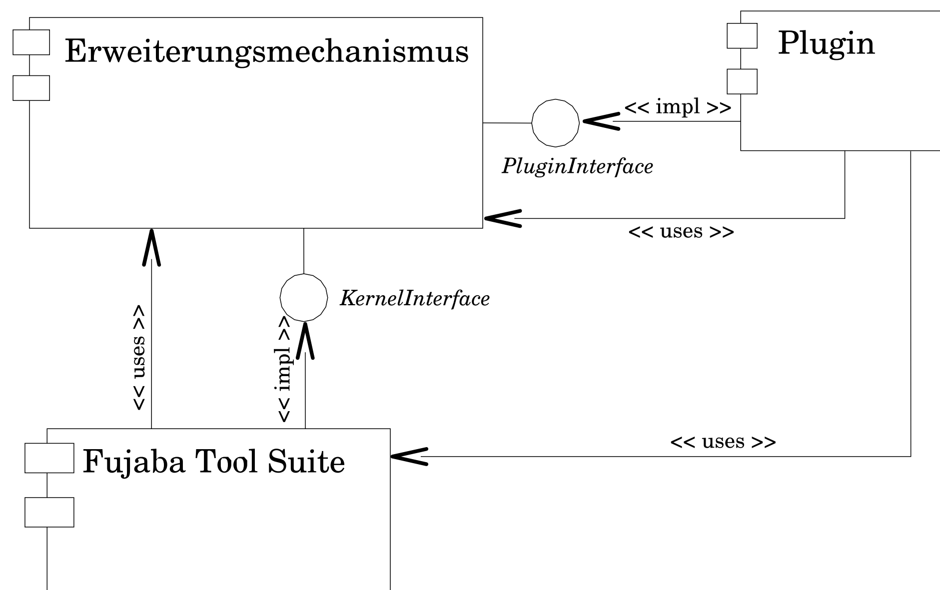

Abbildung 4.1: Schnittstellen des Erweiterungsmechanismus

Abbildung 4.1 zeigt die Schnittstellen des Erweiterungsmechanismus mit anderen Komponenten wie der *Fujaba Tool Suite* und Fujaba Plug-Ins. Aus der Abbildung kann man auch entnehmen, dass der Erweiterungsmechanismus keine Abhängigkeitsrelationen zu anderen Komponenten hat.

4.1.1 Tool-Schnittstelle

Im Kapitel 3.2 wurde konzeptionell die Tool-Unabhängigkeit des dynamischen Erweiterungsmechanismus vorgestellt. Der Erweiterungsmechanismus bietet eine sehr einfach gehaltene Schnittstelle, die lediglich implementiert werden muss, um den Mechanismus zu nutzen.

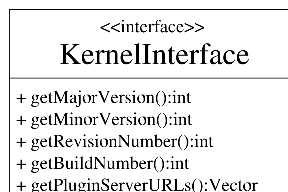

Abbildung 4.2: KernelInterface Schnittstelle

Die Abbildung 4.2 zeigt die **KernelInterface** Schnittstelle, die von der Anwendung, hier *Fujaba Tool Suite*, implementiert werden muss. Die Schnittstelle enthält Versionsinformationen über die Anwendung und eine Liste von URLs, von denen Plug-Ins heruntergeladen und installiert werden können.
Erläuterungen zu den Methoden der Schnittstelle:
+getMajorVersion():int Diese Methode liefert die Major Versionsnummer der Anwendung.
+getMinorVersion():int Diese Methode liefert die Minor Versionsnummer der Anwendung.
Für die Versionierung und Auflösung von Abbhängigkeiten werden lediglich die Major und Minor Nummer berücksichtigt.
+getRevisionNumber():int Die Methode liefert die Revision Versionsnummer der Anwendung.
+getBuildNumber():int Die Methode liefert die Build Versionsnummer der Anwendung.
+getPluginServerURLs():Vector Diese Methode liefert ein Objekt vom Typ **Vector**, welches die URLs der verfügbaren Plug-In-Server enthält, von denen Plug-Ins heruntergeladen werden können. Falls die Anwendung keine Möglichkeit des Plug-In-Downloads und Updates anbieten soll, so kann die Methode auch eine leere Liste (leeren Vector) zurückgeben.

4.1.2 Plug-In-Schnittstelle

Damit Plug-Ins in einer Anwendung wie der *Fujaba Tool Suite* geladen werden können, muss eine Schnittstelle implementiert werden bzw. eine abstrakte Klasse erweitert werden. Neben der Implementierung der Schnittstelle muss jedes Plug-In zwei XML-Dokumente bereitstellen, aus denen die Plug-In Informationen geparst werden. Was genau ein Plug-In benötigt, um als ein Plug-In erkannt und geladen werden zu können, behandelt das Kapitel 4.3.

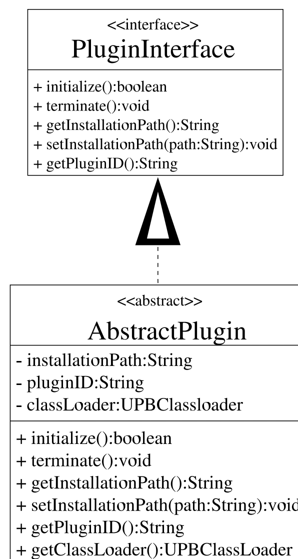

Abbildung 4.3: Fujaba Plug-Ins Schnittstelle mit der dazugehörigen abstrakten Wrapper-Klasse

Die Abbildung 4.3 stellt die Schnittstelle zu Plug-Ins und eine abstrakte Implementierung dieser Schnittstelle als eine Wrapper-Klasse dar. Es wird, statt einer Implementierung der Schnittstelle, die Erweiterung der abstrakten Wrapper-Klasse empfohlen, da dadurch jedes Plug-In immer konform bezüglich der künftigen Schnittstellenerweiterung bleibt.
Eine Implementierung bzw. Überschreibung folgender Methoden ist notwendig:
+initialize():boolean
Diese Methode soll das Plug-In initialisieren. Sie wird ausgeführt, bevor ein Plug-In durch eine Benutzerinteraktion aktiviert wird. Falls die Initialisierung erfolgreich ausgeführt wird, liefert die Methode **true**, sonst **false** zurück. Bei einer fehlgeschlagenen Initialisierung, d.h. wenn die Methode **false** liefert, wird das Plug-In als *dirty* markiert und der Ladevorgang dieses Plug-Ins abgebrochen. Alle Plug-Ins, die von dem betroffenen Plug-In abhängen, werden ebenfalls als *dirty* markiert und nicht geladen.

+terminate():void
Diese Methode terminiert das Plug-In, damit das Plug-In auf eine Beendigung der Anwendung reagieren kann. Bevor die Anwendung beendet wird, werden die **terminate** Methoden aller Plug-Ins aufgerufen, um die Plug-Ins ordnungsgemäß zu beenden und eine konsistente Datenhaltung zu gewährleisten.

+getInstallationPath():String
Diese Methode liefert den absoluten Installationspfad eines Plug-Ins. Diese Methode muss nicht überschrieben werden, da diese Methode lediglich eine Referenz auf das Objektattribut *-installationPath:String* liefert.

+setInstallationPath(path:String):void
Diese Methode muss ebenfalls nicht überschrieben werden, da auch diese Methode auf das Objektattribut *-installationPath:String* zugreift und dessen Wert ändert. Diese Methode kann aufgerufen werden, wenn das Plug-In seinen Installationspfad ändern will.

+getPluginID():String
Diese Methode liefert die ID des Plug-Ins, welche im XML-Dokument *plugin.xml* für die Plug-In-Beschreibung angegeben ist. Die ID eines Plug-Ins ist stets der voll-qualifizierte Klassenname einer Klasse, die die PluginInterface Schnittstelle implementiert oder die abstrakte Klasse **AbstractPlugin** erweitert. Die ID eines Plug-Ins wird auch für die Erzeugung eines für das Plug-In zuständigen Klassenlader benutzt. Diese Methode gibt die Referenz auf das Attribut *-pluginID:String* zurück.

+setPluginID(id:String):void
Sowohl die setPluginID als auch die getPluginID Methoden müssen nicht überschrieben zu werden, da diese wie die getInstallationPath und setInstallationPath lediglich Zugriffsmethoden auf Objektattribute sind. Die setPluginID wird aufgerufen, wenn das Plug-In durch den Plug-In Manager instanziert wird.

+getClassLoader():UPBClassLoader
Diese Methode liefert den für das Plug-In zuständigen Klassenlader zurück. Die Methode hält eine Referenz auf das Attribut *-classLoader:UPBClassLoader*, welches mit dem *pluginID* Attribut einen Klassenlader instanziert. Über diese Referenz werden alle für das Plug-In nötigen Ressourcen geladen.

Ein Plug-In, das die Klasse *AbstractPlugin* erweitert, muss lediglich die *initialize* und *terminate* Methoden in der abgeleiteten Klasse überschreiben. Von einer Überschreibung dieser Methoden kann auch abgesehen werden, wenn das Plug-In in der Initialisierungs- und Terminierungsphase nicht reagieren soll.

4.2 Kern des Erweiterungsmechanismus - Plug-In Manager

Den Kern des Erweiterungsmechanismus bildet die Klasse **PluginManager**. Diese Klasse beinhaltet Routinen, um nach Plug-Ins zu suchen, sie zu parsen und zu laden. Der Plug-In Manager löst Plug-In-Abhängigkeiten auf und erstellt eine Reihenfolge, in der die Plug-Ins geladen werden können. Abhängigkeitszyklen werden erkannt und durchbrochen und durch eine Warnmeldung ausgegeben.

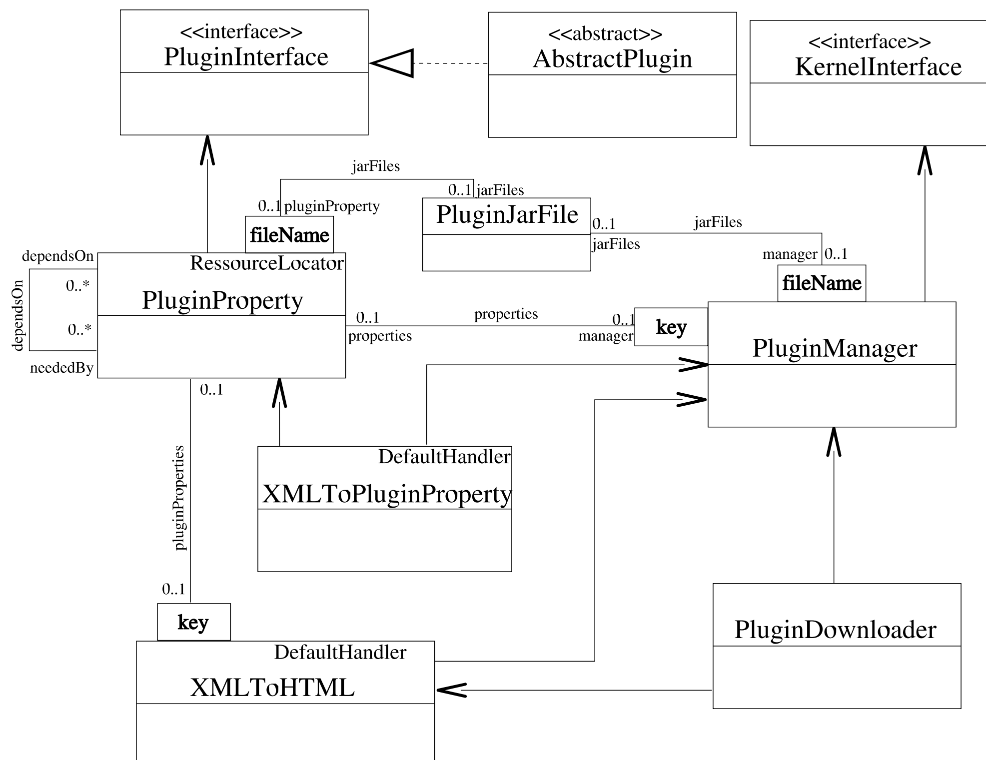

Abbildung 4.4: Ausschnitt aus dem Klassendiagramm des Erweiterungsmechanismus

Um eine Instanz vom **PluginManager** zu erzeugen, muss eine gültige Referenz auf eine **KernelInterface** Schnittstelle dem Konstruktor übergeben werden. Der Default-Konstruktor der **PluginManager** Klasse ist als privat deklariert. Die *Fujaba Tool Suite* bietet eine Implementierung der **KernelInterface** Schnittstelle an und erhält durch einen Aufruf des Konstruktors der **PluginManager** Klasse eine Referenz auf eine **PluginManager** Instanz.

Eine Instanz vom Typ **XMLToPluginProperty** parst das Plug-In-

Beschreibungsdokument (*plugin.xml*) in ein **PluginProperty** Objekt. Eine **PluginProperty** Instanz repräsentiert ein Plug-In im Erweiterungsmechanismus. Der **PluginManager** hat, wie in der Abbildung 4.4 zu erkennen, eine voll-qualifizierte Assoziation zu **PluginProperty**, um alle geladenen Plug-Ins referenzieren zu können. Die Plug-In-Abhängigkeiten sind im Modell durch Selbstassoziation der **PluginProperty** Klasse ausgedrückt. Die Klasse **PluginJarFile** repräsentiert alle Bibliotheken und Ressourcen des Plug-Ins, die zu einer Jar-Datei gepackt sind. Die Bibliotheken und die Plug-In-Jar-Datei, die die Schnittstellenimplementierung bzw. eine Unterklasse der **AbstractPlugin** Klasse enthält, werden im *plugin.xml* Dokument deklariert.

Das GUI-Beschreibungsdokument (*stable.xml*) wird vom GUI-Manager geparst und die benötigten GUI-Elemente wie Schaltflächen, Menüeinträge etc. mit den dazugehörigen Aktionsklassen instanziert. Der GUI-Manager ist nicht im Rahmen dieser Arbeit entwickelt worden, sondern wurde lediglich angepasst. Damit der Erweiterungsmechanismus unabhängig von Fujaba bleibt, ist der GUI-Manager in einem unabhängigen Modul implementiert. Die GUI-Elemente bilden die Schnittstelle der Interaktion des Benutzers mit den Plug-Ins. Abbildung 4.5 stellt ein Use-Case

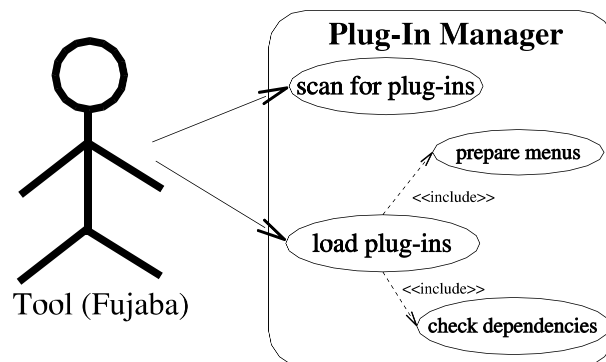

Abbildung 4.5: Use-Case Diagramm über Plug-In Manager

Diagramm dar, welches die angebotenen Funktionalitäten des **PluginManagers** illustriert. Eine Anwendung - hier die *Fujaba Tool Suite* benutzt die Funktionalität *scan for plug-ins*, um in vor- und benutzerdefinierten Verzeichnissen nach Plug-Ins zu suchen. Die gefundenen Plug-Ins werden in einem Abhängigkeitsgraphen des Erweiterungsmechanismus repräsentiert. Durch die Benutzung der Funktionalität *load plug-ins* werden alle gefundenen Plug-Ins geladen, instanziert und initialisiert. Die Laderoutine enthält weitere Routinen, die nur dem **PluginManager** zur Verfügung stehen, um z.B. die Plug-In Abhängigkeiten aufzulösen oder die Plug-In Menüs vorzubereiten und die Ladung zu initiieren. Die *check dependencies* Funktionalität erzeugt eine Reihenfolge, in der die Plug-Ins geladen werden können, dabei wandern

Plug-Ins, die von keinem anderen Plug-In abhängen an den Anfang der Liste. Falls keine eindeutige Reihenfolge erstellt werden kann, wenn z.B. zyklische Abhängigkeiten zwischen Plug-Ins vorhanden sind, wird die zyklische Abhängigkeitskette durchbrochen und dem Anwender eine Warnmeldung ausgegeben. Abbildung 4.6 zeigt ein

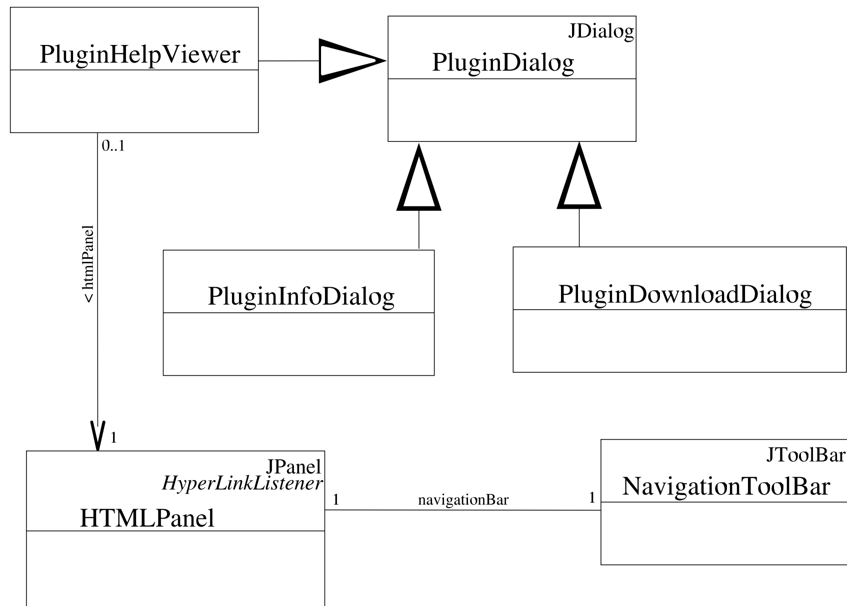

Abbildung 4.6: Klassendiagramm Benutzerschnittstelle

Klassendiagramm über die Benutzerschnittstelle des Erweiterungsmechanismus. In der Abbildung befinden sich alle Klassen, die für eine graphische Interaktion mit dem Benutzer benötigt werden. Die **PluginInfoDialog** Klasse ist für die Anzeige von Plug-In Informationen implementiert. Die **PluginDownloadDialog** Klasse stellt dem Benutzer einen Dialog zur Verfügung, mit dem er Plug-Ins herunterladen und installieren kann, bzw. ein Update für vorhandene Plug-Ins vornehmen kann. Die Klassen **PluginHelpViewer**, **HTMLPanel** und **NavigationToolBar** bilden das Hilfesystem für Plug-Ins. Über das Hilfesystem der Plug-Ins kann der Benutzer Unterstützung bei der Erledigung seiner Arbeit mit den Plug-Ins bekommen. Wie in der Abbildung 4.7 zu erkennen ist, bietet der Plug-In Downloader dem Benutzer zwei Funktionalitäten, *download plug-ins* und *update plug-ins*, an. Aus der Sicht des Download-Mechanismus gibt es keinen Unterschied zwischen den *download plug-ins* und *update plug-ins* Funktionalitäten. In beiden Fällen wird das Plug-In heruntergeladen und in Fujabas Plug-In Verzeichnis installiert. Bei der *update plug-ins* Operation wird lediglich ein vorhandenes Plug-In durch eine neue Version dieses Plug-Ins überschrieben. In Abbildung 4.8 ist ein Interaktionsszenario der *Fujaba Tool Suite* mit der **PluginManager** Klasse zu sehen. Beim Starten von Fujaba

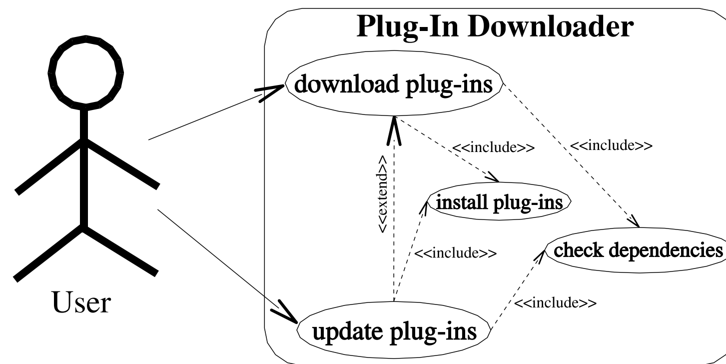

Abbildung 4.7: Use-Case Diagramm über Plug-In Downloader

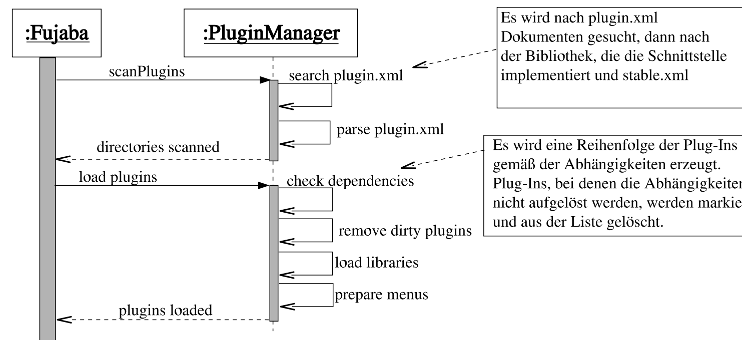

Abbildung 4.8: Interaktionsszenario von Fujaba und Plug-In Manager

wird der **PluginManager** initiiert, um in vor- und benutzerdefinierten Verzeichnissen nach Plug-Ins zu suchen. Nachdem die Verzeichnisse nach Plug-Ins durchsucht wurden, werden die gefundenen Plug-Ins im Modell des Erweiterungsmechanismus repräsentiert. Mit dem Aufruf *load plug-ins* werden alle gefundenen Plug-Ins geladen, initialisiert und die GUI-Elemente für das Laden durch den UI-Manager vorbereitet. Eine Benutzerinteraktion mit dem Download-Mechanismus stellt Abbildung 4.9 dar. Nach dem Aufruf von *showDownloadDialog* durch den Benutzer werden die URLs für Plug-In Download ermittelt. Die XML-Dokumente der Plug-Ins werden heruntergeladen und eine Liste von verfügbaren Plug-Ins dem Benutzer angezeigt. Der Benutzer wählt das Plug-In, das er installieren will. Daraufhin wird ein Statusdialog über den Download bzw. Installationsprozess angezeigt.

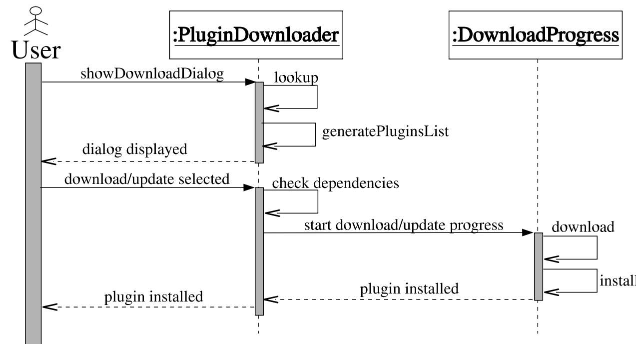

Abbildung 4.9: Interaktionsszenario von Benutzer und Plug-In Downloader

4.3 Dateistruktur eines Fujaba Plug-Ins

Damit der Erweiterungsmechanismus ein Fujaba Plug-In erkennen und laden kann, muss das Plug-In eine vorgeschriebene Struktur aufweisen. Es müssen ein Dokument

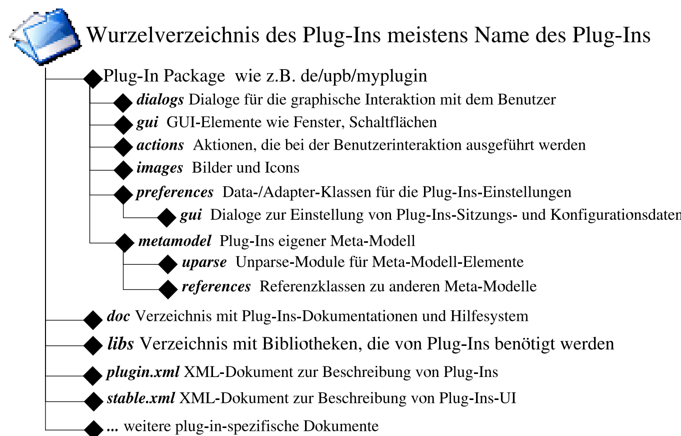

Abbildung 4.10: Typische Dateistruktur von Fujaba Plug-Ins

(*plugin.xml*) für die Beschreibung des Plug-Ins, ein weiteres Dokument für die Definition und Beschreibung von GUI-Elementen (*stable.xml*), sowie ein Java-Archiv (eine JAR-Datei) vorhanden sein, welches die Klasse enthält, die die Plug-In Schnittstelle **PluginInterface** implementiert oder die **AbstractPlugin** Klasse erweitert. Für das Laden und Instanzieren von Fujaba Plug-Ins werden diese Dokumente vorausgesetzt.

Die Plug-Ins können weitere plug-in-spezifische und optionale Dokumente wie ein *libs* Verzeichnis für benötigte Bibliotheken und ein *doc* Verzeichnis für Plug-In-

Dokumentation und Hilfesystem (siehe Abbildung 4.10) enthalten. Die Abbildung 4.10 zeigt eine empfohlene Datei-Struktur eines Fujaba Plug-Ins. Falls das Plug-In keine Dialoge für die Interaktion mit dem Benutzer anbietet, muss man kein Verzeichnis für die Dialoge im Plug-In Datei-Struktur anlegen. In der Abbildung werden alle Dateien und Verzeichnisse als ein JAR-Archiv gepackt (z.B. *MyPlugin.jar*) und weitergegeben. Demnach sieht das Plug-In in Fujabas **plugins** Verzeichnis wie folgt aus, wenn z.B. das Plug-In *MyPlugin* benannt ist.

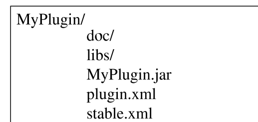

Abbildung 4.11: Dateistruktur des Fujaba Referenz-Plug-Ins

Plug-In Beschreibungsdokument - plugin.xml

Das Plug-In Beschreibungsdokument (*plugin.xml*) ist eine Instanz vom Typ (Document type definition) *Plugin.dtd*. Das *plugin.xml* enthält den Namen, die Version, die benötigte Fujaba Version, benötigte Plug-Ins und eine Beschreibung über das Plug-In. Abbildung 4.12 zeigt die in ein Klassendiagramm transformierte DTD **Plu-**

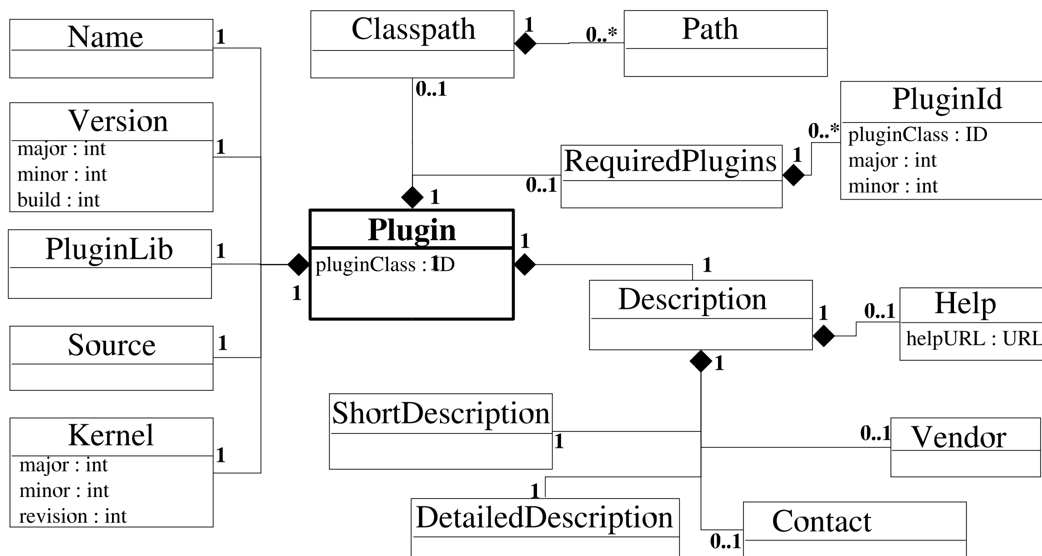

Abbildung 4.12: Klassendiagramm vom Plugin.dtd

gin.dtd. Wie in der Abbildung zu sehen ist, hat das Plugin-Element *Name*, *Version*, *PluginLib*, *Source*, *Kernel* und *Description* als Kind-Elemente und zwei weitere optionale Kind-Elemente *ClassPath* und *RequiredPlugins*.

Plug-In GUI-Elemente - stable.xml

Das Beschreibungsdokument der Plug-In GUI-Elemente (*stable.xml*) ist eine Instanz vom Typ *Userinterface.dtd*. Dieses Dokument enthält alle Aktionen, Schaltflächen und Menüeinträge, die das Plug-In für die Interaktion mit dem Benutzer zur Verfügung stellt.

Die *Fujaba Tool Suite* hat mehrere Beschreibungsdokumente für die GUI-Elemente, die eine Interaktion des Benutzers mit Kernfunktionalitäten des Fujabas ermöglichen. Die Erweiterung der *Fujaba Tool Suite* GUI erfolgt mit GUI-Elemente der Plug-Ins.

Plug-In JAR-Paket

Jedes Plug-In wird als eine gepackte JAR-Datei weitergegeben. Die gepackte JAR-Datei enthält die Klasse, die die PluginInterface Schnittstelle implementiert oder die abstrakte Klasse **AbstractPlugin** erweitert. Diese Datei wird im XML-Beschreibungsdokument des Plug-Ins angegeben. Standardmäßig wird die JAR-Datei wie das Wurzelverzeichnis des Plug-Ins benannt. Die Jar-Datei muss genauso benannt werden, wie Sie im *plugin.xml* Dokument angegeben ist.

4.4 Ein Fujaba Plug-In-Generator - PluginCreator

Der Fujaba Plug-In-Generator ist ein Fujaba Plug-In, das dem Entwickler die im Kapitel 4.3 beschriebene Dateistruktur generiert. Der *PluginCreator* generiert die XML-Dokumente, eine Klasse, die die abstrakte **AbstractPlugin** Klasse erweitert und weitere Klassen für die GUI-Elemente, die im generierten XML-Dokument beschrieben sind.

Der *PluginCreator* generiert einen Rahmen für ein Fujaba Plug-In. Er generiert keine Logik-Klassen und keine Meta-Modelle. Der *PluginCreator* soll dem Entwickler die Arbeit abnehmen, selbst per Hand die XML-Dokumente und Schnittstellenimplementierung zu schreiben. Eine ausführliche Benutzerinteraktion mit dem *PluginCreator* ist im Anhang beschrieben.

4.5 Anpassung und Erweiterung vorhandener Konzepte

Bei der Entwicklung und Realisierung des Erweiterungsmechanismus wurden einige Anpassungen und Erweiterungen an vorhandenen Konzepte vorgenommen. Zunächst werden die betroffenen Konzepte API beschrieben und die vorgenommenen Änderungen erläutert.

4.5.1 Erweiterung und Anpassungen an Klassenlader

Um das Laden von Bibliotheken mit unterschiedlichen Versionen durch unterschiedliche Plug-Ins zu gewährleisten, wurde der Klassenlader von Fujaba dahingehend angepasst, dass jedes Plug-In eine Referenz auf seinen eigenen Klassenlader erhält. Bei der Instanzierung von Plug-Ins wird ein Klassenlader durch Eingabe der Plug-In ID erzeugt und in eine Hashtabelle abgelegt. Mit dem Verwalten der Klassenlader-Instanzen in einer Hashtabelle ist sichergestellt, dass für jedes Plug-In genau ein Klassenlader zuständig ist.

Da für jedes Plug-In genau ein Klassenlader zuständig ist, ist es ohne weiteres nicht möglich, dass ein Plug-In Ressourcen eines anderen Plug-Ins, von dem es abhängt, laden kann. Für dieses Problem wurde eine Zuständigkeitskette (Chain-of-Responsibility) im Klassenlader implementiert. Falls das Laden einer Ressource durch den eigenen Klassenlader fehlschlägt, wird die Kette durchlaufen, um die Ressource zu laden.

4.5.2 Anpassungen am Speichermechanismus und der Projektverwaltung

Mit dem dynamischen Erweiterungsmechanismus ist es möglich, dass Plug-Ins eigene Diagramme in Fujabas Projektdatei speichern. Da beim Speichern von Plug-In Diagrammen keine Information über das entsprechende Plug-In und seine Version mit gespeichert wurde, ist der Speichermechanismus geändert worden, so dass Plug-In Einträge in Fujabas Projektdatei um entsprechende Informationen ergänzt worden. Diese Informationen sind nötig, um festzustellen, welche Plug-Ins benötigt werden, damit die erstellte Projektdatei geladen werden kann.

Die Informationen über Fujaba und Plug-Ins werden im folgenden Format in der Projektdatei gespeichert:

```
# Used Fujaba core
$;The Fujaba kernel;fujaba.core;4;1;0
# used plug-ins
$;My Plugin;de.upb.myplugin.MyPlugin;1;0;0
$;Pattern Specification;de.upb.patternspecification.PatternSpecPlugin;1;1;0
```

Kommentare beginnen in Fujabas Projektdatei mit einem "#" Zeichen. Für das Projekt benötigte Fujaba Tool Suite und Plug-Ins beginnen mit einem "$" Zeichen. Dabei besteht die Zeile aus folgenden Einträgen mit ";" als Trennzeichen:

Name des Plug-Ins oder der Anwendung;Die ID des Plug-Ins oder der Anwendung;Die Major-Nummer;Die Minor-Nummer;Die Build-Nummer

Beim Laden der Projektdatei wird überprüft, ob die Anwendung (*Fujaba Tool Suite*) und Plug-Ins in den richtigen Versionen verfügbar sind. Falls dies nicht der Fall sein sollte, wird in einer Fehlermeldung ausgegeben, welche Plug-Ins bzw. welche Versionen von Plug-Ins benötigt werden, um die entsprechende Projektdatei zu laden.

Alle in der Projektdatei gespeichertern Plug-In Einträge werden mit der Information der Plug-In ID ergänzt, um mit dieser Information den Klassenlader des Plug-Ins zu ermitteln und die Einträge über diesen ermittelten Klassenlader zu laden.

Der angepasste Speicher- und Lademechanismus von Fujaba ist abwärtskompatibel. Projektdateien, die mit früheren Versionen von Fujaba wie Fujaba 3.x können ebenfalls geladen werden. Die geladenen Projektdateien der früheren Fujaba Versionen werden beim Speichern im neuen Format gespeichert.

4.6 Ein Hilfesystem für das Fujaba Plug-In

Damit der Anwender von der *Fujaba Tool Suite* eine Unterstützung bei der Erledigung seiner Arbeit erhält, wurde im Rahmen dieser Arbeit ein auf HTML-basiertes Hilfesystem für Plug-Ins implementiert. Das Hilfesystem besteht aus einem HTML-Browser, der die Plug-In Dokumentation und Hilfeseiten darstellt.

5 Beispielsitzung

Dieses Kapitel behandelt die Interaktion des Erweiterungsmechanismus in Fujaba mit dem Benutzer. Der Anwender kann benutzerdefinierte Einstellungen bezüglich des Ladens und Verwaltens von Plug-Ins vornehmen und diese auch in späteren Sitzungen verwenden.

5.1 Plug-Ins Optionen und Sitzungsdaten

Die Abbildung 5.1 zeigt die Umgebungsoptionen für Fujaba Plug-Ins. Dem Anwender stehen zwei Listen zur Verfügung - eine Liste für Verzeichnisse, in denen nach Plug-Ins gesucht wird und eine Liste für URLs, die für das Herunterladen von Plug-Ins verwendet werden.

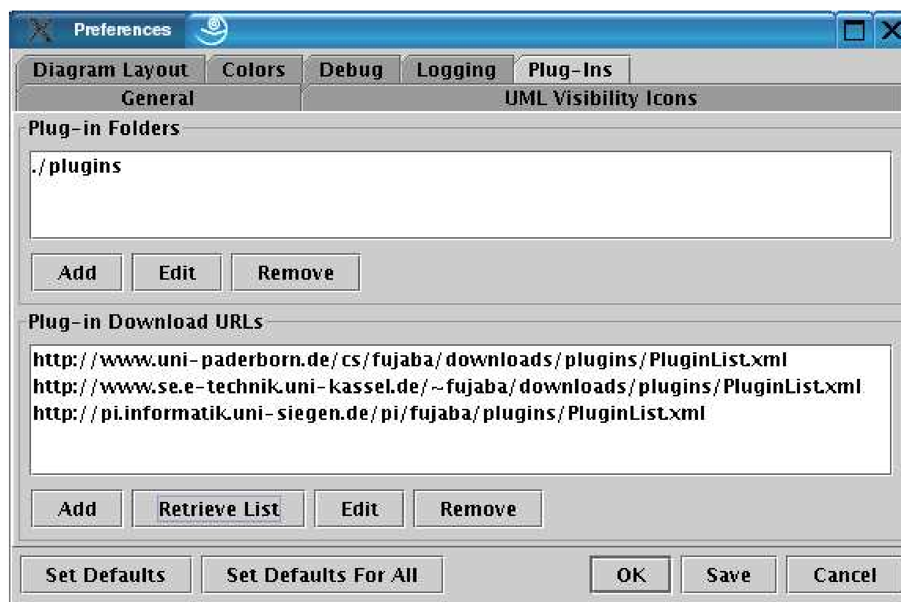

Abbildung 5.1: Dialog um Umgebungsoptionen für Plug-Ins zu setzen

Es gibt sowohl ein voreingestelltes Verzeichnis als auch eine voreingestellte URL, wie in Abbildung 5.1 dargestellt. Diese vordefinierten Einstellungen können durch den Anwender weder gelöscht noch geändert werden. Es können jedoch beliebig

viele Verzeichnisse und/oder URLs mit der jeweiligen *Add* Schaltfläche hinzugefügt werden. Der **PluginManager** sucht in den eingegebenen Verzeichnissen nach Plug-Ins, um diese in Fujaba zu laden.

Die Liste der URLs werden dem Erweiterungsmechanismus über die Implementierung der **KernelInterface** Schnittstelle zur Verfügung gestellt, um eine Installation und Update von Plug-Ins über das Internet zu ermöglichen. Das Kapitel 3.5 beschreibt, wie der Download und Installationsmechanismus funktioniert.

5.2 Informationen über geladene Plug-Ins

Der Informationsdialog, wie in Abbildung 5.2 zu sehen, zeigt Informationen über geladene Plug-Ins. Die Informationen über die geladenen Plug-Ins werden aus dem jeweiligen *plugin.xml* Dokumenten gewonnen.

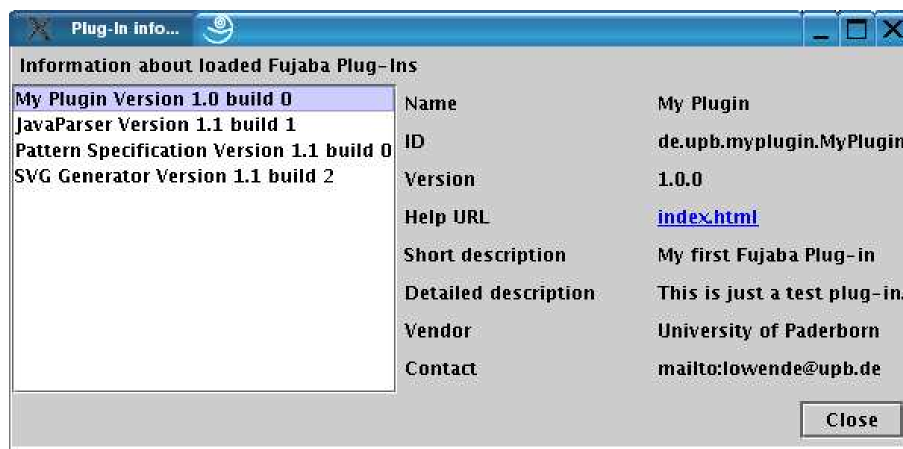

Abbildung 5.2: Informationen über geladene Plug-Ins

Der Informationsdialog über geladenen Plug-Ins wird über das Menü *Help – Plug-ins Info* aufgerufen. Über diesen Dialog kann das Plug-In Hilfesystem aufgerufen werden, falls das Plug-In ein Hilfesystem anbietet.

5.3 Update und Installation von Plug-Ins über das Internet

Der Download-Dialog enthält die Plug-In Informationen über geladene und neue Plug-Ins, die über die *Download/Install* bzw. *Update* Schaltfläche heruntergeladen und installiert werden.

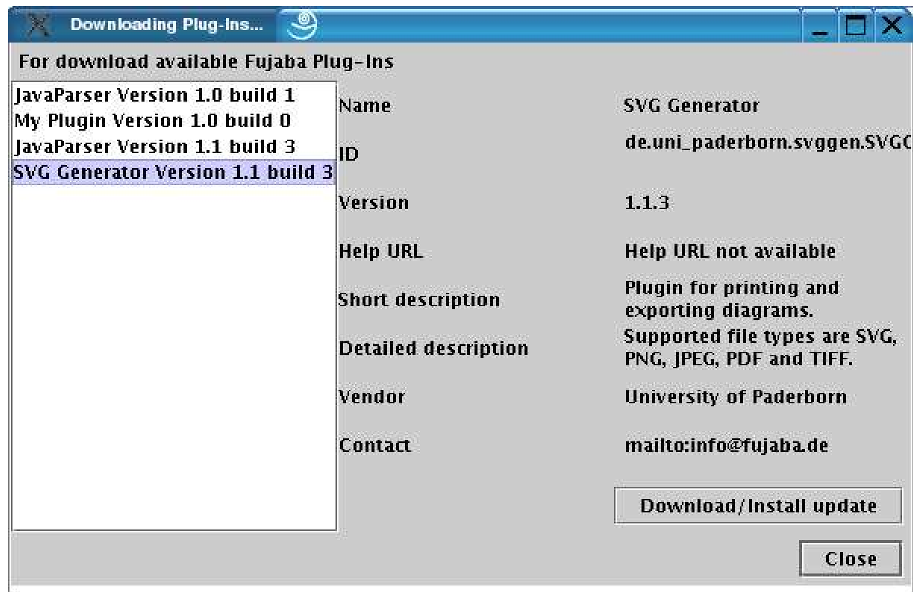

Abbildung 5.3: Dialog um neue Fujaba Plug-Ins herunterzuladen bzw. einen Update vorzunehmen

Der Download-Dialog, mit dem Plug-Ins heruntergeladen und installiert werden können, wird über das Menu *Help – Download Plugins* aufgerufen. Die Abbildung 5.3 zeigt den Dialog zum Herunterladen und Updaten von Fujaba Plug-Ins. Durch das Klicken auf die *Download/Install* bzw. *Update* Schaltfläche wird das jeweilige Plug-In heruntergeladen und in das vordefinierte Plug-Ins-Verzeichnis ($FUJABA/plugins) von Fujaba installiert.

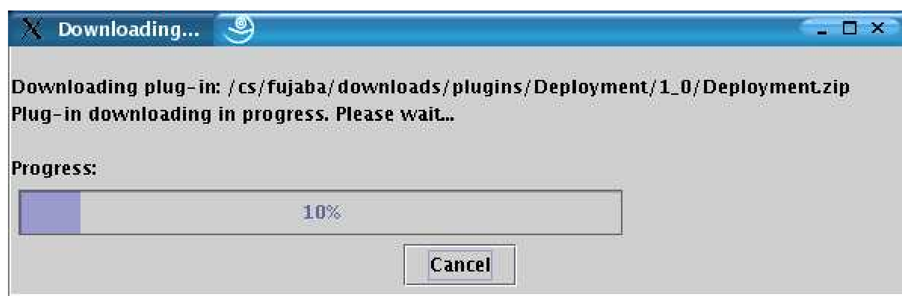

Abbildung 5.4: Dialog über den Download- bzw. Installationsprozess

Der Download- bzw. Installations-Prozess wird in einem weiteren Dialog (siehe Abbildung 5.4) dargestellt, welcher dem Benutzer jeder Zeit die Möglichkeit bietet, den Download-Prozess bzw. die Installation abzubrechen. Beim Abbruch werden sämtliche Dateien, die installiert wurden, wieder gelöscht.

5.4 Speicherung der Plug-In Sitzungsdaten

Abbildung 5.5 zeigt einen Dialog, mit dem der Anwender die Möglichkeit hat, Plug-In spezifische Sitzungsdaten zu speichern.

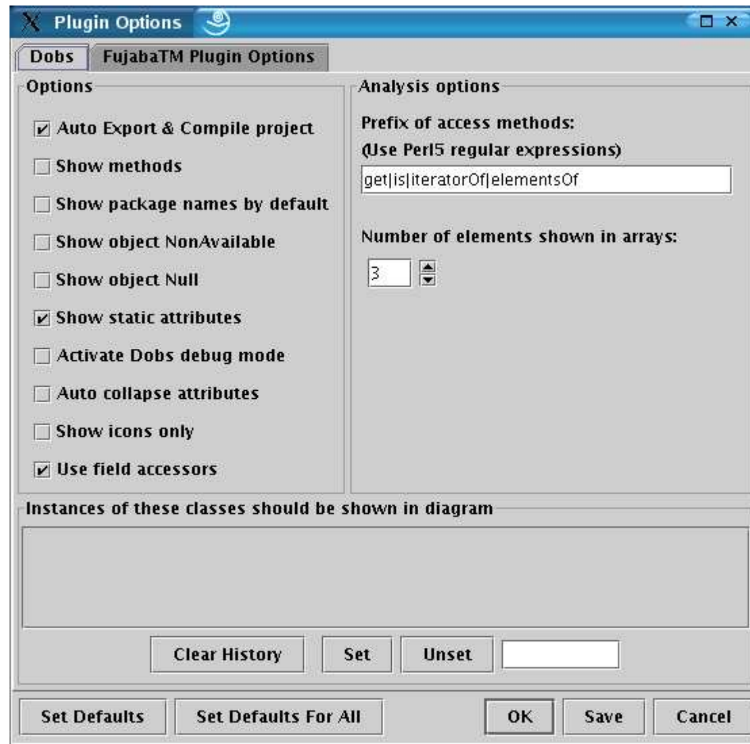

Abbildung 5.5: Dialog über plug-in-spezifische Sitzungsdaten

Die Sitzungsdaten werden in eine für das Plug-in spezifische Datei abgelegt, um diese in späteren Sitzungen wieder zu verwenden. Für detailierte Beschreibung siehe die Fujaba- und Plug-In-Umgebungen.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

Mit dem dynamischen Erweiterungsmechanismus wurden die Probleme bzw. Nachteile von Fujaba, die in Kapitel 1.1 ausführlich beschrieben wurden, zum größten Teil vollständig und in einigen Fällen teilweise gelöst bzw. behoben. Als erstes wichtiges Ziel wurde erreicht, Fujaba zu modularisieren und somit sämtliche Nachteile eines monolithischen Software-Systems zu beheben.

Mit der Anwendungsunabhängigkeit des dynamischen Erweiterungsmechanismus ist es möglich, dass der Mechanismus ohne Änderungen auch für andere Anwendungen benutzt werden kann. Die Anwendungen, die den Mechanismus integrieren möchten, müssen lediglich eine kompakte Schnittstelle implementieren. Mit dieser Eigenschaft ist es möglich, Entwicklungsumgebungen und Frameworks weiter zu modularisieren. Damit kann ein hoher Modularisierungsgrad des Softwaresystems erreicht werden.

Der Download- und Installationmechanismus bietet eine GUI-basierte Interaktion mit dem Benutzer, um eine benutzerfreundliche Möglichkeit zu gewährleisten, Plug-Ins herunterzuladen und zu installieren. Der Benutzer muss den Mechanismus nicht in seinen technischen Details kennen.

Die Plug-Ins haben jeweils einen eigenen Klassenlader, so dass gleiche Bibliotheken in unterschiedlichen Versionen von unterschiedlichen Plug-Ins benutzt werden können. Somit können Fehler insbesondere bezüglich des Ladens und Speicherns von Ressourcen isoliert und schneller gefunden werden.

Das eingebaute Versions- und Konfigurationsmanagement ermöglicht Plug-In Abhängigkeiten aufzulösen und die dazugehörigen Zuständigkeitsketten zu erzeugen. Es wird auch erreicht, dass die Schnittstellenkompatibilität zwichen Fujaba und Plug-Ins und zwischen Plug-Ins untereinander eingehalten wird.

6.2 Ausblick

Mit dem Erweiterungsmechanismus wurde erreicht, dass Fujaba schlanker, zuverlässiger, und sicherer ausgeführt werden kann. Desweiteren kann es auf individuelle Bedürfnisse und Wünsche des Anwenders angepasst werden. Auch die Entwickler können von diesem Mechanismus profitieren, denn sie können sich jetzt gezielt mit den Teilen des Systems beschäftigen, für die sie sich interessieren. Dadurch wird ein enormer Aufwand der Einarbeitung gespart. Zudem können jetzt Entwickler noch flexibler und effizienter an dem für sie relevanten Modul arbeiten. Mit Hilfe des *PluginCreator* Plug-In können Plug-Ins schneller und effizienter entwickelt werden.

Für die Zukunft ist es geplant, den dynamischen Erweiterungsmechanismus um das Konzept des so genannten *Hot-pluging* zu erweitern. Mit diesem Konzept ist es möglich, Plug-Ins während der Laufzeit der Anwendung nachzuladen. Damit braucht der Anwender nicht nach jedem Plug-In Download Fujaba neu starten.

Ein weiterer Plan für die Zukunft ist, dass ein Plug-In mit einer Menge von Plug-Ins, die beispielsweise von diesem abhängen, das Einsatzgebiet einer Konfiguration der *Fujaba Tool Suite* kennzeichnet. Ein Realtime Framework Plug-In für die *Fujaba Tool Suite* mit Realtime Statechart, UML-RT und Modelchecking Plug-In usw. kennzeichnet das Einsatzgebiet der *Fujaba Tool Suite* in Echtzeit-Umgebungen. Die Kennzeichnung kann beispielsweise durch einen Startbildschirm, Informations-Dialog über Fujaba usw. erfolgen. So können unterschiedliche Fujaba Konfigurationen für unterschiedliche Einsatzgebiete erstellt werden, ohne dabei am Kern von *Fujaba Tool Suite* fundamentale Änderungen vorzunehmen. Eine *Fujaba Tool Suite* für Reengineering, oder für Realtime-Umgebungen wären zwei mögliche Beispiel-Konfigurationen.

A Fujaba Plug-in creator [English]

The **PluginCreator** plug-in is a Fujaba plug-in that generates plug-in templates for Fujaba. It also can be run as stand alone application. The **PluginCreator** provides 3 steps to generate plug-ins rapidly. At first you have to describe your plug-in to be created in first step. In the second step you define which plug-ins and libraries your plug-in depends on and in the third step you have to define and add actions (menu items, buttons etc.) to extend the Fujaba core gui by the plug-in you want to create. The following sections describe the steps detailed.

A.1 Plug-in description [Step 1]

This section will guide you to describe plug-in you want to create with **PluginCreator**. The description of plug-in can be done in two steps.

Figure A.1: Dialog to generate plugin.xml document

A.1.1 1st Step

You have to set in this step the fundamental plug-in properties, so that created plug-in can be found and instantiated (loaded into) by Fujaba. The figure A.1 shows you which properties and fields have to be set to generate the plugin.xml document. The generated plugin.xml document will be used by the plug-in manager of Fujaba to parse the plug-in into Fujaba and instantiate it.

The table A.2 describes each property (field) more detailed.

Property (field)	**Description of property**
Plug-in name	This field will be used to give your plug-in a name. The name can be arbitrary but we recomend to name your plug-in like the base directory of your plug-in.
Plug-in base directory	Default, the created plug-in will be stored in $YOUR_HOME_DIR/PLUGIN_NAME, where $YOUR_HOME_DIR is your home directory and PLUGIN_NAME the plug-in name field you have edited.
Library	The jar file that contains the plug-in class field (the class which extends the AbstractPlugin class). The name of the library should not contains spaces and the name should be conform to the allowed file names of your os.
Package	The package is the package prefix to create your plug-in. The directories for your packages will be created automaticly in the plug-in base directory. This field will be also used to create packages for actions, options etc.
Plug-in class	This field is for the class which will extend the Abstract-Plugin class. The Plug-in Class will be geneated automaticly and stored in the generated package directory structure.
Plug-in ID	This field is not editable by the user. The plug-in id will be constructed from package field concatenated with Plug-in class field.

Figure A.2: Table of plug-in properties

A.1.2 2nd Step

In the second step by clicking on **...** button you can set the extended plug-in properties. You have to define here the initial plug-in version, short and detailed

description of your plug-in.

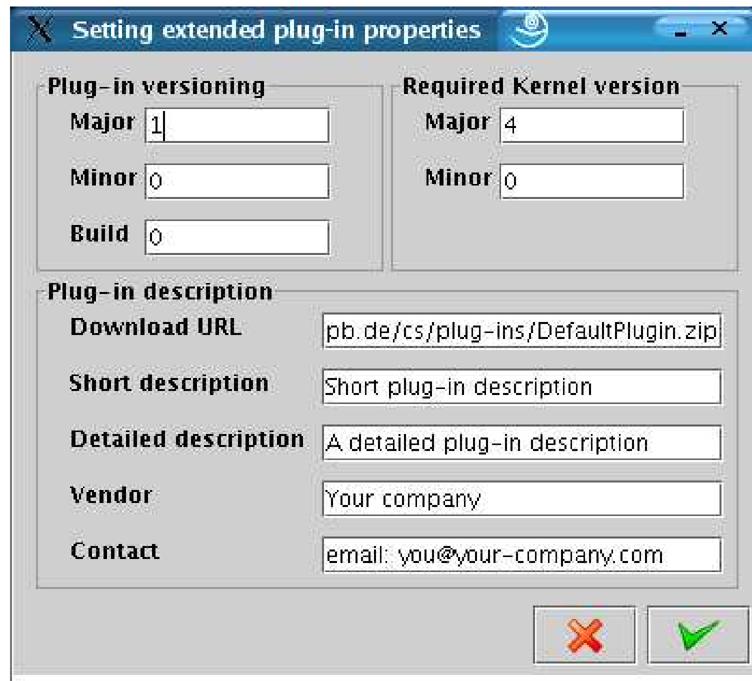

Figure A.3: Extended plug-in property dialog

Moreover you have to define which Fujabe kernel your plug-in will require. The figure A.3 shows wich fields have to be set. The table A.4 describes each property.

A.2 Plug-in requirement [Step 2]

This section will guide you how to define plug-in requirement. If your plug-in do not require any library or plug-in you can skip reading this section and go to the third step to generate stable.xml document. The **PluginCreator** provides a dialog (see figure A.5) to add required libraries and plug-ins. By clicking on the add button a dialog is showed to define a required plug-in. The required plug-in is specified with a plug-in id attribute and version number (major and minor number). The figure A.6 shows that attributes. The fields in figure A.6 are described in the table A.7.

By clicking on the Add button of the required library panel you get a file select dialog to select a library. Only jar files can be added to the required libraries.

Property (field)	**Description of property**
Major	The plug-ins major number.
Minor	The plug-ins minor number.
Build	The plug-ins build number.
Kernel major number	Major number of required Fujaba kernel.
Kernel minor number	Minor number of required Fujaba kernel.
URL	The URL where your plug-in can be found on www. You can let this field by default but should not be empty. This field will be used by the fujabas plug-in downloader to download and update plug-ins. For loading and instantiating of plug-ins in Fujaba this field is not relevant.
Short description	A short description of your plug-in.
Detailed description	A detailed description of your plug-in.
Vendor	Vendor of the plug-in.
Contact	The contact adress i.e e-mail adress.

Figure A.4: Table of plug-in properties

The selected library will be copied to $PLUGIN_BASE_DIR/libs, where the $PLUGIN_BASE_DIR is defined in section A.1.

A.3 Generating stable.xml document [Step 3]

To generate the stable.xml document you have to define actions (menu items, buttons etc.) which will extend the Fujaba core menu. The stable.xml document extends the Fujaba core gui by the in this step added actions. At least one action will be added by default. The actions you add have to be modified to make a sence by invoking that actions because only templates for action will be created. The figure A.8 shows the dialog that contains a list to add/remove actions. By clicking the add button actions can be added to the list. The dialog to add actions is shown in figure A.9. The fields of the dialog are described in table A.10.

Figure A.5: Plug-in requirement dialog

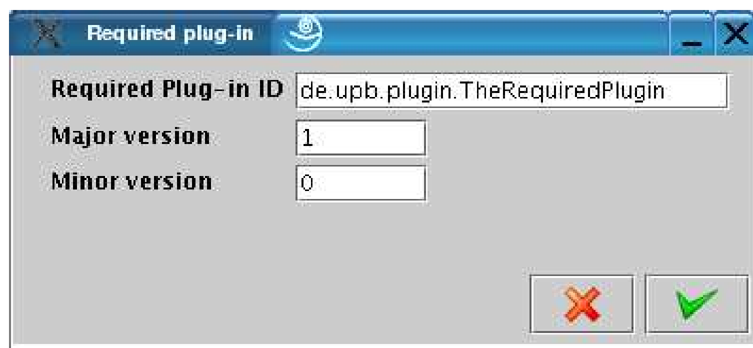

Figure A.6: Dialog to add required plug-ins

Property (field)	Description of property
Required Plug-in ID	The id of a plug-in that your plug-in should depends on. Important, if you define a plug-in dependency and the required plug-in could not be found and loaded by Fujaba then the loading of your plug-in will also be skipped. The required plug-in id should be correct and available. You can get the plug-in id by copying the id attribute of the corresponded plugin.xml document.
Major	The major number version of the required plug-in will be required by your plug-in
Minor	The minor number version of the required plug-in will be required by your plug-in

Figure A.7: Table of required plug-ins attributes

Figure A.8: Dialog to generating stable.xml document

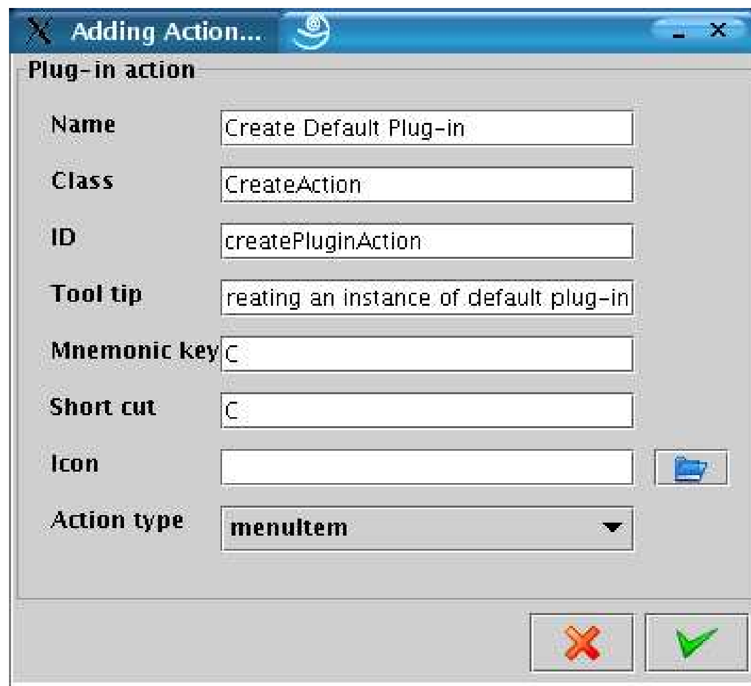

Figure A.9: Dialog to add actions

Property (field)	**Description of property**
Action name	The action name will be displayed in the Fujaba menu.
Class name	The class that extends the AbstractAction to handle the action. The actionPerformed(ActionEvent e) methode of this class will be invoked by executing the action.
ID	The action id. The id have to be unique in the whole stable.xml and the fujabas core gui (UserInterface.xml) documents. This field will be used in stable.xml documents to reffer to that action.
Tool tip text	Tool tip text that have to be showed by drawing mouse over the action item (menu item, button etc.).
Mnemonic key	The mnemonic key to underline a character in action name field. By pressing the specified key the action will be invoked.
Short cut	The short cut to invoke the action i.e. Ctrl+C
Icon	An icon to iconify the action (menu item, button etc.). You can use the file selection dialog to choose an icon by clicking the file open button. The selected icon will be copied in to the $PLUGIN_BASE_DIR/images if exists. If the field is empty the Fujabas none.gif icon will be used.
Action type	You have to choose an item from the action type combo box. The action type can be menuItem, radioButton, checkBox.

Figure A.10: Table of action attributes

Literatur

[ADC04] APPLE DEVELOPER CONNECTION, : *Cocoa: Dynamically Loading Code.* : http://developer.apple.com/documentation/Cocoa/Conceptual/LoadingCode/Concepts/Plugins.html: Apple Inc., January 2003: last visited April 2004

[Eli00] ELIËNS, Anton: *Principles of Object-Oriented Software Development.* 2nd Edition. Addison-Wesley, 2000

[LVM95] LUCKHAM, David C. ; VERA, James ; MELDAL, Sigurd: Three Concepts of System Architecture / Stanford University. July 1995 (CSL-TR-95-674). – Forschungsbericht

[OTI04] OBJECT TECHNOLOGY INTERNATIONAL, Inc.: *Eclipse Platform Technical Overview Version 2.1.* : http://www.eclipse.org/whitepapers/eclipse-overview.pdf: Object Technology International, Inc., February 2003: last visited April 2004

[Sac99] SACHWEH, S.: *Ein Kooperatives Konfigurationsmanagementsystem*, Universität-Gesamthochschule Paderborn, Diss., March 1999

Index

Ausblick, 40

Eclipse Plattform-Architektur, 6

Fujaba Plattform-Architektur, 5
Fujaba Tool Suite, 5
Fujaba-Grundlagen, 5

Interaktion, 35

Kernel-Interface, 22

Lösung, 3

plug-in creator, 41
Plug-In Downloads, 36
plug-in gui, 44
Plug-In Manager, 25
plug-in requirement, plug-in dependency, 43
Plug-In Sitzungsdaten, 38
Plug-In Update, 36
Plug-Ins, 18
Plug-Ins Optionen, 35
Plug-Ins Schnittstelle, 23
plugin-info, 36
plugin.xml, plug-in description, 41

Sitzungsdaten, 38
Struktur, 3

Tool-Schnittstelle, 22

Versions- und Konfigurationsmanagement, 7

Werkzeugbau, 6

Zusammenfassung, 39