TUHH
Technische Universität Hamburg-Harburg

STS

Erweiterung eines Kompilierer-Rahmenwerks zur Generierung von konzeptorientierten Inhaltsverwaltungsanwendungen

Diplomarbeit

Mathias Freier

Betreut von:
Prof. Dr. J.W. Schmidt
Dr. H.-W. Sehring

Februar 2005
Technische Universität Hamburg-Harburg
Software Systems Group (STS)

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Kode-Beispiele

KAPITEL 1

Einleitung

In der vorliegenden Arbeit geht es darum, ein Übersetzer-Rahmenwerk zu erweitern. Dieses Rahmenwerk startet Generatoren, die aus einem Modell Inhaltsverwaltungs-Systeme für beliebige Plattformen generieren. Der Übersetzer wird, den Anforderungen der herauszustellenden Benutzergruppen entsprechend, in eine Ablaufumgebung eingebettet, so dass die Handhabung wesentlich vereinfacht wird. Zudem wird die Funktionalität des Übersetzer-Rahmenwerks erweitert und dessen Ablauf für den interaktiven Betrieb optimiert. Ein wesentlicher Bestandteil dieses Projektes ist die Benutzerführung. Arbeitsprozesse werden von der neuen Software begleitet oder auch automatisiert.

1.1 Einführung in die Konzeptorientierte Inhaltsverwaltung

Dass Computer zum Speichern und Bereitstellen von Daten Verwendung finden, ist hinlänglich bekannt. Die Archivierung geht oft schon über *die* von losgelösten Inhalten hinaus.

Ein Beispiel: Es sei ein ***Liebesbrief*** ein Inhalt. Nun kann ein Computer angewiesen werden, diesen Brief zu speichern und bei Bedarf wieder anzuzeigen. Der Speichernde betrachte den Brief in einem bestimmten Kontext und habe eine ***Intention*** diesen Brief zu speichern. Nehme man an, ein Betrachter - zu einem späteren Zeitpunkt - lese diesen Brief in einem anderen Kontext[1].

Könnte dieser Brief nicht Beleg eines historischen Ereignisses, eines Verrates vielleicht, gewesen sein? Oder sollte dargestellt werden wie poetisch Briefe verfasst werden können?

Die ***Intention***, mit der jener Brief gespeichert würde, ist verloren gegangen.

Obiges Beispiel verlangt danach, dass Inhalte zusammen mit ihrem *Konzept*

[1] Es kann sich auch um die selbe Person handeln.

gespeichert werden. Ein neuer Ansatz ist die konzeptorientierte Inhaltsverwaltung nach [1], wie es in der Analyse im Abschnitt 2.1 vorgestellt wird.

1.2 Generatortechnologie zur Systemerzeugung

Anwendungen zum Umgang mit konzeptorientierten Inhalten werden individuell zur Struktur der zu speichernden Daten erzeugt. Dazu erstellt ein Anwender – im Folgenden Modell-Definierer genannt – ein Modell zur Repräsentation eines Konzeptes zur Inhaltsverwaltung. Bei dem Modell handelt es sich um eine Kodedatei der in [1] vorgestellten Sprache. Ähnlich wie auch bei Programmiersprachen wie Java, C, etc. müssen diese Quelldateien kompiliert werden. Genauer gesagt werden die Anwendungen zum Umgang mit konzeptorientierten Inhalten passend zu den Modellen generiert.

Auch nach Erzeugung des Systems kann das Modell, das dem System zugrunde liegt, als Vorlage einer Abänderung (Verfeinerung) dienen. Das aus der Verfeinerung generierte System beinhaltet das System des Originals. Das System verhält sich evolutionär.

Die Speicherung der Inhalte und Konzepte wird nicht auf eine bestimmte Zielplattform beschränkt. Die Systemerzeugung wird durch ein Rahmenwerk und durch zur Zielplattform passende Generatoren realisiert. Ein solches Übersetzer-Rahmenwerk wurde zu [1] als Prototyp implementiert. Es handelt sich hierbei um ein durch Kommandozeilen gesteuertes Programm.

1.3 Erweiterung durch Integration in eine Entwicklungsumgebung

Durch Betrachtung der Benutzerrollen von Personen, die mit dem oben beschriebenen Modell-Übersetzer arbeiten (siehe Abschnitt 2.5), wird der Bedarf einer Vereinfachung der Benutzung deutlich. Diese Rollen, sowie aus deren Arbeitsprozessen resultierende Anforderungen an notwendige Programmartefakte, werden analysiert. Es entsteht eine Erweiterung des Rahmenwerks mit Elementen, die dem Benutzer die Arbeit erleichtern.

Die Erweiterung umfasst das bereits bestehende Übersetzer-Rahmenwerk mit optimierter Ablaufsteuerung und zusätzliche Funktionalität. Es wird ein konzeptionelles Design erstellt (siehe Kapitel 3), das auf die Anforderungen der Benutzer eingeht.

Für eine Implementation der Erweiterung werden mögliche Technologien evaluiert. Entsprechend den Anforderungen der Benutzer wird der Übersetzer (Übersetzer-Rahmenwerk sowie Analyse-Komponenten und Generatoren) in die Plattform und Entwicklungsumgebung *eclipse* eingebunden.

Der entwickelte Prototyp unterstützt sowohl die Arbeitsprozesse der Modell-Definierer (siehe oben), als auch die der Komponenten-Entwickler (Programmierer der Generatoren und anderer Komponenten). Beide Rollen werden im Abschnitt 2.5 genauer erläutert.

Modell-Definierer werden bei der Erstellung von Asset-Modellen unterstützt. Die Erzeugung von Modellen und Konfigurationen für die Systemgenerierung wird durch Dialoge, Wizards und automatisierte Funktionen erleichtert. Die Systemerzeugung geschieht dann komfortabel innerhalb eclipses. Dabei wird dem Benutzer Übersicht über den Prozessfortschritt gegeben, mit der Möglichkeit den Prozess zu unterbrechen.

Bei dem vom Übersetzer Produzierten handelt es sich um Quellkode, dieser muss dann zumeist von anderen Kompilierern – z.B. dem Java-Übersetzer – weiterverarbeitet werden. Daraufhin muss die neue Anwendung dorthin verteilt werden, wo sie auch zum Einsatz kommt. Für diese Weiterverarbeitung kann die erweiterte Plattform benutzt werden.

Komponenten-Entwickler können von den Funktionen der Entwicklungsumgebung profitieren und ihre Komponente komfortabel programmieren, einbinden und testen. Von der Komponente generierte Quellkode-Module können automatisch von entsprechenden anderen Kompilierern weiterverarbeitet werden, so dass Fehler in der Generierung unter Umständen gleich offensichtlich werden. Sämtliche Arbeitsprozesse des Komponenten-Entwicklers können innerhalb der Entwicklungsumgebung ausgeführt werden. Die Arbeit der Entwickler wird durch das erweiterte Rahmenwerk beschleunigt und teilweise automatisiert. Dadurch kann sich die Aufmerksamkeit des Entwicklers auf die eigentliche Aufgabe der Komponente richten. Eine interne Laufzeitbeobachtung ermöglicht es den Entwicklern nachzuvollziehen was innerhalb des Rahmenwerks geschieht. Zusätzlich hilft eine angepasste Umgebung für automatisierte Tests bei der Eingrenzung von Fehlern in der Komponenten-Entwicklung.

Es werden verschiedene Arten der Hilfestellung für die Benutzer realisiert. Dabei wird darauf geachtet, möglichst viele Konzepte zur Benutzerführung zu verwenden. Es wird jedoch kein Anspruch auf inhaltliche Vollständigkeit der Benutzerführung erhoben. Eine spätere Erweiterung des Prototyps wird somit vereinfacht, da die vorhandenen Implementationen als Vorlage dienen können.

Die Integration der Komponenten (Analyse-Komponenten und Generatoren) und deren Ablaufsteuerung ist vollständig durch ein Plugin realisiert worden. Grafische Elemente wie Dialoge, Ansichten und Einstellungsseiten erlauben eine Interaktion des Rahmenwerks und der Komponenten mit dem Benutzer.

Der entwickelte Prototyp wird bereits von Komponenten-Entwicklern verwendet und wird weiterentwickelt werden.

1.4 Überblick über die Arbeit

In der Analyse (Kapitel 2) wird die Idee konzeptorientierter Inhaltsverwaltungssysteme vorgestellt und anschließend wird der Prozess der Systemerzeugung beschrieben. Dazu findet aus technischer Perspektive und aus Sicht der Anwender eine Analyse der Arbeitsprozesse statt. Es werden im Abschnitt 2.5 Benutzerrollen definiert und resultierende Forderungen formuliert, die es zu erfüllen gilt. Anschließend werden im Abschnitt 2.7 Technologien betrachtet, die als Basis einer Erweiterung dienen könnten. Die Wahl eclipses als Integrationsplattform

wird erläutert.

Im Design (Kapitel 3) wird ein erweitertes Konzept zur Systemerzeugung vorgestellt. Die Architektur der Implementationsplattform eclipse wird im Abschnitt 3.4 beschrieben. In den folgenden Abschnitten werden in eclipse realisierte Konzepte vorgestellt, die das Design der Erweiterung bestimmen.

Die Implementation (Kapitel 4) zeigt einen Überblick über das realisierte Plugin auf Ebene der Klassen und Pakete. Es wird auf Details der Implementierung eingegangen. Automatisierte Tests werden in diesem Kapitel im Abschnitt 4.7 erstmals vorgestellt und deren Verwendung aufgezeigt.

Das Kapitel 5, mit dem Titel Benutzerführung, beschäftigt sich mit Maßnahmen, die zur Anleitung von Anwendern durch das Programm oder, in der Phase der Inbetriebnahme, persönlich, erarbeitet wurden. Die Werkzeuge, die die Benutzer unterstützen, basieren auch auf den in der Analyse (Kapitel 2) definierten Anforderungen.

Das Kapitel 6 beinhaltet eine Bewertung des Projektes und zeigt weiterführende Aufgaben auf.

KAPITEL 2

Analyse

In diesem Kapitel wird das vorhandene Rahmenwerk, wie es einleitend beschrieben wird, analysiert und es werden Anwendungsfälle betrachtet, die zu der im Rahmen dieser Arbeit vorgenommenen Erweiterung des Systems führen.

Zuerst wird konzeptionell untersucht, aus welchen Komponenten der Übersetzer besteht. Die Vorgehensweise der Systemerzeugung wird aus technischer Sicht des Übersetzers und in Bezug auf den Arbeitsprozess des Benutzers betrachtet. Es werden anschließend Benutzerrollen erarbeitet und Anforderungen an zukünftige Erweiterungen des Übersetzers herausgestellt.

Die Analyse schließt mit einem Einblick in vorhandene Technologien, auf denen eine Erweiterung des Übersetzers basieren könnte, und vergleicht diese untereinander.

2.1 Konzeptorientierte Inhaltsverwaltung: COCoMa

Was denn mit konzeptorientierter Inhaltsverwaltung[1] gemeint ist, wird im Folgenden auf Basis der Arbeit [1] erklärt. An dieser Stelle wird auf die sehr aufschlussreichen, einleitenden Kapitel dieser Arbeit verwiesen. Um den Rahmen dieser Zusammenfassung nicht zu sprengen, wird auf vieles bei der Reproduktion der Gedankengänge verzichtet.

Einleitende Gedankengänge der Arbeit sind:

Es reicht dem Menschen nicht aus, beliebige Inhalte ohne Kontexte und Beziehungen, besser gesagt Konzepte zu betrachten. Es ergibt sich die Notwendigkeit mediale Inhalte gemeinsam mit deren eingrenzenden Konzepten zu bündeln. Diese Kopplung ergibt *Assets*, repräsentiert durch Inhalt plus Konzept. Der Konzeptteil besteht aus *Daten*, *Referenzen* und *Bedingungen*.

Es folgt daraus die Notwendigkeit einer Abbildung der soeben beschriebenen Darstellung auf die mathematisch-logische Ebene um eine Speicherung im

[1] Engl.: **C**oncept-**O**riented **Co**ntent **Ma**nagement, kurz: **COCoMa**

Computer zu ermöglichen.

In [1] wird ein Modell entwickelt zur Repräsentation von:

- Eigenschaften von Inhalten,
- Beziehungen zwischen Inhalten
- und Gesetzmäßigkeiten von Inhalten und Beziehungen.

Dieses Modell unterstützt die Forderungen nach Offenheit und Dynamik.

Offenheit bedeutet, dass die Darstellung nicht auf einen vorgegeben Satz von Konzepten beschränkt, sondern dass es dem Anwender möglich ist, diese individuell zu definieren.

Dynamik bezieht sich hier auf Änderbarkeit von Darstellungsformen. Dynamik ist insbesondere deswegen notwendig, weil Inhalte individuell interpretiert werden, als auch die Auffassung der Personen nicht statisch ist. Es ist demnach notwendig, ein System evolutionär zu gestalten.

2.2 Komponenten zur Erstellung von COCoMa-Systemen

Um die Offenheit (siehe oben) der Systemerzeugung gewährleisten zu können, wird individuell zum oben genannten Modell ein System generiert. Da ferner keine bestimmte Zielarchitektur vorausgesetzt wird, ist es notwendig, Generatoren durch ein Übersetzer-Rahmenwerk dynamisch anzusteuern.

Ein Übersetzer-Rahmenwerk, einschließlich der Generatoren, muss folgende Vorgehensweisen beinhalten:

- Aufnahme der neuen Quelldateien
- Aufnahme der früheren Versionen des Modells indem vorhandene Modelle kopiert und angepasst werden. Es wird in diesem Zusammenhang von Personalisierung oder auch Verfeinerung gesprochen.
- Validierung des Modells
- Konfiguration der Generatoren
- Generierung von Anwendungskode
- Generierung von Kode, der zur Anbindung des Modells zu Speichersystemen notwendig ist.
- Evtl. Weiterverarbeitung des Kodes und Verteilung der resultierenden Module.

Anders als bei einem Java-Übersetzer muss in diesem Fall der Dynamik des Systems (siehe Abschnitt 2.1) Rechnung getragen werden. Eine spezielle Architektur wird durch zu generierende Module verwirklicht, detailliert beschrieben

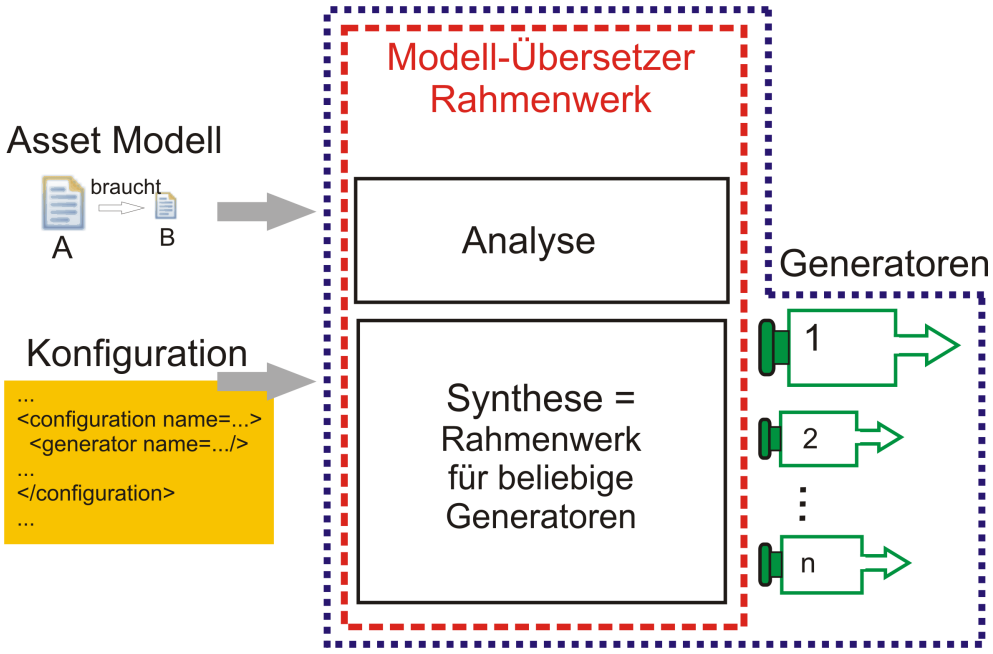


Abbildung 2.1: Komponenten des Übersetzers und Begriffsklärung

in [1]. Auf die unterschiedlichen zu erzeugenden Module wird in dieser Arbeit nicht weiter eingegangen.

Das Schaubild 2.1 zeigt eine Übersicht über die Komponenten eines Übersetzers. Der blau punktierte Rahmen umfasst alle Teile eines Übersetzers, der rot gestrichelte nur die des Kompilierer-Rahmenwerks. Das Rahmenwerk ist als fester Bestandteil eines Übersetzers zu sehen. Die Generatoren können jedoch variieren. Ein möglicher Übersetzer besteht also aus dem Kompilierer-Rahmenwerk und einer Auswahl von Generatoren.

Quelle der Kompilation sind ein Asset-Modell und eine Konfigurationsdatei, wie in der Abb. 2.1 auf der linken Seite zu sehen. Das Modell definiert die Art und Verarbeitung von Assets im zu erzeugenden System. Die Konfigurationsdatei dient zur Steuerung des Übersetzers. In ihr enthalten ist auch, welche Generatoren für die Synthesephase verwendet werden, und damit, auf welcher Zielplattform das System lauffähig ist.

Der Übersetzer wird durch die Konfigurationsdatei bestimmt.

Ausgabe des Übersetzers ist Quellkode für ein System zur konzeptorientierten Inhaltsverwaltung. Dabei kann es sich z.B. um Java-Quellkode wie auch um Anweisungen handeln, die Tabellen in Relationalen Datenbanken erzeugen können. Solche Systeme setzen auf andere Systeme auf. Das kann ein vorhandener Broker, eine Datenbank o.ä. sein. Für jede Ziel-Plattform werden ein oder mehrere Generatoren entwickelt. Die Wahl der Generatoren ist also abhängig von der Ziel-Plattform.

Die Begriffe: „Übersetzer“ „Kompilierer“ werden in dieser Arbeit gleichbedeutend verwendet. Damit ist das Kompilierer-, beziehungsweise Übersetzer-Rahmenwerk inklusiv einer beliebigen Auswahl von Generatoren gemeint. Es

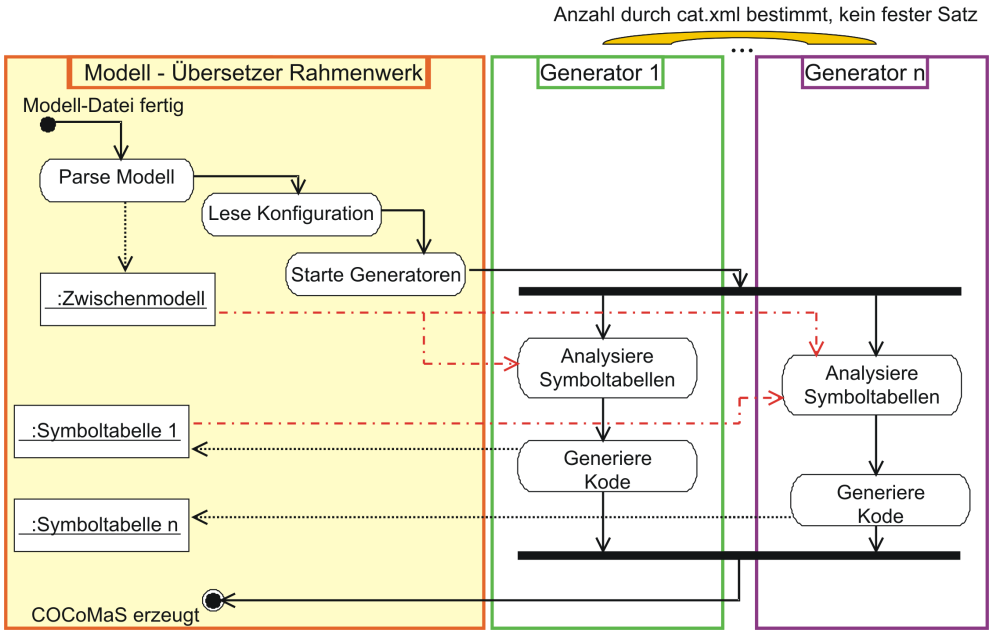


Abbildung 2.2: Vorgang der Systemerzeugung, Quelle: [1]

handelt sich hier nicht genau um eine Übersetzung im *klassischen* Sinne, vielmehr um eine Systemerzeugung, deswegen wird auch dieser Begriff in der Arbeit verwendet. Da ein Asset-Modell die Grundlage ist, wird auch von einem Modell-Übersetzer gesprochen. Wenn die Betrachtung auch die Aufgabe der Einbindung von Generatoren oder deren Erstellung umfasst, wird von dem Übersetzer Rahmenwerk auch als Übersetzer Umgebung gesprochen. Nach der Analysephase wird von einem RAHMENWERK PLUS gesprochen. Hierbei handelt es sich um die neu entwickelte Erweiterung des im Folgenden noch genauer zu betrachtenden Übersetzer Rahmenwerks. Es wird in diesem Zusammenhang auch nur von einem RAHMENWERK oder im Kontext der Integrationsplattform von einem RAHMENWERK PLUGIN oder COMPILERCORE gesprochen. Für die Bezeichnung der Rahmenwerk-Erweiterung wird die Schrift in KAPITÄLCHEN formatiert.

Folgend wird wieder auf die ursprüngliche Form des Rahmenwerks eingegangen.

2.3 Analyse von Modellen und Synthese von COCoMa-Systemen

Die Grafik 2.2 zeigt den Ablauf eines Übersetzungsvorgangs in Form eines Aktivitätsdiagramms in UML-Notation[2]. Die linke Swimlane mit der orangenen Umrandung zeigt das Übersetzer-Rahmenwerk. Die Swimlanes rechts stellen Generatoren dar. Der Fokus dieser Betrachtung liegt auf der Synthesephase.

[2]Vergleiche dazu [3], [4]

Die Analysephase wird hier durch die erste Aktion dargestellt: Parse Modell. Das Asset-Modell wird in der Analysephase validiert und es wird ein flüchtiges Zwischenmodell daraus erstellt.

Die Synthesephase beginnt. Dazu wird als erstes die Konfigurationsdatei ausgelesen. Es handelt sich hier um eine Datei im XML-Format[3]. Darin enthalten sind Generator-Kennungen und Parameter für deren Aufruf. Die Generatoren werden gestartet. Als Eingabe für einen Generator werden die soeben genannten Parameter aus der Konfiguration, das Zwischenmodell und eventuell Symboltabellen, die andere Generatoren erstellt haben, benötigt. Ausgabe eines Generators sind eine flüchtige Symboltabelle und Quellkode. Diese Symboltabellen sind weiter aufbereitete Daten auf Grundlage des Zwischenmodells. Die Symboltabellen hier sind nicht mit jenen eines klassischen Kompilierers zu verwechseln, da sie hier weitaus mehr umfassen, als Schlüssel-Wert Zuweisungen. Es handelt sich bei den Symboltabellen um Java-Klassen, die komplexe Objekte bereitstellen können. Produkt des Übersetzers ist der Quellkode für ein System zur konzeptorientierten Inhaltsverwaltung. - In der Abb. 2.2 mit COCoMaS[4] bezeichnet.

2.4 Arbeitsprozess von der Modelldefinition zur Systemerstellung

Die Eingabe für den Modell-Übersetzer verlangt nach einem Asset-Modell. Das Modell muss in einem Editor erstellt werden. Bislang gibt es für die Asset-Modell Erstellung noch keine Unterstützung in Form eines angepassten Editors.

Für die Definition einer Konfigurationsdatei könnte ein XML-Editor Verwendung finden. Das würde zwar eine Übersicht über die Struktur des Dokumentes schaffen, eine Kopplung von Generator-spezifischen Eigenschaften und dem, was man schreiben will, gibt es jedoch nicht. Parameter-Bezeichner könnten ebenso falsch geschrieben werden wie die zu ladenden Generator-Klassen.

Über die Kommandozeile wird der Übersetzer in Form einer Stapelbearbeitung aufgerufen. Dazu sind viele Parameter notwendig. Nicht zu vergessen ist hierbei, dass, gemäß der Funktionsweise einer Java Virtual Machine (JVM), die Pfade der Generatoren zum Laden von Klassen mit eingebunden werden müssen.

Nach Ablauf der Übersetzung sind die entstandenen Quelldateien mit entsprechenden Werkzeugen weiter zu verarbeiten. Beispielsweise wird ein Java-Kompilierer verwendet, um aus dem generierten Quellkode lauffähigen Java-Bytekode zu erzeugen. Ein Datenbank-Management Werkzeug könnte aus Anweisungen einer Datenbank-Definitionssprache konkrete Tabellen erzeugen, etc.

Wenn alle Quellkode-Teile so umgesetzt wurden, dass das System lauffähig ist, wird dieses dann gestartet. Wie das geschieht hängt von der Art der erzeugten Systeme ab.

[3] Siehe dazu [26]
[4] Kurz für: COCoMa **S**ystem

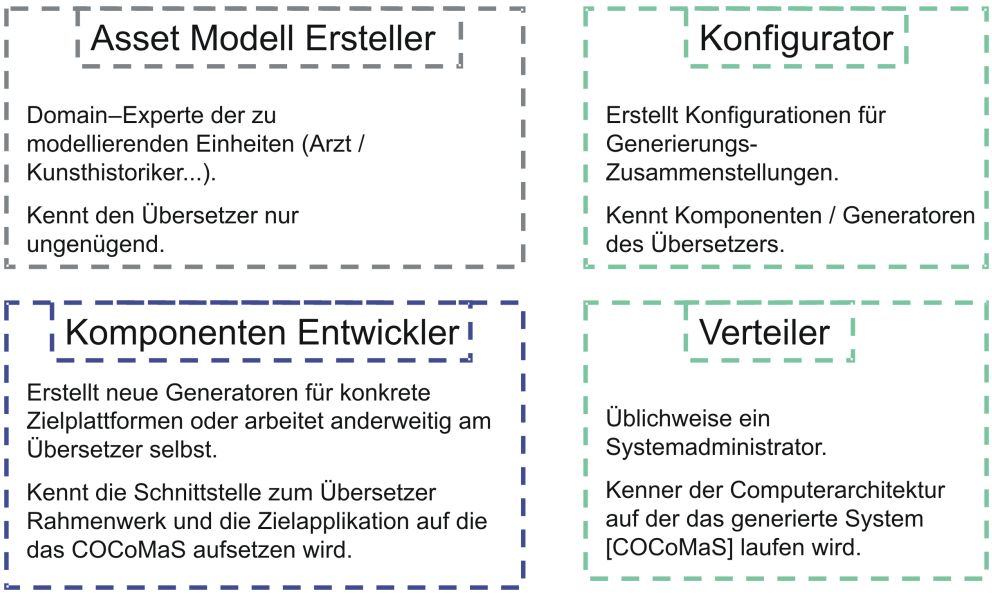


Abbildung 2.3: Rollen von Benutzern im Überblick

2.5 Rollen und Anwendungsfälle

Es sind nun im folgenden einige Rollen von Benutzern zu unterscheiden. Die Kompilierer Umgebung hat verschiedene Benutzergruppen zu unterstützen. In der Abb. 2.3 sind die vier Benutzer-Rollen übersichtlich aufgezeigt.

An dieser Stelle wird ein typisches Anwendungsscenario eingeführt, damit die verschiedenen Rollen am Beispiel deutlich werden und ein Eindruck von der Funktionsweise des zukünftigen Systems vermittelt.

2.5.1 Asset-Modell Ersteller

Der imaginäre Professor Wagner, Leiter einer medizinischen Abteilung zur Analyse von Krankheitsfällen hat sich mit dem aufsteigenden Unternehmen zur Erschaffung konzeptorientierter Inhaltsverwaltungssysteme COCoMa-Tec. in Verbindung gesetzt. Der Professor übernimmt als Domainexperte die Aufgabe der Erstellung von Assets. Dazu muss er die Asset-Definitionssprache erlernen und dann Assets definieren. Es sind sowohl medizinische Dokumente wie digitale Röntgenbilder, Labor- und Testberichte zu speichern, als auch die Umstände, unter denen Diagnosen gemacht wurden. Dies betrifft behandelnde Ärzte, Mitarbeiter verschiedener Institutionen, technischer Reifegrad der Analysegeräte und vieles mehr, was dem Professor noch gar nicht in den Sinn kommt. Einer seiner Mitarbeiter nimmt die gleiche Rolle ein und kooperiert mit dem Professor. Sie arbeiten beide an dem gleichen Modell.

Professor Wagner und sein Mitarbeiter nehmen die **Rolle: Asset-Modell Ersteller** ein.

Die Aufgaben der Asset-Modell Ersteller sind in der Grafik 2.4 aufgezeigt. Als erstes wird das Asset-Modell erstellt. Für den Fall einer Verfeinerung eines

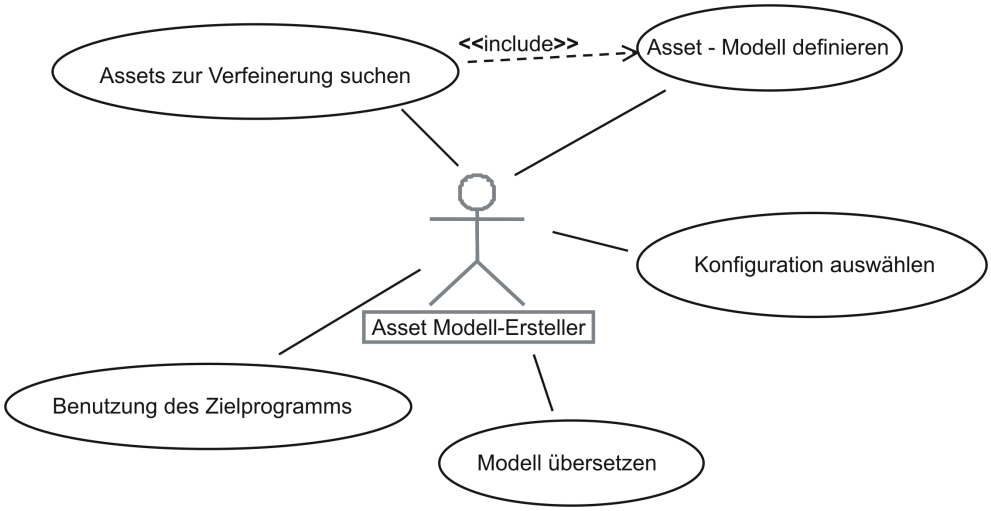


Abbildung 2.4: Anwendungsfalldiagramm des Asset-Modell Erstellers

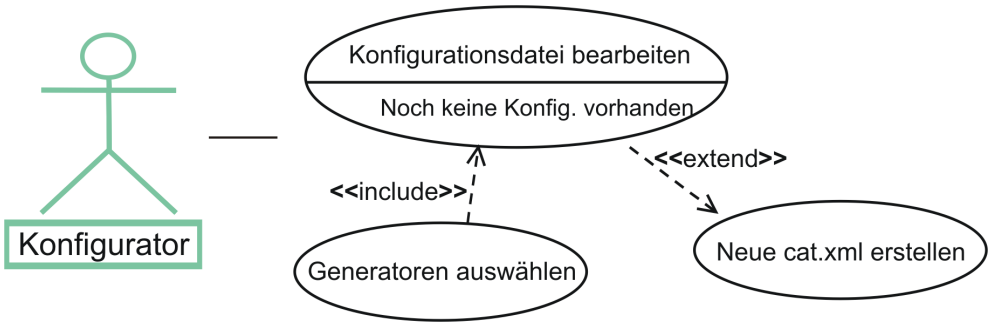


Abbildung 2.5: Anwendungsfalldiagramm des Generator-Konfigurators

bereits bestehenden Modells muss dieses zuerst gesucht und importiert werden. Dann wird mit den richtigen Einstellungen das Modell kompiliert. Von den Einstellungen der Generatoren hat weder der Professor noch der Mitarbeiter zu Beginn des Projektes Ahnung. Die Generatoren haben Quellkode generiert, dieser wird dann automatisch oder manuell weiter kompiliert. Dabei handelt es sich zum Beispiel um Java-Quellkode und dazu um einen Java-Übersetzer. Der Professor ist dann auch der Benutzer des entstandenen Programms und wird schon in die erste Version Daten eingeben, die dann auch in Folgeversionen weiter Verwendung finden.

2.5.2 Generator Konfigurator

Damit aus dem Asset-Modell auch die richtigen Programme werden, muss jemand der sich mit dem Kompilierer auskennt, die richtige Konfiguration vornehmen. Das wird die Firma CoCoMa-Tec. nicht von seinen Kunden erwarten. Ein Kundendienstmitarbeiter der Firma wird für Prof. Wagner ein Dokument

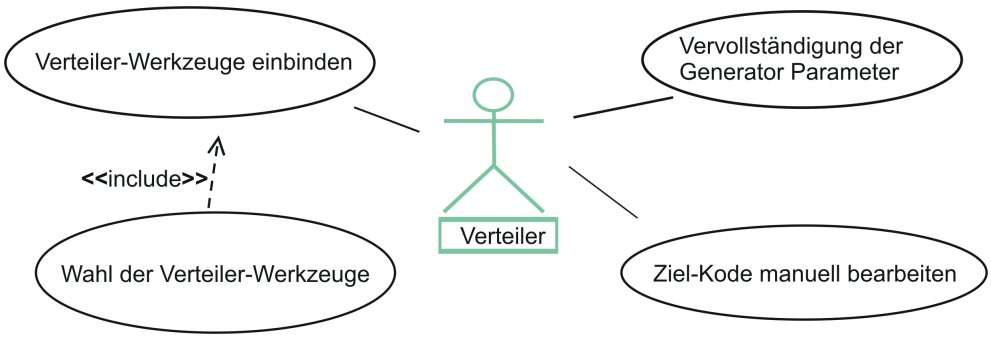


Abbildung 2.6: Anwendungsfalldiagramm des System-Administrators

zur Konfiguration der Generatoren erstellen[5]. In der Abb. 2.5 ist das noch einmal bildlich dargestellt.

Man stelle sich vor, dass einige Übersetzungen sehr viel Zeit benötigen. Vielleicht muss eine der Komponenten sogar Kontakt mit entfernten Ressourcen oder Programmen aufnehmen. Dann möchte der Benutzer bestimmt nicht, dass jedesmal vollständig kompiliert wird. Dabei muss man bedenken, dass eine Systemerzeugung in zukünftigen Umgebungen im Hintergrund geschehen könnte, vielleicht bei der Speicherung der Modell-Datei. Teilt man nun den gesamten Generierungsprozess in verschiedene Konfigurationen, kann man langwierige Arbeiten selektiv aktivieren. Dabei ist allerdings darauf zu achten, dass Generatoren manchmal die Vorarbeit anderer benötigen. Jede Konfiguration muss dahingehend vollständig sein. Es ergeben sich gerichtete azyklische Graphen von Abhängigkeiten.

Der Kundendienstmitarbeiter übernimmt die **Rolle: Generator Konfigurator**.

2.5.3 Verteiler

Aus dem gleichen Haus wie der Professor aus Abschnitt 2.5.1 kommt der Verteiler. Der Verteiler hat genaue Kenntnis über die Architektur des Computernetzes, dass später für die Inbetriebnahme des Inhaltsverwaltungssystems verwendet wird. Dazu gehören auch Datenbankanbindungen mit Passwörtern und Einstellungen, die den Ressourcen und der Verteilung des Computernetzes angepasst werden müssen. Üblicherweise kennt ein System-Administrator, oder bei einfachen Systemen auch der Kundendienstmitarbeiter aus Abschnitt 2.5.2, notwendige Parameter. Verteiler-Werkzeuge können ganz genau so wie Generatoren eingebunden werden, benötigen aber entsprechende Parameter, über die zumeist nur ein System-Administrator Kenntnis hat. Im Diagramm 2.6 kann man sehen, dass der Verteiler eine vorhandene Konfigurationsdatei ergänzt. Manchmal kann es auch sein, dass der Verteiler generierten Quellkode anpassen muss.

Die **Rolle: Verteiler** übernimmt ein System-Administrator.

[5]Im späteren Verlauf wird dieses Dokument als Konfigurationsdatei mit der Namenskonvention cat*.xml auftreten

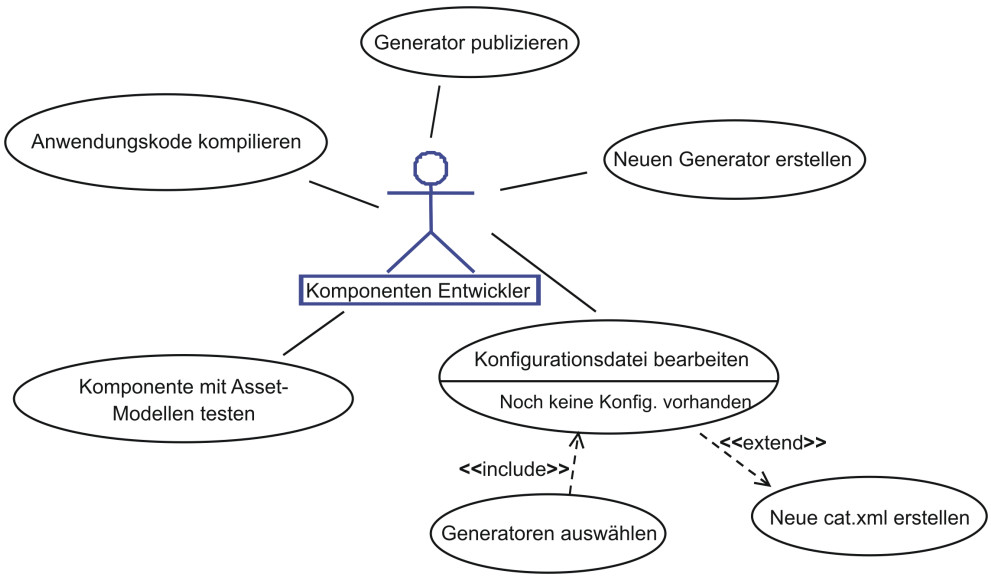


Abbildung 2.7: Anwendungsfälle des Übersetzer-Komponenten Entwicklers

2.5.4 Übersetzer-Komponenten Entwickler

Benutzer wie in den Abschnitten 2.5.1, 2.5.2 und 2.5.3 beschrieben gibt es noch nicht. Zum jetzigen Zeitpunkt[6] sind mehrere Studenten damit beschäftigt Generatoren für den Übersetzer zu konstruieren. Diese Studenten und auch der Leiter des Projektes nehmen die Rolle des Übersetzer-Komponenten Entwicklers ein.

Eine Komponente für das Kompilierer Rahmenwerk kann sowohl Bestandteil der Analyse- als auch der Synthesephase sein, siehe Abschnitt 2.3. Es ist denkbar, dass nach der Synthesephase noch Komponenten angesprochen werden, die eine Verteilung oder andersartige Verarbeitung vornehmen. Die Aufgaben eines Komponenten Entwicklers, vergleiche Abb. 2.7, bleiben auch nach Gründung von COCoMa-Tec. bestehen. Stellvertretend für alle Arten von Komponenten wird in der Abb. von Generatoren gesprochen. Der Komponenten Entwickler muss genaue Kenntnis der Kompilierer Umgebung haben und seine Testprojekte selbstständig einrichten und übersetzen.

Mitarbeiter von COCoMa-Tec nehmen die **Rolle: Kompilierer-Komponenten Entwickler** ein.

2.6 Resultierende Anforderungen

Aus den Rollen und Anwendungsfällen des Abschnittes 2.5 resultieren Wünsche und Anforderungen von Benutzern, die über die Funktionalität des Übersetzers hinausgehen. Dazu werden die beiden wichtigsten Rollen betrachtet. Zum einen der Endkunde, der Modell-Ersteller (Abschnitt 2.5.1) und zum an-

[6] Herbst 2004

deren der Übersetzer-Komponenten Entwickler (Abschnitt 2.5.4), der in der Entstehungsphase des Übersetzers selbstredend Bedeutung findet.

2.6.1 Anforderungen des Asset-Modell Erstellers

Bei der Erstellung eines Modells war vor den Ergebnissen dieser Arbeit keinerlei Hilfe vorhanden. Der Benutzer muss sich einen Überblick über die Modell-Spezifikation verschaffen und diese Dokumentationen selbstständig suchen. Das ist zumeist der erste Schritt eines neuen Benutzers.

→ **Übersichtliche Darstellung von Dokumentationen.** Anzeige von helfenden Dokumenten findet in einer Übersicht und an genau *einem* Ort statt.

Sobald der Benutzer weiß wie man ein Modell erstellt, kann es an die Definition desselben gehen. Dazu wird ein Editor benutzt. Die Wahl des Editors steht dem Benutzer frei. In dem Prozess der Modell-Definition wird der Benutzer bislang von keinem Editor unterstützt. Tippfehler werden erst nach der Übersetzungsphase deutlich.

→ **Editoren, die die Modell-Definition erleichtern.** Das kann das Hervorheben von Schlüsselwörtern, oder auch eine automatische Vervollständigung von Modell-Kode sein.

→ **Anzeige von Fehlern im Editor.** Lexikalische sowie syntaktische Fehler der Modell-Definition sollen schon im Editor angezeigt und vom Benutzer korrigiert werden. Das spart unnötige Übersetzer-Läufe.

→ **Grafische Editoren.** Die textbasierte Modell-Definition soll durch grafische Elemente ergänzt werden. Dadurch gewinnt die Definition an Übersicht. Außerdem kann die Erstellung auch dadurch beschleunigt werden, da grafische Elemente oft eine Art Makro-Funktion haben. Sie können mehrere textbasierte Elemente zusammenfassen.

Um den Übersetzer starten zu können, muss der Benutzer eine vom **Generator-Konfigurator** erstellte Konfiguration auswählen. Dies geschieht über den Eintrag des Bezeichners im Übersetzer-Aufruf. Der Bezeichner kann aus der Konfigurationsdatei oder aus Dokumentationen heraus gesucht werden.

→ **Einfache Konfigurationsauswahl.** Eine grafische Oberfläche die eine Übersicht der Konfigurationen zeigt und die Auswahl ermöglicht, beschleunigt den Startvorgang des Übersetzens.

Soll die Übersetzung mit einem Modell und einer Konfiguration gestartet werden, ist sicherzustellen, dass von der Laufzeitumgebung alle Komponenten, hauptsächlich die Generatoren, gefunden werden. Der Benutzer muss für eine Übersetzung das Programm zur Modell-Definition verlassen. Das kostet Zeit und zwingt den Benutzer, sich in Gedanken mit der technischen Seite des Übersetzungsvorgangs zu beschäftigen. Auf der Modell-Definition sollte jedoch der Fokus liegen.

→ **Installationsunterstützung der Komponenten.** Ein einheitlicher Mechanismus zur Erweiterung des Laufzeitsystems um zusätzliche Komponenten. Dabei wird eine Übersicht gegeben, welche Versionen der Komponenten verfügbar und installiert sind.

→ **Übersetzung auf Knopfdruck oder gar automatisch im Hintergrund.** Der Ersteller eines Modells kann sich voll seiner Aufgabe widmen. Einmal gemachte Einstellungen werden zum Lauf des Übersetzers genommen und es werden keine weiteren Eingaben des Benutzers für jeden neuen Übersetzungsvorgang erwartet.

Sollte die Systemerzeugung viel Zeit in Anspruch nehmen oder durch technische Hürden verlangsamt werden oder gar stehenbleiben, hat der Benutzer im Fall des Kommandozeilen gesteuerten Übersetzers keine Möglichkeit, sich über den aktuellen Status des Vorganges zu informieren. Der Benutzer bekommt weder einen Hinweis darauf welche der Komponenten den Fehler aufweist, noch ist er in der Lage, den Prozess kontrolliert zu stoppen.

→ **Überblick über den Prozessfortschritt.** Eine Anzeige soll dem Anwender verdeutlichen, welcher Generator gerade läuft und wie weit dieser fortgeschritten ist.

→ **Abbrechen der laufenden Systemerzeugung.** Eine Unterbrechung kann in der Form stattfinden, dass in einem früherem Lauf generierte Teile erhalten bleiben. So fällt das Aussetzen eines „Service“ möglicherweise gar nicht ins Gewicht.

Um das generierte System laufen zu lassen, sind noch Maßnahmen zur Weiterverarbeitung und der Verteilung notwendig. Diese Aufgaben können sehr komplex sein. Dann gilt es, das System zu starten, zu benutzen und zu testen. Je nach Generatorwahl kann die Weiterverarbeitung unterschiedliche Werkzeuge verlangen. Auch mit diesen hat der Benutzer sich dann zu beschäftigen. Dadurch entfernt er sich zeitlich und gedanklich immer mehr von der Modell-Erstellung. Bei dem Test des fertigen Systems geht es oft gerade darum, die soeben gemachten Änderungen im System wiederzufinden und sich von deren korrekter Funktion zu überzeugen.

→ **Voreinstellungen und Ablauf der Weiterverarbeitung.** Einmal vom **Verteiler** eingestellte Werte sorgen für die automatische Weiterverarbeitung des System-Quellkodes und dann für die Verteilung.

→ **Einfaches Starten des fertigen Systems.**

2.6.2 Anforderungen des Übersetzer-Komponenten Entwicklers

Die Person, die die Rolle des Komponenten-Entwicklers einnimmt, muss de facto auch alle anderen Rollen ausfüllen. Ein neuer Generator verlangt nach neuen Konfigurationen, von denen noch niemand anderes eine Ahnung haben

kann, bevor der Generator geschaffen ist. Es erfordert neue Arten der Weiterverarbeitung des erzeugten Quellkodes und dessen Verteilung. Die Aufgaben eines Modell-Erstellers müssen zu Testzwecken auch übernommen werden.

→ **Einheitliche Entwicklungs-, Test- und Ablaufumgebung.** Ein neuer Generator muss in einer Programmierumgebung erzeugt werden. Dann muss dieser Generator sich in einem Testszenario beweisen. Er wird in einen Übersetzungsvorgang eines Testmodells eingebunden. Das dadurch erzeugte System muss dann gestartet werden.

Der Generator-Ersteller[7] möchte verschiedene Einstellungen für seinen Generator ermöglichen. Diese Einstellungen sollen möglicherweise von dem Modell-Entwickler gemacht werden. Es handelt sich hier also um Einstellungen, die über *diejenigen* des **Konfigurators** hinaus gehen.

→ **Persistenzmechanismus von Einstellungen mit grafischer Oberfläche.** Der Generator-Entwickler implementiert nicht für jeden Generator die Speicherung und die Wiederherstellung von Konfigurations-Daten. Vielmehr entscheidet sich der Generator-Bauer nur, welcher Art die Werte sind (Dateiname, Verzeichnis, Zeichenkette, Zahl. . .). Die Präsentation der Einstellungen fordert keinen Aufwand vom Generator-Entwickler.

Auch ein Komponenten-Entwickler muss durch Dokumentationen angeleitet werden. Da die Architektur, für die ein Generator Kode erstellt, beliebig ist, muss der Entwickler sich an entsprechender Stelle informieren. Des weiteren muss der Entwickler die Spezifikationen des Übersetzer-Rahmenwerks beachten. Um einen Generator zu integrieren ist auch einiges nachzulesen. Der Entwickler schreibt kein Programm, das für sich alleine lauffähig ist, sondern eine Komponente für ein Rahmenwerk.

→ **Einbindung von Dokumentationen.** Das Rahmenwerk betreffende Dokumentationen sind übersichtlich zusammen mit jenen für den **Modell-Definierer** erreichbar.

→ **Einfache Integration eines Generator-Prototyps.** Ein lauffähiger Generator muss, auch von Einsteigern, leicht erstellt werden können.

Neben der Produktion von Quellkode kann es notwendig sein, dass der Generator mit Benutzern interagiert. Das kann zweierlei Zweck verfolgen. Für die Entwicklungsphase vereinfachen Ausgaben der Komponente die Fehleranalyse. Im laufenden Betrieb wird bei längeren Aktionen der Benutzer über den Prozessfortschritt informiert oder es werden auch automatische Aktionen der Komponente dem Benutzer erklärt.

→ **Kommunikation über Ansichten und Dialoge.** Dialoge ermöglichen, zusätzlich zu dem oben erwähnten, von dem Benutzer Eingaben in Ausnahmefällen.

[7] Als Beispiel eines Komponenten-Entwicklers.

Für die Entwicklung in der Gruppe ist es wichtig, Quellkode zu synchronisieren und andere Daten auszutauschen.

→ **Unterstützung für Entwicklung in der Gruppe.** Für die Synchronisation von Quellkode sind viele Systeme verfügbar. Als Beispiele seien hier *Perforce*[8] oder *CVS*[9] genannt.

Letztendlich muss eine *fertige* Komponente publiziert werden. Auch Nachfolgeversionen müssen zu dem Kunden gelangen. Wichtig ist dabei, dass der Kunde eine Übersicht über die aktuelle Version der Komponente bekommen kann. Von dem Benutzer wird nicht erwartet, dass er sich mit beliebig vielen Installationsmechanismen auskennt.

→ **Publikation durch Updatemechanismus mit Versionierung.** Alle Komponenten werden auf die gleiche Art eingebunden. Dabei denke man auch an eine einfache oder automatisierte Aktualisierungsfunktion.

2.6.3 Feststellung des Bedarfs an einer Erweiterung

Aus den Anforderungen der vorangehenden Abschnitte wird deutlich, dass ein Bedarf an einer Erweiterung der Übersetzer-Rahmenwerks vorliegt. Man kann nicht sämtlich Forderungen mit eigens entwickelter Software realisieren, dazu wäre der zeitliche Aufwand zu groß. Zusammenfassend kann man sagen, dass sich die Anforderungen auf folgende grobkörnige Komponenten beschränken.

- Ansteuerung der verschiedenen Prozessschritte, abhängig von der Benutzerrolle
- Integration aller Komponenten in eine einheitliche Umgebung
- Erweiterung um grafische Elemente zur Interaktion zwischen Komponenten und Benutzern
- Schnittstellen zur späteren Erweiterung der Umgebung
- Unterstützung der Programmierarbeit eines Komponenten-Entwicklers

Dabei soll die Eigenentwicklung auf ein Minimum beschränkt werden.

2.7 Evaluation möglicher Technologien

Um zu einer Entscheidung bezüglich der Technologie zu kommen, mittels derer die Kompilierer Umgebung implementiert wird, müssen zuerst die Rahmenbedingungen festgesteckt werden.

Eine der Vorbedingungen ist es, bestehende Komponenten, wie Parser oder Generatoren, weiterhin verwenden zu können. Ein von der Kommandozeile aus gesteuerter Kompilierer ist zur Zeit der Aufgabenstellung schon vorhanden.

[8] Kommerzielles Produkt
[9] **C**oncurrent **V**ersions **S**ystem, open source

Ebenso einige Komponenten, wie z.B. ein Generator für Schnittstellen, passend zum Asset-Modell. In Absprache sollen folglich auch kommende Programme in Java geschrieben werden.

Betrachtet man insbesondere die Anforderungen des Komponenten-Entwicklers aus Abschnitt 2.6.2, so können serviceorientierte [10] oder andere auf entfernte Kommunikation angewiesene Architekturen nicht in Betracht fallen. Die Reaktionszeit ist besonders in den Arbeitsprozessen eines Entwicklers bedeutend.

2.7.1 Apache Ant als Vertreter von Skriptinterpretern

Als erst einmal einfachste Lösung könnte eine Skriptsprache angesehen werden, die die Ablaufsteuerung übernehmen kann. Die Möglichkeiten von *Apache Ant*[11] sind beispielsweise sehr vielfältig. Es bestehen auch schon viele Skripte; Ant-Tasks, die nützlich sein könnten. Ant ist weitestgehend unabhängig von *Plattformen*[12]. Das ist Vor- und Nachteil zugleich. Eine Abhängigkeit zu einer Plattform kann insofern Probleme bereiten, als dass man auf die Unterstützung und Fehlerbeseitigung für die Plattform angewiesen ist. Allerdings fehlen dann auch die in einer Plattform enthaltenen Funktionen. Darauf wird im Folgenden noch eingegangen.

Da es keinen festen Satz von Generatoren gibt, sondern fast beliebige Zusammenstellungen dieser, kann das Skriptkonstrukt unüberschaubar werden. Andere Entwickler müssten dann auch Skripte erstellen.

2.7.2 Die Plattformen NetBeans, eclipse und IntelliJ

Es werden drei Plattformen vorgestellt, die eine grafische Erweiterung zulassen. Außerdem handelt es sich bei NetBeans, eclipse und IntelliJ IDEA um **I**ntegrated **D**evelopment **E**nvironments (IDEs).

Die von *Sun Microsystem* übernommene und geförderte Entwicklungsumgebung NetBeans ist eine IDE und Plattform mit hohem Bekanntheitsgrad. Es basieren viele von Suns Programmen auf der NetBeans Plattform, *Sun One Studio* kann hier genannt werden. Auch die Liste anderer Softwarehersteller, die NetBeans als Plattform nutzen, ist lang. In der Abb. A.1 im Anhang ist eine Beispielansicht von NetBeans zu sehen. NetBeans ist frei verfügbar und erweiterbar. Es stellt sich auf der Internet-Seite [25] vor.

[10] Bekannt als SOA = **S**ervice **O**riented **A**rchitectures.
[11] Siehe: [23]
[12] Auf die Besonderheiten von Plattformen und IDEs wird im Abschnitt 2.7.3 eingegangen.

Wie auch die Namensgebung von eclipse schon vermuten lässt[13], stehen eclipse und NetBeans in Konkurrenz. Eclipse ist nach NetBeans entstanden und wird von IBM und OTI gefördert. Seit 2001 wird eclipse von einer wachsenden Gemeinschaft großer Firmen der Software-Branche entwickelt, eine Auswahl der bekanntesten ist hier zu sehen.

Es ist erlaubt, auf der Basis von eclipse eigene Distributionen zu erstellen und zu vermarkten. Laut [12] wurden über 100 Produkte, eclipse als Plattform nutzend, erstellt. Nähere Informationen zu eclipse findet man in [7], [5], [10], [6] und [8]. Im Anhang ist eine Beispielansicht A.3 von eclipse zu sehen.

Bei IntelliJ handelt es sich um eine ebenfalls um eine Plattform. In dieser Arbeit wird eine kommerzielle Distribution betrachtet: IntelliJ IDEA von der Firma JetBRAINS. Da die Ansprüche der Entwickler[14] hoch sind, aufgrund des frei verfügbaren Angebotes an IDEs, reicht es nicht aus, IntelliJ losgelöst von kommerziellen Produkten zu betrachten. Eines dieser Distributionen heißt IntelliJ IDEA und kostet für einen Benutzer knapp 500 Euro. In [24] finden sich nähere Informationen zu IntelliJ IDEA. Die Beispielansicht A.2 im Anhang vermittelt einen Eindruck von der Präsentation von IntelliJ.

2.7.3 Geforderte Funktionalität der Einbettungs-Umgebung

Aus der Analyse ergibt sich: Zweierlei Funktionalität muss die neue Laufzeitumgebung von Haus aus mitbringen.

- **Vorteile einer IDE** - Eine Entwicklungsumgebung kann viele Funktionen, die ein Programmierer nicht missen möchte, bieten.
  - Programmierung wird durch Fehleranzeige im Quellkode erleichtert.
  - Intelligente Analyse des Kodes zur Programmierzeit
  - Umfangreiche Navigation im Quellkode
  - Verschiedene Ansichten des Kodes
  - Automatische Vervollständigung
  - Generierung von Quellkode auch in größerem Maßstab
  - Refactoring, automatische Änderungen von Referenzen bei Umstrukturierungen
  - Wiederherstellung alter Versionen des Quellkodes

[13] eclipse = Sonnenfinsternis - Sun = Sonne
[14] Hier: Komponenten-Programmierer

  - Unterstützung für Teamarbeit
  - Test Rahmenwerk, zum Beispiel JUnit

- **Erweiterbarkeit einer Plattform** - Man kann das Einbinden der Kompilierung im Hintergrund oder auf Knopfdruck realisieren und auf die Arbeitsumgebung und Ressourcenverwaltung der Plattform zugreifen. Eine Plattform sollte Erweiterungspunkte für die grafische Oberfläche bereitstellen. Dabei können oft Makropakete als Abstraktion von grafischen Elementen benutzt werden. Es sind erweiterbare Ansichten, Menüs, Symbolleisten, Einstellungsdialoge und vieles mehr vorhanden. Ohne die Erweiterbarkeit würde die Umgebung nicht für eine Integration der Übersetzer-Umgebung genügen. Ein wichtiges Kriterium für die Erweiterbarkeit ist Qualität, Quantität und Verfügbarkeit von Dokumentationen.

Bezüglich dieser Funktionen werden die vorgestellten Programme im folgenden Abschnitt verglichen. Es wird herausgestellt inwiefern sie den soeben aufgelisteten Anforderungen genügen.

2.7.4 Gegenüberstellung: IntelliJ IDEA – NetBeans – eclipse

Die Menge der vorhandenen Funktionen von den IDEs NetBeans, IntelliJ IDEA und eclipse unterscheiden sich kaum. Die aus dem vorangegangenen Abschnitt bekannten Funktionen einer IDE werden sämtlich von allen drei Programmen angeboten. Es werden nun Unterschiede zwischen den Programmen aufgezeigt.

1. NetBeans und IntelliJ benutzen AWT und Swing[15] für die Visualisierung, siehe Abb. A.1 und A.2 im Anhang. Eclipse hat eine eigene Oberfläche, die die Bildschirmelemente der jeweiligen Betriebssysteme direkt anspricht, siehe Ansicht A.3 im Anhang. Das eclipse Graphical User Interface (GUI) hat den Ruf, schneller, stabiler und leichter erlernbar zu sein. Außerdem muss sich der Benutzer nicht an eine neue Oberfläche gewöhnen, da die des Betriebssystems genutzt wird. Damit einher geht der Nachteil, dass sich eclipse auf verschiedenen Betriebssystemen unterschiedlich darstellt. Es kann zwar meistens davon ausgegangen werden, dass eine Person eclipse nur auf einem Betriebssystem benutzt, haben sich allerdings mehrere Personen mit einer Installation von eclipse zu befassen[16], entstehen Schwierigkeiten durch die Darstellung.

2. Alle drei Programme unterstützen Java 1.5, die eclipse-Version 3.0 hat allerdings noch Probleme in der Auto-Vervollständigung und in der Kode-Generierung von 1.5-spezifischem Kode.

3. Ein grafischer GUI-Designer für zu schreibende Anwendungen ist in IntelliJ IDEA integriert. NetBeans und eclipse können jedoch auch entsprechend erweitert werden.

[15] In Java implementierte Benutzeroberfläche
[16] Man denke z.B. an eine Präsentation.

4. Eclipse und IntelliJ IDEA bieten eine Update-Unterstützung, so können Komponenten versionsgesteuert ergänzt werden.

5. Integration der Testumgebung JUnit in eclipse und IntelliJ IDEA, das zuvor vorgestellte Ant wird ebenfalls von beiden integriert.

6. IntelliJ IDEA bietet eine integrierte Enterprise Java Beans-Generierung.

7. IntelliJ IDEA bietet keinen kostenlosen email-support.

8. IntelliJ IDEA bietet eine Integration von zwei Java Übersetzern: Javac und Jikes. NetBeans und eclipse beschränken sich auf Javac.

9. Eclipse bietet Debug-Unterstützung für die Plugin-Erstellung. Es können neue Plugins leicht integriert und zur Laufzeit kann deren Kode verändert werden. Solche Veränderungen wirken sich sofort aus und machen eine erneute Integration unnötig.

10. Eclipse hat mit dem Eclipse Modeling Framework eine Model Driven Architecture (MDA) - Komponente, die als Vorbild für den „Asset-Modell →Konzeptorientiertes Inhaltsverwaltungssystem - Kompilierer“ dient.

11. Die Fülle an Dokumentationen zu eclipse ist sagenhaft. Die Gemeinschaft der Entwickler in eclipse ist sehr produktiv in der Erstellung von Einführungen, Anleitungen, Dokumentationen und hat Newsgroups mit sehr hoher Beteiligung.

12. Eclipse führt eine lokale Historie der Dateien seiner Arbeitsumgebung. Dateien können wieder hergestellt werden, selbst wenn sie gelöscht wurden.

13. Das Builder Konzept von eclipse wird in Abschnitt 3.5 beschrieben.

14. NetBeans bietet Unterstützung von Java 2 Micro Edition für Programme für *Mobile* Geräte.

15. Durch die Integration von *JFluid* in NetBeans ist es möglich die Leistungsfähigkeit und den Ressourcenverbrauch von Programmen zu analysieren.

Die Gewichtung der oben genannten Punkte anhand der Anforderungen wird im folgenden Abschnitt vorgenommen.

2.7.5 Das Werkzeug der Wahl: eclipse

Es soll eine Entscheidung bezüglich der Technologie, auf die eine Erweiterung des Rahmenwerks aufbaut, getroffen werden. In den vorangehenden Abschnitten wurde deutlich, dass nur eine Technologie den Ansprüchen entspricht, die sowohl eine erweiterbare Plattform, als auch eine ausgereifte IDE bietet.

IntelliJ IDEA ist als Vertreter kommerzieller Produkte vorgestellt worden. Da ein kommerzielle Produkt unter Einschließung der IDE erstellt werden soll,

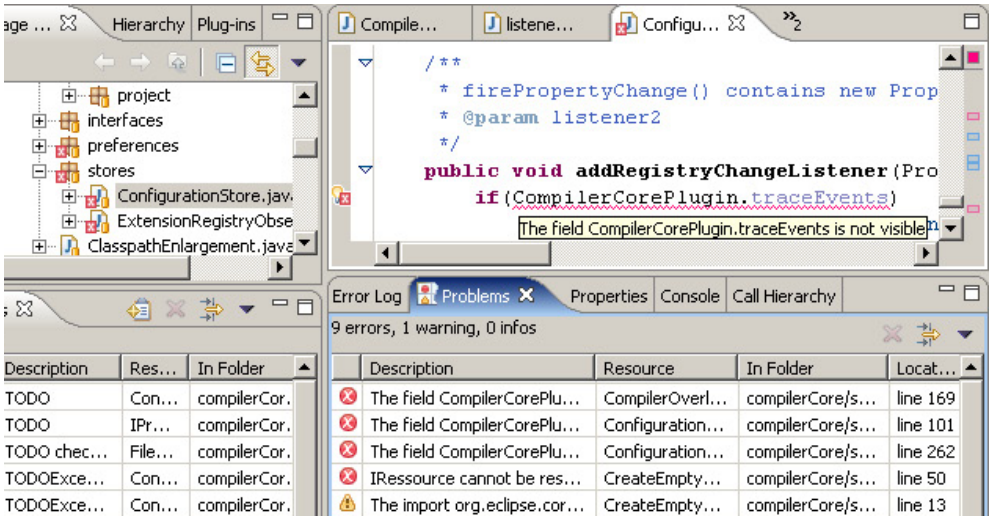


Abbildung 2.8: Marker zeigen Fehler im Asset-Quellkode

man vergleiche hierzu das Szenario aus dem Abschnitt 2.5.4, kommen solche Produkte nicht in Frage.

Für die Auswahl der Technologie wird nun auf die in dem vorangegangenen Abschnitt definierten Punkte eingegangen. Aufgrund der Update-Funktion (Punkt 4), der Integration der Testumgebung JUnit (Punkt 5), der Plugin-Entwicklungs-Unterstützung (Punkt 9), der Dokumentations-Vielfalt (Punkt 11) und wegen des Builder-Konzepts (Punkt 13) wird eclipse als Plattform gewählt. Die in eclipse **nicht** vorhandenen Funktionen (Punkte 6, 8, 14 u. 15) werden nicht für die vorliegende Aufgabe benötigt.

2.7.6 Verwendbare Funktionalität eclipses

Eclipse als Plattform bietet eine Menge an Funktionalität die einfach mitbenutzt werden kann. Dazu wird eclipse an entsprechenden Stellen, wie in Abschnitt 3.4.1 beschrieben, erweitert.

- *Workbench bereichern* - Man kann die vorhandene Ansicht, den Workbench, an vielen Punkten bereichern. Menüs, Fenster, Views[17] können mitbenutzt oder auch vollständig neu erstellt und eingebunden werden.
- *Editoren hinzufügen* - Wie der Java-Editor oder der grafisch gestaltete Manifesteditor des PDE[18] können individuell für bestimmte Dateien Editoren definiert werden.
- *Workspace mitbenutzen* - Das Dateisystem wird von eclipse mit Metadaten verfeinert, und somit kann auf eine Historie ebenso zugegriffen werden wie auf eine Unterstützung für verteilte Entwicklung im Team.

[17] Fenster, die beliebigen Inhalt darstellen können.
[18] Eine Ergänzung zu eclipse: **P**lugin **D**evelopment **E**nvironment

- *Von Einstellungensdialogen profitieren* - Einstellungen können Ressourcebezogen und global genutzt werden. Neue Einstellungsmöglichkeiten werden in die vorhandene Struktur integriert.
- *Wizards, Cheat Sheets, Hilfe* - Der Benutzer kann durch verschiedene Mechanismen durch Erzeugungsprozesse geführt werden.
- *Integrierte Kompilierung* - Für den Vorgang der Übersetzung von Quelldateien beliebiger Sprachen ist eine Ablaufunterstützung vorgesehen.
- *Fehlerbehandlung vereinheitlicht* - Wie in Abb. 2.8 zu sehen ist, werden Fehler in Quelldateien schnell übersichtlich dargestellt und sind aus der Übersicht heraus mit einem 'Klick' erreichbar.
- *Anpassung der Arbeitsumgebung* - Die Auswahl der Views und Editoren für eine Ansicht ist individuell. Es können auch neue Perspektiven definiert werden.
- *Updatemechanismus mit Versionierung* - Alle in eclipse eingebundenen Teile können selektiv verwaltet und erneuert werden; eclipse bietet dazu die notwendige Übersicht wie auch Wizards zur leichten Ausführung von Installationen.
- *Vereinigung von Arbeits-, Test-, und Ablaufumgebung* - Generator-Programmierer können bei der Erstellung des Generators eclipses Java-Unterstützung nutzen. Ist ein Generator soweit fertig, dass er in das RAHMENWERK eingebunden werden kann, kann man ihn in eclipse als Plugin testen. Vom Generator erzeugte Systeme können können ebenfalls in eclipse gestartet werden. Wie so etwas in eclipse praktikabel funktioniert, kann im Abschnitt 3.9 nachgelesen werden.

KAPITEL 3

Design

Dieses Kapitel führt eine Erweiterung des zuvor vorgestellten Übersetzer Rahmenwerks ein. Diese Erweiterung wird konzeptionell erläutert und Komponenten werden vorgestellt. Daraufhin wird auf Grundlage der erarbeiteten Erweiterung und den Ergebnissen der Analyse die Plattform *eclipse* vorgestellt. Da es sich um eine Integration in eine schon vorhandene Architektur handelt, muss diese an vielen Stellen erläutert werden. Die Art und Weise der Integration ist dann zumeist offensichtlich. Nach einer Einführung in die grobe Struktur eclipses und Einführung von Begriffen werden die folgenden Konzepte, wie sie in der Plattform realisiert sind, erläutert werden.

- **Builder** (Abschnitt 3.5.2) Builder werden benutzt, um Verarbeitungslogik intelligent steuern zu können.
- **Laden von Plugins und Klassen** (Abschnitt 3.6) Das Laden von Klassen und Bibliotheken innerhalb des Bundle-Mechanismus wird erläutert. Wie Erweiterungen und dynamisches Laden funktionieren wird nachgegangen.
- **Rahmenwerk zur Erstellung von Konfigurationen** (Abschnitt 3.7) Es wird ein Rahmenwerk erstellt, das sowohl vorhandene Mechanismen eclipses nutzt, als auch eine Möglichkeit schafft, Komponenten dritter eine Einbindung zu erlauben. Dabei wird die Verwaltung persistenter Werte nicht von den eingebundenen Komponenten übernommen.
- **Verfolgung der internen Vorgehensweisen** (Abschnitt 3.8) Es werden Optionen zur Verfolgung der internen Zustände und Ereignisse erläutert. Komponenten-Entwickler nach Abschnitt 2.5.4 können dann nachvollziehen was innerhalb des erweiterten Rahmenwerks geschieht.
- **Benutzerführung** (Abschnitt 5) Der Benutzerführung ist ein eigenes Kapitel gewidmet und ist somit nicht Teil des hier vorgestellten Designs.

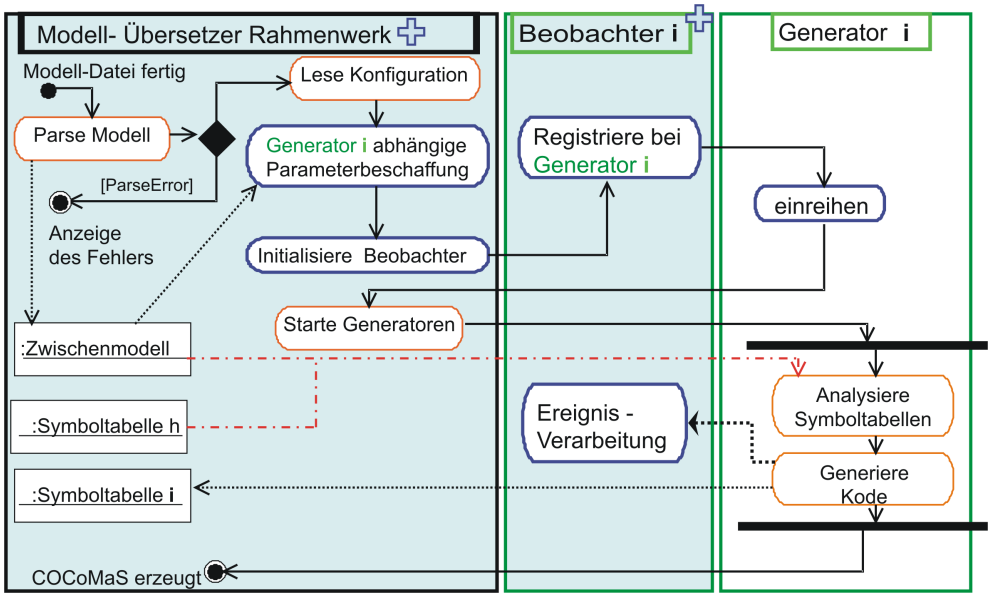


Abbildung 3.1: Vorgang der Systemerzeugung im Rahmenwerk Plus

Prinzipiell kann man davon ausgehen, dass die Umsetzung der Konzepte, also die Architektur, auf dem noch vorzustellenden Mechanismus der *Erweiterungen* (siehe Abschnitt 3.4.1) beruht. Details und Aufbau der Komponenten werden dann im nachfolgenden Kapitel *Implementation* betrachtet, da zur Erklärung oft auf die Ebene von Klassen und Schnittstellen übergegangen werden muss.

Werden Bezeichner in `Courier-Schrift` dargestellt, handelt es sich im Design um Dateien oder Ordner. An manchen Stellen ist es auch notwendig, Klassen und Bezeichner zu erwähnen. Es wird dafür die selbe Schriftart gewählt.

3.1 Ablauf der Erweiterung: Rahmenwerk Plus

In der Analyse wird die Funktionsweise des Rahmenwerks zur Ansteuerung von Generatoren erörtert. Dazu wird dort ein Aktivitätsdiagramm eingeführt (vgl. Abb. 2.2). Auf diesem Ablauf aufbauend zeigt Abb. 3.1, was sich für die Erweiterung des Rahmenwerks ändern wird. Das Rahmenwerk in seiner erweiterten Form wird RAHMENWERK PLUS genannt und wird in Grafiken durch ein blaues Plus-Zeichen dargestellt.

In der Abb. 3.1 sind drei Komponenten zu sehen.

Das neue Rahmenwerk ist blau hinterlegt.

Der Beobachter ist Teil des neuen RAHMENWERKS. Zudem ist er an einen bestimmten Generator gebunden. Die grüne Umrandung, in der gleichen Farbe wie der Generator daneben, deutet den Bezug zu einem bestimmten Generator an.

Der Generator ganz rechts ist grün umrandet. Er ist einer der Generatoren, die ablaufen. Es können auch noch andere Generatoren an Übersetzungsvorgän-

gen beteiligt sein. Ein Generator wird hier stellvertretend für alle gezeigt. Man beachte, dass die parallele Ausführung im Diagramm nicht auf den Bereich des *Generators i* beschränkt ist.

Die orange umrandeten Aktionen haben sich gegenüber dem originalen Rahmenwerk nicht verändert. Neuerungen werden mit einem blauen Rahmen gekennzeichnet.

Wie aus dem ursprünglichen Ablauf bekannt ist, wird das Modell zuerst einer Analyse unterworfen. Stellt sich heraus, dass das Modell fehlerhaft ist, so wird die Systemerzeugung abgebrochen und der Benutzer entsprechend informiert. Andernfalls wird, wie auch ursprünglich vorgesehen, die Konfigurationsdatei eingelesen und somit festgestellt, welche Generatoren benötigt werden. Neu ist nun, dass die Beschaffung der Parameter abhängig vom Generator geschieht. Auf die PARAMETERBESCHAFFUNG wird in folgenden Abschnitten eingegangen. Sodann wird ein BEOBACHTER initialisiert. Dieser Beobachter kann eine von dem Generator-Entwickler definierte Komponente sein, oder, falls nicht vorhanden, eine des RAHMENWERKS. Dieser Beobachter registriert sich bei dem Generator, für den er zuständig ist. Der Generator reiht den Beobachter in eine Liste von Ereignisempfängern ein. Wichtig zu bemerken ist hierbei, dass der Generator in seinem Aufbau *nicht* in der Weise verändert wird, dass er ohne die neue Ablaufumgebung nicht ablauffähig wäre. Der Generator kann wie im folgenden zu sehen ist, Ereignisse senden und somit mit der Umgebung, in die er eingebettet ist, kommunizieren.

Da ein `Generator` nicht extra für eclipse geschrieben wird, sondern auch in Rahmenwerken lauffähig sein soll, die nicht in eclipse eingebunden sind, ist der Beobachter der einzige Berührungspunkt des `Generator` zu eclipse. Wie im Entwurfsmuster Beobachter (siehe [2]) beschrieben, werden die Beobachter dynamisch angesprochen und erlauben eine lose Kopplung der Komponenten.

Das Rahmenwerk ist austauschbar geblieben. Der Generator kann ebenso im original Übersetzer-Rahmenwerk ablaufen.

3.2 Parameterbeschaffung Plus

Die im Abschnitt 3.1 beschriebene Auflösung von Parametern soll nun erläutert werden. Dafür betrachte man das Schaubild 3.2.

Zentrale Komponente ist der **Parameter-Lieferant**. Das RAHMENWERK sieht vor, einen Lieferanten zu stellen für den Fall, dass der Generator keinen eigenen definiert. Auch hier ist das an dem blauen Kreuz oben rechts in der Ecke des Lieferanten wie auch an dem grünen Rahmen zu sehen, der die Verbindung zu dem bestimmten Generator ' **i** ' anzeigt. Der Lieferant des RAHMENWERKS PLUS liest Parameter aus der *Konfigurationsdatei*, hier orange unterlegt, aus. Sollte der gewünschte Parameter nicht vorhanden sein, wird ein *Dialog* geöffnet und der Benutzer zur Laufzeit der Systemerzeugung zur Eingabe aufgefordert. Sollten andere Arten der Parameterbeschaffung notwendig sein, müssen diese durch einen vom Generator mitgelieferten *Parameter-Lieferanten* besorgt werden. Das Auflösen der Parameter kann beliebig gehandhabt werden, zum

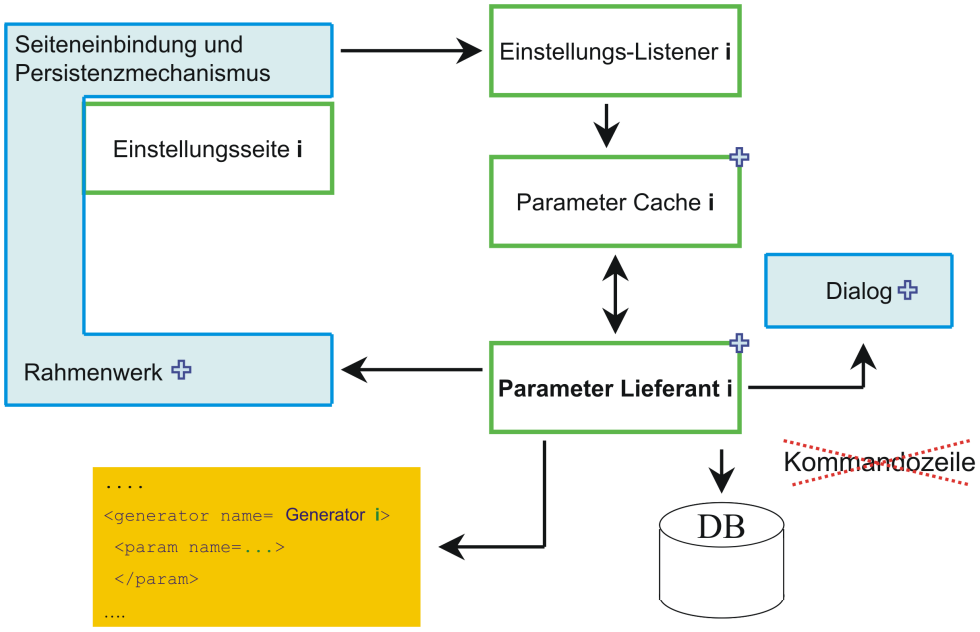


Abbildung 3.2: Parameterbeschaffung für Generatoren im Rahmenwerk Plus

Beispiel durch das Ansprechen einer *Datenbank*[1]. Sollen Parameter vom Benutzer über eine grafische Oberfläche eingegeben werden, so bietet das RAHMENWERK PLUS die Möglichkeit, *Einstellungsseiten* zu publizieren. Die Seiten müssen dabei von dem Generator-Ersteller für einen bestimmten Generator definiert werden. Es soll jedoch nicht verlangt werden, dass er in einer bestimmten Syntax grafische Elemente arrangieren muss. Vielmehr soll der Entwickler nur drei Dinge für einen Parameter festlegen.

- Die Art des Parameters: Dateiname, Verzeichnis, Zeichenkette, logischer Wert, etc.
- Der interne Name zum Speichern und Auflösen des Parameters.
- Eine Zeichenkette, die dem Benutzer für die Parameterauswahl angezeigt wird.

Um die Speicherung und das Wiederherstellen des vom Benutzer eingegebenen Parameters kümmert sich das Rahmenwerk. Ein Parameter-Lieferant kann dann vom Rahmenwerk unter Verwendung des internen Namens eingegebene Werte auflösen.

Damit nicht für jeden Lauf die Parameter neu beschafft werden müssen, werden diese zwischengespeichert. Dafür bietet das RAHMENWERK einen Parameter-Cache. Dieser hält die Werte vor, bis entweder die gesamte Umgebung schließt, die Werte durch Änderung der Konfiguration ungültig werden oder bis der Benutzer manuell zum Löschen der Werte auffordert. Sollen die Werte invalidiert werden, wenn der Benutzer an den grafischen Einstellungsseiten

[1] Hier durch das Symbol mit DB angedeutet.

Änderungen vornimmt, muss der Generator eine eigene Komponente als Ereignisempfänger eintragen lassen, die dann den Zwischenspeicher für ungültig erklären kann.

3.3 Lebensdauer Plus

Im Vergleich zu dem Kommandozeilen gesteuerten Übersetzer hat sich die Lebensdauer der Komponenten im Rahmenwerks Plus verlängert. Ursprünglich wurde das gesamte Java-Laufzeitsystem mit einem Systemerzeugungsprozess zusammen gestartet und beendet. Das ist offensichtlich für eine integrierte Übersetzung nicht praktikabel. Benötigte Instanzen wie Generatoren, Parameter-Lieferanten, Parameter-Caches und Beobachter wie auch Parameter können für weitere Läufe gehalten werden.

Dadurch wird zweierlei erreicht. Instanzen, die in einem fortlebenden Java-Laufzeitsystem nicht mehr referenziert werden, würden so lange Speicher belegen, bis das Laufzeitsystem diese erkennt und beseitigt.

Das Instanzieren der Komponenten nimmt Rechenzeit in Anspruch. Es könnten in der Initialphase einer Komponente lang andauernde Maßnahmen notwendig sein, wie zum Beispiel die Kontaktaufnahme zu einer entfernten Datenbank.

Es ergeben sich auch Nachteile aus der Lebensdauer Plus. Da das Java-Laufzeitsystem nicht nach der Übersetzung beendet wird, werden reservierte Ressourcen nicht automatisch freigegeben. Ein Generator, der sich auf automatische Freigabe verlassen hatte, wird nun nur *einmal* lauffähig sein, da er bei dem zweiten Durchlauf dieselbe Ressource nicht mehr reservieren kann. Der Generator muss beim neuen Rahmenwerk **ablaufinvariant** sein.

Wenn sich die Voraussetzungen, die für die Wahl der Komponenten gesorgt haben, ändern, müssen die Komponenten invalidiert werden. Das gleiche gilt für zwischengespeicherte Parameter. Das ist jedoch ein im Vergleich zum Leistungsgewinn der Zwischenspeicherung ein geringer Mehraufwand.

3.3.1 Verwendung von Caches

Caches haben einige Nachteile: Die enthaltenen Werte müssen aktuell gehalten werden, Caches benötigen Speicherplatz, Referenzen auf sie müssen gesammelt werden und es benötigt Zeit, sie zu initialisieren. Es muss darauf geachtet werden, dass sich die Voraussetzungen für die im Cache enthaltenen Werte nicht ändern, besser gesagt, es müssen die Werte eines Caches bei Änderung aktualisiert werden.

Wenn ein Wert in einer Datei gespeichert ist und als Parameter für den Lauf eines Generators dienen soll, gibt es zwei Möglichkeiten. Zum einen kann man den Wert zum Zeitpunkt, an dem man ihn benötigt, aus der Datei auslesen, ihn benutzen und dann verwerfen. Zum anderen kann man ihn vor der ersten Benutzung, beziehungsweise auch direkt davor, auslesen, speichern, benutzen und halten. Das erscheint bei einmaliger Nutzung Ressourcenverschwendung zu sein. Ohne Zweifel ist es das auch wenn der Wert nur einmal benutzt werden soll. Braucht man den sich über einen unbestimmten Zeitraum nicht ändernden

Wert öfter, fällt das zeitintensive Auslesen des Wertes weg. Dies wäre eine nicht unerhebliche Leistungssteigerung bei vielen Werten und häufiger Benutzung. Es stellt sich demnach die Frage, wie häufig ein Wert unverändert benutzt werden kann, zumal die Anforderungen der Parameter erst dann feststehen, wenn auch das Zwischenmodell aus der Analysephase vorliegt[2].

3.3.2 Bewertung von Caches im Rahmenwerk Plus

Es werden folgende Annahmen gemacht:

- Ein Kompilationsprozess findet *häufig* statt.
- Es herrscht keine *Speicherknappheit*.
- Der Verwaltungsaufwand der Caches ist im Vergleich zur Anzahl der zu cachenden Objekte nicht *überproportional* hoch.
- Die Änderungsrate der Werte ist *gering* über die Zeit.
- Einige der in den Caches gespeicherten Werte sind nur *langsam* zu beschaffen.
- Alle beteiligten Komponenten sind mit der Verwendung von Caches einverstanden.

Alle Punkte, bis auf den letzten, beinhalten einen *relativen* Anteil, der individuell zu beurteilen ist. Für den Fall, dass eine Komponente des Rahmenwerks nicht mit zwischengespeicherten Werten arbeiten kann, muss eine Möglichkeit geschaffen werden, Caches zu umgehen, oder besser noch Caches individuell zu gestalten.

Ein Beispiel dafür wird in Abschnitt 3.7.4 dargelegt, in dem es um in die eclipse-Umgebung integrierte Einstellungen geht.

Für die Abschätzung ist es notwendig, einen typischen Anwendungsfall des Kompilierers zu betrachten. In der Analyse der Arbeit wird ein solcher beschrieben (Abschnitt 2.5). Der dort beschriebene Asset-Modell Ersteller Abschnitt 2.5.1 ist der, auf den es ankommt. Er ist der Kunde, an den sich später das System richtet[3]. Professor Wagner definiert in diesem Anwendungsfall das Modell und ist darauf angewiesen, dass eclipse ihm Hilfen für seine Arbeit bereitstellt. Beim Speichern des Modells wird dieses überprüft und Fehler werden bekannt. Da der Professor sich mit der Definitionssprache noch nicht gut auskennt, ist gerade diese Funktion von großer Wichtigkeit. Die Parameter der Generatoren ändern sich so gut wie nie bei dem Professor.

Das ist genau ein Fall, für den es zweifellos Sinn macht, Caches zu haben. Bei häufiger Kompilation werden die notwendigen Parameter nur einmal aufgelöst und oft verwendet. Die Caches werden selten erneuert und somit entfällt auch dieser Rechenaufwand. Die beteiligten Verarbeitungsinstanzen brauchen auch nicht immer wieder neu erzeugt werden.

[2] Man vergleiche hierzu Abschnitt 3.1

[3] und der später den Wert des Systems honoriert

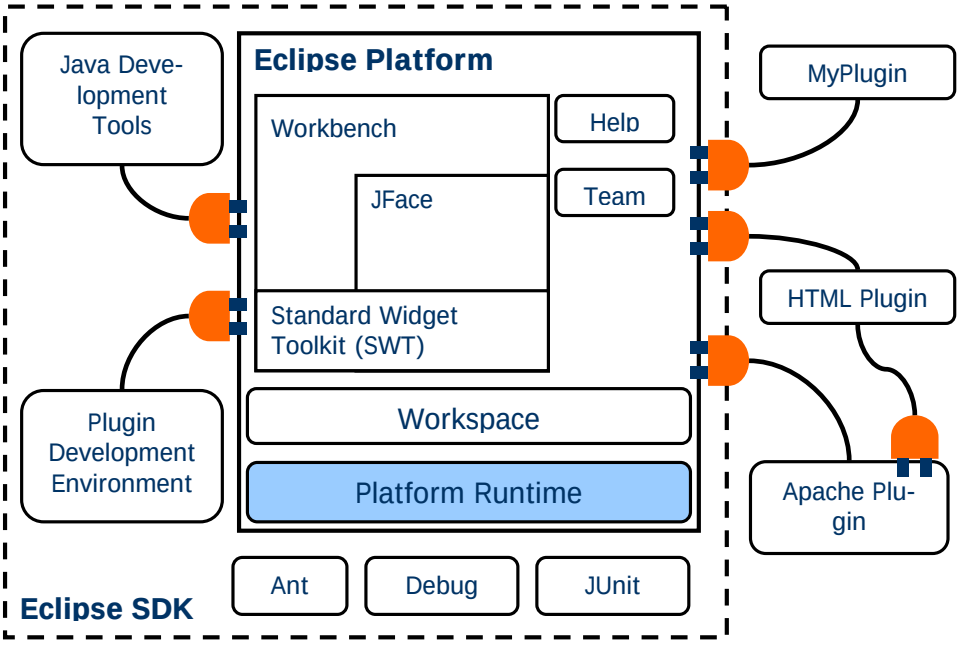


Abbildung 3.3: Eclipse Architektur

3.4 Architektur von eclipse

Um die vorgestellten Teile des Designs zusammen mit den Anforderungen aus Abschnitt 2.6 realisieren zu können, wird eclipse als Plattform verwendet. Die Auswahl eclipses wird im Abschnitt 2.7.4 beschrieben.

Eclipse besteht aus einem Kern, der in der Lage ist, Plugins auszuführen und aus diversen Plugins. Zur Verdeutlichung dient die Grafik 3.3. Wie man aus der Grafik erkennen kann, besteht das herunterladbare Eclipse Standard Development Kit (SDK) aus mehreren Plugins und der Einheit, die für den Ablauf der Plugins verantwortlich ist, hier 'Platform Runtime' genannt. Die Plugins: Java Development Tools, kurz JDT, so wie das Plugin Development Environment, kurz PDE, sind nützlich für die Erstellung eigener Plugins. In eclipse besteht quasi alles aus Plugins. Der minimalistische Kern eclipses, der nicht vollständig aus Plugins besteht, wurde in der Version 3.0 stark überarbeitet. Dieser Kern ist für das Laden der Komponenten verantwortlich. Wie das Laden von Klassen in eclipse bewerkstelligt wird, wird im Abschnitt 3.6 genauer betrachtet. In den folgenden Abschnitten werden Begriffe erläutert, die öfter benötigt werden. Sie unterteilen eclipse logisch in die Bereiche: Ablaufmanagement, Speicherzugriff und Benutzeransicht.

3.4.1 Erweiterbarkeit von eclipse

Eclipse besteht aus einem Laufzeitkern und beliebigen Plugins. Die Plugins beschreiben sich selbst in ihren Manifestdateien. Diese Manifestdateien werden von eclipse ausgelesen, ohne dass das zugehörige Plugin geladen werden muss.

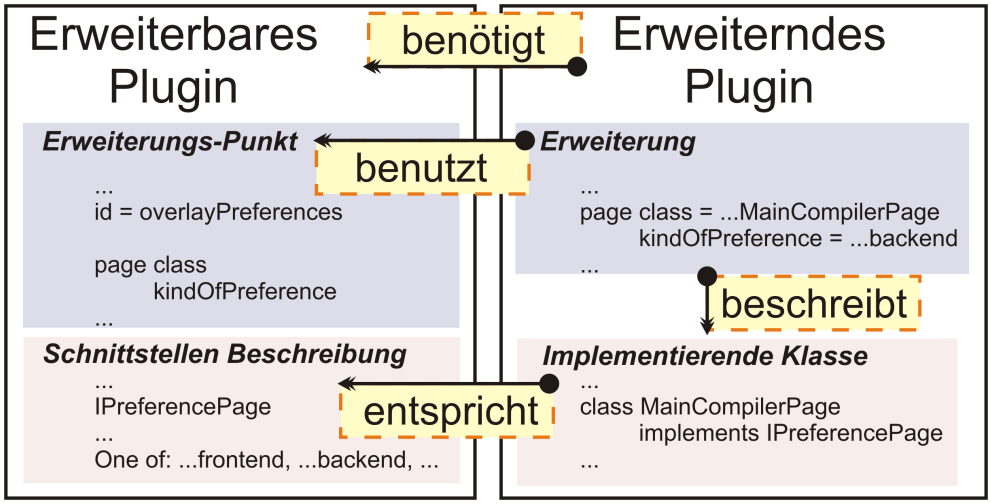


Abbildung 3.4: Erweiterungskonzept eclipses

Eine *Plugin-Registry* wird erstellt und gefüllt.

Damit Plugins geladen werden, müssen sie sich und ihre Einstiegspunkte bekannt machen. Die Manifestdateien `plugin.xml` und `MANIFEST.MF`, die jedes Plugin je nach eclipse-Version aufweist, enthalten alle wichtigen Informationen, die für die Zusammenstellung der Plugins notwendig sind.

Es werden Abhängigkeiten zwischen Plugins definiert, um die Reihenfolge, in der diese geladen werden müssen, einhalten zu können. Plugins werden erst geladen, wenn sie benötigt werden; man spricht hier von 'lazy loading'. Dazu werden alle Manifestdateien gelesen und Stellen, an denen sich ein Plugin beteiligen möchte, herausgestellt. Solche Stellen werden als *extension-points* (Erweiterungspunkte) bezeichnet. *Extension*s (Erweiterungen) werden von den Plugins definiert, um an diesen *Erweiterungspunkten* geladen zu werden. Dieses soll durch die Abb. 3.4 verdeutlicht werden. Das erweiternde Plugin beschreibt eine Abhängigkeit zu dem Plugin, das den Erweiterungspunkt bereitstellt. Wenn das den Erweiterungspunkt anbietende Plugin aktiviert ist und erweiternde Plugins benötigt, kann es die *Plugin-Registry* nach Erweiterungen durchsuchen. Danach wird eclipse dazu aufgefordert, Klassen, die hinter den Eintragungen stehen, zu laden. Dadurch werden dann erweiternde Plugins aktiviert und durchlaufen ihre Initialphase. Das eine Erweiterung anbietende Plugin bekommt von eclipse also Klassen zurückgeliefert. Wichtig ist nun, dass diese Klassen aus den erweiternden Plugins genau die Schnittstelle implementieren, wie sie in der Erweiterungspunkt-Beschreibung angegeben worden sind.

Beispielsweise wird das Kompilierer Plugin erst dann geladen, wenn eine Komponente davon benötigt wird; eine Datei eines Projektes mit der „Asset Definition Nature“ (Siehe Abschnitt 4.2) wird gespeichert, der Einstellungsdialog eines Plugins wird betrachtet, etc.

Seit der Version 3.0 ist eine Liste der Plugins über einen sogenannten *BundleContext* und die *Erweiterung* über eine *ExtensionRegistry* zu erreichen. Das Konzept der *Erweiterungen* ist erhalten geblieben.

3.4.2 Eclipse Workspace

Der Workspace organisiert die Verwaltung von Projekten und deren Metadaten. Projekte können verschiedene Naturen haben: Java, C, Plugin, etc. Alle Dateien werden in Verzeichnissen gespeichert, auf die auch von außerhalb von eclipse zugegriffen werden kann. Dateien, Verzeichnisse und Projekte werden als `Resource`n bezeichnet und es werden vom Workspace Betriebssystemunabhängige Zugriffsmechanismen zur Verfügung gestellt. Außerdem werden Metadaten gehalten. Auch die Fähigkeiten zur Unterstützung für das Programmieren im Team und zur Rekonstruktion von Quellkode aus einer automatisch generierten Historie sollen hier kurz Erwähnung finden.

3.4.3 Eclipse Workbench

Mit dem Workbench wird jene Plugin-Sammlung bezeichnet, welche für die Darstellung von eclipse-Plugins zuständig ist. Es können Menüs erweitert und Editoren, Views und Navigatoren angezeigt werden. Grafische Elemente können zu sogenannten Perspektiven zusammengestellt werden. Eine Unterstützung für Dialoge und Hilfeeinbindung machen eclipse benutzerfreundlicher. Bestandteil des Workbench ist das 'Standard Widget Toolkit' (SWT), welches, anders als AWT bei Java, grafische Elemente durch die im Betriebssystem vorhandene Oberfläche darstellt. Somit wird die gewohnte Arbeitsumgebung erhalten und viel an Zuverlässigkeit und Geschwindigkeit gewonnen. Abstrahiert wird SWT durch JFace. JFace stellt Ansichten, Editoren und andere grobkörnige Grafikelemente zur Verfügung. Benutzereingaben werden von dem Workbench entgegengenommen und nach dem Prinzip ereignisbasierter Systeme weitergeleitet. Der Aufbau kann mit folgenden Entwurfsmustern nach [2] beschrieben werden: Proxy[4], Memento[5], Adapter[6]. Detailliert beschrieben wird der Aufbau in [7].

3.5 Eclipse Builder-Konzept als Bearbeitungslogik

Um es noch einmal zu verdeutlichen: eclipse ist eine Ablaufumgebung für Werkzeuge. Im SDK eclipses ist eine Unterstützung für Java enthalten, andere Bearbeitungslogiken können in eclipse integriert werden. Das bedeutet, dass eine Möglichkeit vorgesehen sein muss, beliebige Dateien zu interpretieren. Diese Interpretatoren nennen sich Builder. Mehrere Builder können einem Projekt zugewiesen werden. Als Beispiel sei hier für ein Java-Projekt der Java-Builder genannt. Ein Projekt ist ein Java-Projekt, wenn es die Natur: `org.eclipse.jdt.core.javanature` (JavaNatur) besitzt. In diesem Abschnitt wird erst der Bedeutung von Naturen und Buildern im Allgemeinen nachgegangen. Die Vorgehensweise von Buildern im Allgemeinen (Abschnitt

[4]Mehr dazu im Abschnitt 5.1.
[5]Speichermechanismus zur Wiederherstellung von Ansichten über Sitzungen hinaus.
[6]Ressourcen verschiedener Art können unter Umständen gleich behandelt werden.

3.5.2) ist ein Konzept, das zu der Verwendung von eclipse für die Übersetzer-Unterstützung eingeladen hat. Sehr viel komprimierter als in den Folgesektionen beschreibt der Artikel [9] die Funktionsweise von eclipse-Buildern.

3.5.1 Naturen von Ressourcen

Die Hauptaufgabe von Naturen ist es, ein Projekt mit gegebenen Werkzeugen, Plugins und Eigenschaften zu verknüpfen. Dabei kann ein Projekt eine oder mehrere Naturen (engl: nature) haben. Oben genannt wurde die Natur: JavaNatur. Das JDT-Plugin ist mit jenen Projekten verknüpft. Das Vorhandensein von bestimmten Naturen ist ausschlaggebend für viele Automatismen. Öffnet man z.B. den Einstellungsdialog eines Projektes[7], so erscheinen Seiten, die nur in einem Zusammenhang mit Java-Entwicklungen Sinn machen, siehe Abb. 3.5. Menüs gestalten sich entsprechend den zugeordneten Naturen. Die Naturen fungieren hier als Filter.

Naturen können auch eine *Art* haben, zum Beispiel: *xml-nature*. Plugins haben die Möglichkeit, sich diese allgemeinere Gruppenzugehörigkeit ebenso wie die spezielle Natur zu Nutze zu machen. Auch Vorbedingungen für die Zuweisbarkeit, oder besser zu deren Aktivierung, können durch die Art einer Natur limitiert sein. So macht es vielleicht nur Sinn, keine spezielle, durch ein Plugin definierte, sondern eine allgemeine Natur von der *Art*: xml-nature aktiv zu haben[8], da sich die realisierenden Plugins ansonsten in die Quere kommen würden.

Abhängigkeiten zwischen Naturen können beschrieben werden. Wo das wichtig ist, wird im Abschnitt 3.5.5 deutlich. Falls eine Kompilierung der Java-Ressourcen vor den Aufgaben einer anderen Natur zu erledigen ist, wird die JavaNatur als Voraussetzung für die Lauffähigkeit der Natur angegeben[9].

Naturen sind eng mit *Buildern* verknüpft. Bevor zu einer genauen Beschreibung der Builder übergangen wird, sollen noch kurz die Vorteile der Assoziation von Naturen zu Buildern Erwähnung finden. Sollte eine Natur nicht lauffähig sein, es wurden oben Bedingungen an eine Natur erwähnt, werden zugewiesene Builder zur Laufzeit übergangen. Das kann auch Sinn machen, falls zwei Entwickler gemeinsam an einem Projekt arbeiten und einer der beiden nicht alle notwendigen Plugins zur Verfügung hat. Der Build-Prozess würde dann stillschweigend nur dort ausgesetzt werden.

3.5.2 Build-Arten

Builder sind Verarbeitungsklassen, die automatisch aufgerufen werden können, wenn sich etwas in einem Projekt ändert. Dieses kann ein Übersetzer sein oder auch beliebige andere Verarbeitungslogik. Im Folgenden wird auch manchmal von Kompilierern stellvertretend für Builder gesprochen. Ein *Ant-Builder*[10]

[7] *Properties* sind projektbezogen, nicht zu verwechseln mit *Preferences*, die für den gesamten Workspace gelten.
[8] Man spricht hier von einem *one-of constraint*.
[9] Auch *requires constraint* genannt.
[10] Siehe Abschnitt 2.7.1.

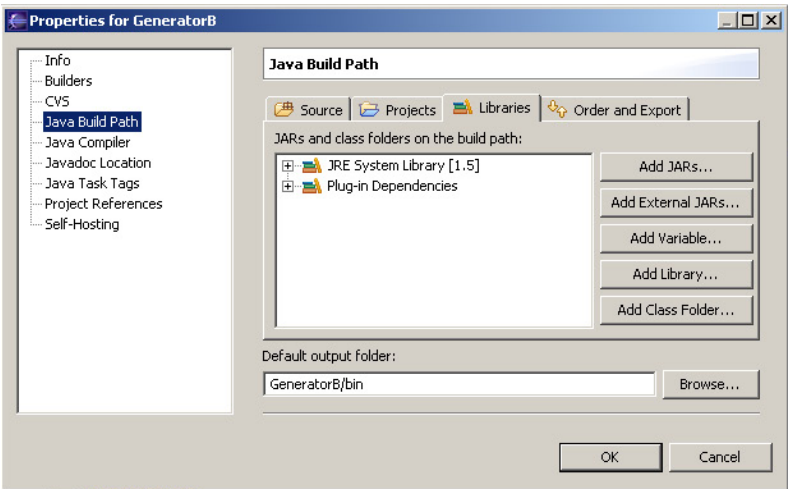


Abbildung 3.5: Projektbezogener Einstellungsdialog eines Projektes

beispielsweise interpretiert *Ant-Skripte*, die in einem Projekt residieren.

Aus dem obigen Abschnitt sind Naturen bekannt. Eine Zuweisung von Buildern geschieht meistens mit dem Eintragen einer Natur. Eine Natur kann auch mehrere Builder eintragen.

Es sind drei Arten des Build-Prozesses zu unterscheiden.

- **Incremental Build** – Wann immer im Workbench *Build Project* oder *Build All* aufgerufen wird wird solch ein Prozess ausgeführt. Ein Builder wird in diesem Fall mit einem `resource delta` (Delta) aufgerufen. Dieses ist eine hierarchische Beschreibung von allen Änderungen seit dem letzten Build-Prozess. Ziel dieses Vorgehens ist es, nur dann Aktionen zu starten, falls die Änderungen dieses auch notwendig machen. Ein Builder hat demnach die Aufgabe, das Delta dahingehend zu untersuchen. Soweit die normale Vorgehensweise. Sollte ein Kompilierer lange Zeit oder noch gar nicht während des Lebenszyklus' von eclipse ausgeführt worden sein, ist das Delta leer. Eclipse nimmt sich das Recht heraus, ein Delta nach einer unbestimmten Anzahl von Änderungen zu löschen. Es wird hier abgewogen zwischen den Kosten für eine Speicherung der Änderungen und den Kosten eines *Full Build* (siehe nachfolgend).

- **Auto Build** – Autobuild und Incremental Build unterscheiden sich lediglich in der Art des Aufrufs. Bei jeder Änderung an `Resource`n wird Autobuild ausgeführt, die Implementation im Kompilierer ist demnach identisch. Autobuild kann in den globalen Einstellungen an- und ausgestellt werden. Seit neuesten Spezifikationen[11] wird der automatische Build-Prozess auch von Zeit zu Zeit ohne das Speichern von Dokumenten im Hintergrund ausgeführt.

[11] Zur Zeit: eclipse 3.0

- **Full Build** – Komplettes Kompilieren ohne Rücksicht auf die Notwendigkeit. Egal ob ein Kompilieren etwas Neues erzeugt oder nicht, alle Stati werden verworfen und der Übersetzungs-Prozess beginnt von neuem. Nach dem Befehl *clean...* wird ein Full Build ausgeführt.

Es stellt sich die Frage, warum nicht nur auf Veränderung einer Ressource reagiert werden sollte. Dieses Konzept ist schon lange bekannt und auch auf anderen Plattformen verfügbar. Solch eine Vorgehensweise wäre einfach dadurch zu implementieren, dass man im Workspace einen *ResourceChangeListener* einträgt.

3.5.3 Gegenüberstellung: Beobachter von Ressourceänderungen – Builder

`ResourceChangeListener` (RCL) sind Beobachter nach [2] und werden bei Änderungen an Ressourcen sofort in beliebiger Reihenfolge informiert. Es folgt eine stichwortartige Auflistung zum Vergleich von RCL und Buildern.

- Einstellung Autobuild an: Nahezu gleiche `resource delta`s für RCL und Builder, Builder empfangen jedoch keine Marker-Deltas[12] oder Synchronisationsinformationen.
- Einstellung Autobuild aus: Builder werden nur gestartet, wenn sie explizit dazu aufgefordert werden.
- Builder haben eine Reihenfolge.
- Build-Reihenfolge der Projekte wird durch Abhängigkeiten festgelegt.
- Benutzer wählt *Build all*: Builder werden für alle Projekte zu diesem Zeitpunkt gestartet. Für RCL ist so etwas nicht vorgesehen.
- Builder können voneinander abhängen, RCL nicht. Einstellungen für Builder bleiben auch nach der Synchronisation von *Team*-Projekten erhalten. Einstellungen werden in einer Meta-Datei gespeichert und bei der Verteilung mit aktualisiert.
- RCL werden über alle Änderungen informiert, Builder nur über die seines Projektes und über diejenigen, für die er Interesse bekundet hat.
- Die Ausführungsreihenfolge kann entscheidend sein. In welcher Reihenfolge Ereignisse versandt werden, ist unbestimmt; man kann sich jedoch versichern, ein Ereignis vor einem Build-Prozess zu bekommen.
- Builder haben eine Unterstützung für eine Überwachung des Fortschritts, für den Abbruch und für die Fehlerausgabe.
- Oben genannte Build-Arten werden unterschieden.

[12] Änderungen an Markern, siehe Abschnitt 3.5.6

- Lebensdauer und Persistenz eines Builder wird von der Plattform verwaltet. Einmal für ein Projekt eingetragen, bleibt der Builder auch über mehrere Sitzungen zugewiesen.

3.5.4 Ablaufreihenfolge von Beobachtern und Buildern

Die Ereignisarten, denen `resource delta` beigefügt sind (`POST_CHANGE, PRE_BUILD, POST_BUILD`[13]), beinhalten Informationen über Änderungen, die zwischen zwei diskreten Zeitpunkten im Workspace vorgefallen sind. Welches Zeitintervall ein `resource delta` genau protokolliert, ist eine genauere Betrachtung wert. Zum Verständnis sind Zwischenschritte eingebaut, die solche diskreten Zeitpunkte markieren. Die nun folgende Tabelle schafft eine Übersicht über die Intervalle der einzelnen Ereignisarten, wie sie bis zur eclipse-Version 2.1 gültig sind.

1. Workspace-Zustand 1
2. Irgend eine Änderung tritt auf
3. Workspace-Zustand 2
4. Alle `PRE_BUILD` listener werden über alle Änderungen zwischen den Zuständen 1 und 2 informiert.
5. Die Inkrementellen Builder werden gestartet.
6. Workspace-Zustand 3
7. Alle `POST_BUILD` listener werden über alle Änderungen zwischen den Zuständen 1 und 3 informiert.
8. Workspace-Zustand 4
9. Alle `POST_CHANGE` listener werden über alle Änderungen zwischen den Zuständen 1 und 4 informiert.

Zu beachten ist, dass sich die Zeitintervalle der unterschiedlichen Ereignisarten überlappen. Falls ein Beobachter auf mehr als eine Ereignisart reagiert, hat man darauf zu achten, dass Änderungen in den `resource delta`s doppelt auftauchen können. Meistens ist es am geeignetsten, einen `listener` unter `POST_CHANGE` zu registrieren, da dann alle Änderungen seit der letzten Meldung erfasst sind.

Seit der eclipse-Version 3.0 hat sich an diese Stelle einiges geändert. Da das Rahmenwerk Plus Plugin während der Implementationsphase entsprechend nachbearbeitet worden ist, sollen die Änderungen auch hier vorgestellt werden.

[13] Ehemals: `PRE_AUTO_BUILD und POST_AUTO_BUILD`

Der Build-Prozess läuft nun im Hintergrund ab, die oben genannten Punkte 5, 6, 7 und 8 entfallen. Dafür werden parallel, zu nicht genau definierten Zeitpunkten, diese Aktionen ausgeführt. Der Build-Prozess läuft also separat ab. Alle Änderungen werden erfasst und den `PRE_BUILD`- sowie den `POST_BUILD-listenern` vollständig mitgeteilt. Keine Änderungen gehen denen nunmehr verloren. Man beachte, dass Änderungen am Workspace nicht unbedingt unmittelbar die Benachrichtigung von `BUILD-listenern` nach sich ziehen.

Änderungen an Konfigurationsdateien müssen vor dem Build-Prozess von dem Rahmenwerk übernommen werden, damit die Systemerzeugung mit den neuen und nicht mit den zwischengespeicherten Einstellungen vorgenommen wird. In der eclipse Version 2.1 reicht dafür ein `PRE_BUILD-listener` nicht aus, da Änderungen zwischen Zustand 2 bis Zustand 1 des neuen Durchlaufs keine Aktivierung nach sich ziehen. Eclipse ist in seiner Version 3.0 aus diesem Grund besser für die vorliegende Aufgabe geeignet als Vorgängerversionen.

3.5.5 Ablaufreihenfolge der Builder

Wie schon zuvor erwähnt, laufen eingetragene Builder in einer definierten Reihenfolge ab. Grobkörnig ist erst einmal die Reihenfolge festzulegen, in der die Projekte abgearbeitet werden. Dann folgt die Reihenfolge innerhalb der Projekte. Die sogenannte *workspace build order* kann sowohl von einem Plugin über eine Methode des eclipse Workspace gesetzt werden, wie auch vom Benutzer in der eclipse-Benutzeroberfläche unter *Build Order*[14].

Vorsicht ist dabei geboten; Beziehungen der Projekte zueinander müssen beachtet werden. Wird die Reihenfolge manuell nicht festgelegt, so wird sie von eclipse einmalig, auf der Basis der Abhängigkeiten der Projekte untereinander, wie folgt berechnet: erst werden alle Projekte mit Abhängigkeiten aufgelistet[15], danach all jene, die noch übrig sind. Geschlossene Projekte werden ignoriert.

Die Ausführungsreihenfolge der Projekte ist aufgezeigt. Nun wird jene innerhalb der Projekte betrachtet. Es können mehrere Builder für ein Projekt eingetragen sein. In der Projektbeschreibung ist die Reihenfolge festgelegt, vgl. hierzu Abb. 3.6. Die Abb. zeigt ausgewählte Instanzen zur Laufzeit. Es wird ist eine Liste von `ICommand`-Objekten angelegt. Diese *Befehle*, vergl. [2], können auch mit Parametern versehen werden, die dann an die Instanz der realisierenden Klasse beim Aufruf: `build()` übergeben werden. Die Befehle werden in der Reihenfolge der Listeneinträge nacheinander ausgeführt. Das Erstellen der Befehle wird üblicherweise von den Naturen übernommen. Bei dem Eintrag einer Natur muss diese explizit einen Befehl an die richtige Stelle einreihen; dabei sind Abhängigkeiten der Naturen untereinander zu berücksichtigen.

Dann muss im Einzelnen entschieden werden, ob ein Builder ausgeführt wird oder nicht. Bei einem `full build` (siehe Abschnitt 3.5.2) findet der Aufruf statt. Bei `incremental-` oder `auto build` wird die Notwendigkeit zuvor überprüft. Der Build-Vorgang findet nur statt, wenn eine Ressource von Inter-

[14] Menüpunkt: *Window* →*preferences* →*Build Order*
[15] Zyklische Abhängigkeiten werden beliebig aufgelöst.

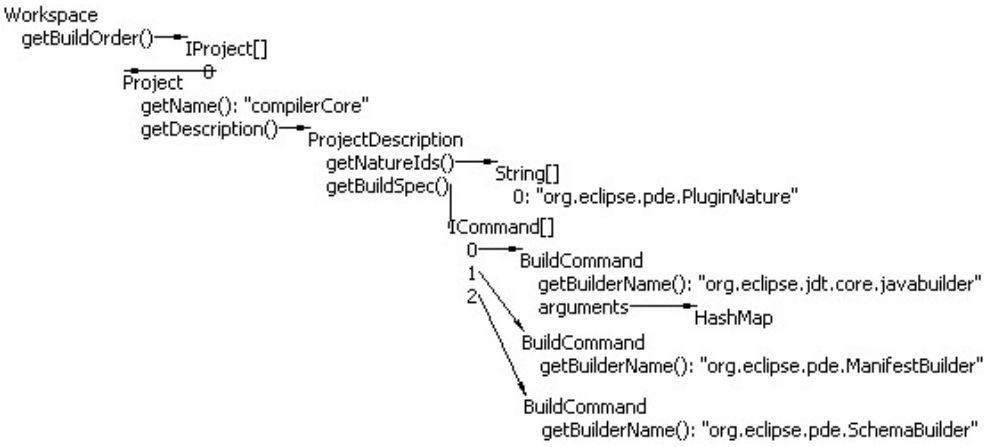


Abbildung 3.6: Ablaufreihenfolge der Builder

esse verändert worden ist. Der Builder hat nach seinem Durchlauf zurückzugeben, welche Projekte in seinem Sichtbereich sein sollen. Projekte, bei denen der Builder eingetragen ist, sind nicht aufzuführen. Somit kann beim nächsten Durchlauf evaluiert werden, ob Änderungen eine Ausführung des Builder notwendig machen.

Es gibt auch Fehlersituationen, die das Auslassen eines Builders begründen. Sobald das Problem behoben ist, nimmt der Builder wieder an dem Prozess teil. Folgende Fehlerbedingungen sind denkbar:

- Das den Builder beinhaltende Plugin fehlt
- Das den Builder beinhaltende Plugin ist als `disabled` gekennzeichnet. Das geschieht entweder durch einen Fehler während der Aktivierung oder durch eine fehlende Voraussetzung.
- Ein Fehler bei der Instanzierung des Builder ist aufgetreten
- Der Builder gehört zu einer Natur, und diese Natur fehlt oder ist *disabled*.

3.5.6 Problembehandlung während des Build-Prozesses

Buildern steht der Mechanismus des Werfens von *Ausnahmen*[16] zur Verfügung. Wird eine `Exception` zur Laufzeit eines Builder geworfen, bleiben andere Builder dennoch aktiv und deren Ablauf wird dadurch nicht gestört.

Das Werfen von Ausnahmen wird üblicherweise bei fehlerhaftem Kode des Builder benutzt. Fehler im Umgang mit eclipse-eigenen Methoden werden zumeist auch als Ausnahmen weitergeleitet.

Falls in der Analysephase lexikalische oder syntaktische Mängel festgestellt werden, sollten sogenannte `Marker`[17] eingesetzt werden. Damit wird auch, wie

[16] In Java `Exception` genannt.
[17] Eine Subklasse des Typs `IMarker.PROBLEM`

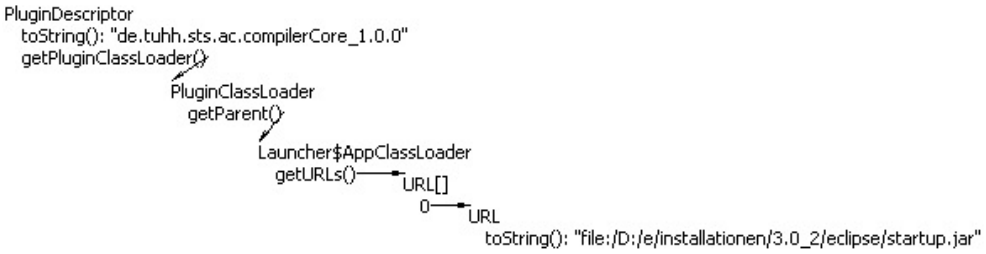


Abbildung 3.7: Übergeordneter Plugin-`class-loader`

in der Analyse Abschnitt 2.6 gefordert, der Benutzer bei der Erstellung eines Modells unterstützt.

Es gibt drei Arten von Markern:

- Problems - um ungültige Stati auszudrücken
- Tasks - um benutzerdefinierte Erinnerungspunkte zu markieren[18]
- Bookmarks - um einen Ort zu markieren, an dem man später schnell springen möchte

Erstere Fehlermarker werden nach Beendigung der Analysephase dem Benutzer gezeigt, und die Synthesephase wird nicht eingeleitet. Die Anzeige erfolgt durch eine bereits in eclipse integrierte Ansicht. Wie solche Marker eingebunden werden, wird im Abschnitt 4.3 im Implementation-Kapitel beschrieben.

3.6 Komponenten-Installation und dynamische Pfaderweiterung

Bei eclipse hat jedes Plugin seinen eigenen `ClassLoader` (Classloader) und sein eigenes Pfadverzeichnis für Klassen. Einige tiefgreifende Änderungen wurden für die eclipse-Version 3.0 eingeführt. Zuerst wird das vorherige Konzept beschrieben und dann auf die Änderungen hingewiesen. So wird ein vollständiges Bild der zugrunde liegenden Konzepte entstehen.

Ein Classloader ist für die Initialisierung von Klassen verantwortlich. Dafür muss der Ort, an dem sich der ausführbare Kode für diese `Klasse` befindet, dem Classloader bekannt sein. Dieser Ort wird durch einen Pfad aufgelöst. Pfade verweisen auf Verzeichnisse oder auf Bibliotheken. Klassen müssen aus der Sicht eines Classloaders eindeutig in ihrer Namensgebung sein. `Pakete` können Klassen enthalten. Die Paketnamen bilden mit dem Klassennamen zusammen die eindeutige Identifikation der Klasse.

Ein eigener Classloader bedeutet auch einen eigenen Namensraum für die Klassen eines Plugins. So können zum Beispiel verschiedene Klassen mit gleichem Namen in separaten Plugins benutzt werden.

[18] Bekannt als `TODO`s

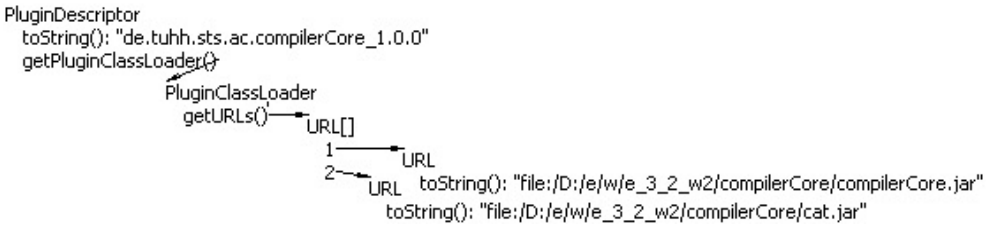


Abbildung 3.8: Einbindung eigenen Kodes und importierter Bibliotheken

Sobald der Plugin-Classloader eine Klasse lädt, wird das Plugin aktiviert, mit allen daraus resultierenden Ladevorgängen. Pfade werden in der folgenden Reihenfolge aufgelöst:

1. **Plugin-Classloader Vater**. Das ist der übergeordnete Classloader. Dieser stellt den Zugang zu den eclipse Start-Klassen zur Verfügung. In der Abb. 3.7 ist zu sehen, dass die Bibliothek: `startup.jar` zur Verfügung gestellt wird.

2. **Plugin-Classloader selbst**. Es wird alles, was in der Manifestdatei unter der Rubrik „runtime“ in dem Plugin definiert ist, durchsucht. In Abb. 3.8 sind Verweise auf Bibliotheken des MODELL ÜBERSETZERS nachzuvollziehen. Die KOMPILIERER-UMGEBUNG PLUS ist in die Bibliothek `compilerCore.jar` gepackt. Des weiteren ist hier beispielhaft eine propagierte Bibliothek aufgezeigt. Diese Bibliothek wird von dem Rahmenwerk selbst benutzt und für Klienten teilweise zur Verfügung gestellt.

3. **Importierte / vorausgesetzte Plugins**. Als letztes wird auf verwiesene Plugins zugegriffen. Auch diese Beziehungen sind in der Manifestdatei definiert (ohne Abbildung). Die Verweise werden über die Plugin-Identifikation geknüpft und nicht über den Pfad ihrer Bibliotheken. Das hat den Vorteil, dass die Entwickler dieser Plugins auch die Gestaltung und Aufteilung ihrer Bibliotheken ändern können, ohne dass es die Funktionalität des abhängigen Plugins beeinträchtigt (siehe [5]).

Es stellt sich nun die Frage, an welcher Stelle die Kette der Classloader endet. Hierzu sei auf Abb. 3.9 verwiesen. Der *Standard Java*-Classloader hat keinen übergeordneten Classloader. Es taucht in dieser Kette kein Classloader auf, der die CLASSPATH-Variable, die im Betriebssystem definiert ist, auflöst.

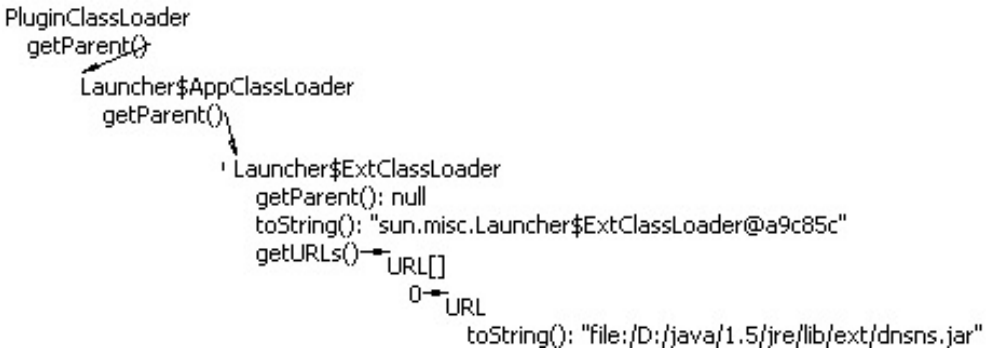


Abbildung 3.9: Pfadauflösung bis zum Java-Classloader

Seit der Version 3.0 sind in eclipse weitreichende Änderungen aufgetreten. Nachdem, wie im ersten Punkt oben beschrieben wird, der übergeordnete Plugin-Classloader konsultiert wurde, wird zwischen zwei Classloadern ausgewählt:

1. Ein anderer Classloader, der seine Verantwortlichkeit zuvor explizit definiert hat.

2. Wie im zweite Punkt zuvor beschrieben: der Plugin-Classloader selbst.

Diese abgeänderte Reihenfolge soll zur Beschleunigung des Ladevorgangs dienen. In der Manifestdatei eines Plugins wird dargelegt, welche Pakete die Bibliotheken enthalten und welche zur Verfügung gestellt werden. So kann *a priori* festgestellt werde, ob ein vorausgesetztes Plugin eine Pfadauflösung ermöglicht oder nicht.

Nachteil dieser Vorgehensweise ist allerdings, dass Plugins, die zuvor[19] funktionierten, nun Fehler aufweisen können. Wenn ein Plugin von dem anderen abhängig ist und die gleiche Bibliothek von beiden Plugins zur Verfügung gestellt wird, so wird ein entsprechender Pfad nicht eindeutig aufgelöst und eine `ClassCastException` wird geworfen. Leider gibt die Ausnahme keinerlei Hinweis auf seinen Ursprung.

Es reicht seit eclipse 3.0 aus, von einem neuen Plugin auf das Übersetzer Plus Plugin zu verweisen, um die Klassen wie
`de.tuhh.sts.cocoma.compiler.generators.Generator`
benutzen zu können[20]. Diese Klasse ist in der Bibliothek `cat.jar`, wie in Abb. 3.8 zu sehen, enthalten. Es kann und muss genau angegeben werden, welche Pakete exportiert, d.h. zur Verfügung gestellt werden sollen.

Es wurde beschrieben, dass eine CLASSPATH-Variable, die im Betriebssystem definiert ist, nicht von den Classloadern eclipses gesehen wird. Eine neue Komponente des Übersetzers kann demnach nicht dadurch eingebunden werden, dass diese Variable um eine zusätzlichen Pfad erweitert wird. Neue Bibliotheken sollen über eigenständige Plugins eingebunden werden. Ein Plugin

[19] Kompatibilität mit eclipse 2.1
[20] Generator-Komponenten müssen eine Erweiterung dieser Klasse enthalten.

beschreibt sich selbst in einer Manifestdatei. In dieser Datei: `plugin.xml`[21] können beliebig viele Bibliotheken publik gemacht werden.

Das bedeutet, dass Komponenten entweder über die Laufzeit-Eintragungen der KOMPILIERER UMGEBUNG PLUS in der Manifestdatei hinzugefügt werden oder als eigenständiges Plugin. Dabei kann auch ein Trick zur Anwendung kommen. Die KOMPILIERER UMGEBUNG kann dynamisch, also während der Laufzeit, Plugins generieren und einbinden. Doch dazu mehr im Kapitel zur Implementation; Abschnitt 4.4.1.

Dass Generatoren, die nicht für den Einsatz in eclipse gemacht wurden, aber dennoch dort lauffähig sein müssen, ist so vorgesehen, um eine Abhängigkeit von eclipse zu vermeiden. Neben dem interaktiven soll auch der Batchbetrieb unterstützt werden. Sollte sich herausstellen, dass Laufzeitfehler durch eclipse verursacht werden, können einmal geschriebene Generatoren weiterhin benutzt werden.

Im Folgenden wird angenommen es handle sich um einen Generator; für andere Komponenten gilt selbstverständlich Entsprechendes. Was für Vorteile entstehen dadurch, dass man ein eigenständiges Plugin manuell definiert und nicht auf eine Bibliothek der alt bekannten Form zurückgreift?

- Plugins können problemlos hinzugefügt werden. Eclipse unterstützt das Erweitern durch Komponenten mit einem Wizard. Zu finden ist dieser unter *help →Software Update →Find and Install....* Hier können auch verschiedene Versionen von Plugins (hier Generatoren) einfach geladen werden. Es ist demnach nicht notwendig, den Kern des KOMPILIERERS zu erneuern, um neue Generatoren einzutragen.

- Generatoren können individuell zusammengestellt werden. Nicht jeder braucht alle Generatoren, sondern nur einen bestimmten Satz. Die Beschreibungen der Plugins können eine Auswahl erleichtern. Der in eclipse integrierte Update-Manager hilft bei der Installation der Komponenten.

- Falls eine deskriptive Beschreibung allein ausreicht, ist keine Instanzierung notwendig. Für eine Übersicht der Generatoren reicht es z.B. aus, die Manifestdateien der Plugins zu lesen, ohne die Plugins zu aktivieren oder irgendwelche Klassen daraus zu laden.

- Abstürze tangieren den Rest des Programms nicht. Falls sich ein Generator als fehlerhaft erweisen sollte, können die anderen Generatoren weiter von dem Kern des ÜBERSETZERS angesprochen werden. Abstürze eines Generators wirken sich nur auf die ihm zugeteilten Aufgaben aus – und auf diejenigen Aufgaben der Generatoren, die auf die Vorarbeit des ersteren angewiesen sind.

- Die Funktionalität eines Generators kann individuell angepasst werden. Es können umhüllende Klassen für den eigentlichen Generator erstellt

[21] Seit eclipse Version 3.0 zusätzlich die Datei `MANIFEST.MF`.

werden, da die Instanzen des Generators nicht direkt angesprochen werden, sondern diejenigen, die in der Manifestdatei angegeben sind[22].

- Die Funktionalität eines Generators kann individuell erweitert werden. Ein Generator-Plugin könnte zusätzliche Funktionen unterstützen. Beispielsweise könnte ein Generator einen Einstellungsdialog liefern und somit die Eingabe von Parametern für diesen Generator erleichtern (Abschnitt 3.7).

Die oben genannten Punkte können auch anders erreicht werden. Die Architektur von eclipse sieht jedoch ausdrücklich die Erweiterung durch Plugins vor und stellt so vieles zur Verfügung, was ansonsten umständlich (besser oder schlechter sei dahingestellt) von Hand programmiert werden müsste.

3.7 Konfiguration der Systemerstellung

Es wurde schon darauf hingewiesen, dass die Umgebung Plus mehrere Aufgaben erfüllen soll, die über die reine Funktionalität eines Kompilierers hinausgehen. Um die Systemerzeugung konfigurieren zu können, wird neben der Konfigurationsdatei, siehe Abschnitt 2.3, eine Persistenzkomponente integriert werden. Für den Anwender der Übersetzung wird sich diese Komponente als grafische Einstellungsseiten darstellen. Im Folgenden werden die in eclipse vorhandenen Mechanismen vorgestellt. Anschließend wird eine Erweiterung erklärt.

3.7.1 Kombinierte Einstellungsseiten: OverlayPages

Generell sind in eclipse zwei Arten von Einstellmöglichkeiten zu unterscheiden. Zum einen Einstellungen für ein Plugin: *Preferences*, zum anderen projektbezogene Einstellungen, die den entsprechenden Ressourcen zugeordnet werden: *Properties*. Während bei ersterem die Schlüssel-Wert-Paare in einem dem Plugin zugewiesenen Speicherort platziert werden, werden Projekteigenschaften in Metadateien der Projekte gespeichert, die in der Speicherhierarchie im Dateisystem liegen. Ressourcen unter eclipse sind auch von außerhalb als Dateien und Ordner erreichbar. Projekte werde als Ordner gespeichert und enthalten eine Datei mit dem Namen: `.project`, in der ihre Einstellungen gespeichert werden. Ein Vorteil dieser Speicherungsform ist es, dass bei der Entwicklung in der Gruppe diese Einstellungen mit verteilt werden, wenn man diese synchronisiert.

Wendet man sich nun der Aufgabe einen einfachen Persistenzmechnismus zu schaffen, zu, so wird man feststellen, dass beide Arten der Speicherung von Eigenschaften notwendig sind. Ein Generator-Entwickler wird seinen Generator in verschiedenen Ausprägungen laufen lassen wollen. Er möchte für jedes seiner Projekte entscheiden können, ob sie automatisch kompiliert werden, wohin sie verteilt werden und mit welchen Parametern die Systemerzeugung versorgt werden soll. Nicht jeder möchte jedoch diese feine Granularität nutzen. Es

[22] Diese Klassen müssen vom Rahmenwerk Plus definierte Schnittstellen erfüllen.

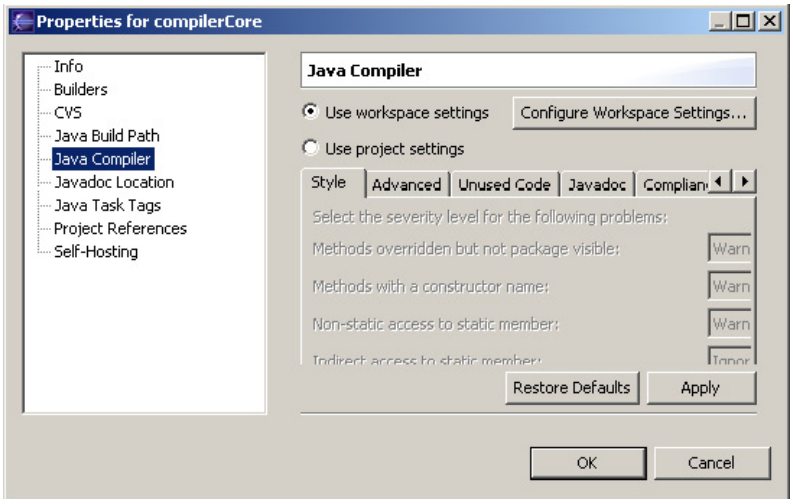


Abbildung 3.10: Projektbezogener Einstellungsdialog des Java-Übersetzers

wäre sehr unpraktisch, wenn Einstellungen, die alle Projekte betreffen, auch in allen separat eingestellt werden müssten. Zudem kann man unterscheiden, welche Einstellungen bei einer Entwicklung in der Gruppe mit verteilt werden sollen und welche nicht.

Es stellt sich somit die Frage, wie diese beiden Einstellmöglichkeiten nun praktikabel kombiniert werden können. Der Einstellungsdialog für den Java-Kompilierer: Abb. 3.10, liefert hier ein exzellentes Vorbild. Es handelt sich hier um die *Properties* des Projektes `compilerCore`. Man kann in Abb. 3.10 gut sehen, dass eine Entscheidung getroffen werden kann, ob die *Preferences* oder die *Properties* gelten sollen. Angenehm ist dann noch zu bemerken, dass man über den Schalter: *Configure Workspace Settings...* direkt zu den *Preferences* gelangen kann und dass man in den *Properties* keine Einstellungen vornehmen kann solange diese nicht aktiviert sind. Letzteres erhöht beträchtlich die Übersicht für den Benutzers.

Solch eine kombinierte Einstellungsseite wird im Folgenden *OverlayPage* genannt werden[23]. Der uniforme Aufbau der *OverlayPages* macht eine zweifache Implementierung für *Preferences* und *Properties* unnötig und hilft dem Benutzer, sich zu orientieren. Die Details der Implementierung werden im Abschnitt 4.6 beschrieben.

3.7.2 Die generische OverlayPage

Wenn ein GENERATOR dem Benutzer Einstellungen anbieten will, könnte er genau diesen soeben beschriebenen Aufbau selbst replizieren, wie es bereits von dem RAHMENWERK PLUS getan wurde. Das ist nicht notwendig. Andersherum kann der GENERATOR auch nicht durch einen fest kodierten Befehl aufgefordert werden, Informationen für eine Generierung von Einstellungsseiten zu liefern.

[23] Nach [14]

Das würde dem Konzept, Generatoren flexibel einbinden zu können, widersprechen. Es muss also eine Möglichkeit geschaffen werden, bei der Erstellung einer *OverlayPage* dynamisch all jene Plugins zu erfassen, die Einstellungen publizieren wollen. Das kann über den Erweiterungs-Mechanismus von eclipse geschehen (siehe Abschnitt 3.4.1).

Den Aufbau der Einstellungsseiten kann man in der Abb. 4.15 auf Seite 80 sehen. In der Abb. findet man einen Reiter mit der Bezeichnung: „GeneratorC Preferences“. Das „GeneratorC-Plugin“ hat keinerlei anderweitige Bedeutung, als dass es zu Demonstrationszwecken eine Erweiterung definiert.

In der linken Seite der Abb. sind drei Menüpunkte für die Kompilierer-Umgebung vorgesehen. Diese Titel beginnen mit *Asset Compiler* und unterteilen die Einstellungen in drei Bereiche.

- Der mit *Analysis* bezeichnete Bereich dient für Einstellungen bezüglich des Frontend des Übersetzer Rahmenwerks.
- Der Bereich *Generation* bezieht sich auf das Backend des Übersetzers und
- *General* deckt übergreifende Einstellungen ab.

Komponenten können wie das „GeneratorC-Plugin“ Einstellungsseiten den drei Kategorien hinzufügen.

Diese Aufteilung sorgt für eine bessere Übersicht für den Benutzer und auch für den Komponenten-Entwickler. Es wird davon ausgegangen, dass noch mehrere Komponenten an die Kompilierer-Umgebung angeschlossen werden.

Es gilt für die meisten Benutzer, dass die Analysephase mit den allgemeinen Einstellungen bei allen Projekten lauffähig ist. Die Einstellungen von Gruppenmitgliedern sollten nicht übernommen werden, sowie auch die eigenen zumeist nicht propagiert werden sollen. Eine Trennung von den Generatoren-Einstellungen ist demnach sinnvoll. Die Einstellungen für Generatoren können für jedes Projekt individuell eingestellt werden müssen.

Sobald der Einstellungsdialog sichtbar werden soll - in dieser Betrachtung ist es egal, ob *Preferences* oder projektbezogene *Properties* - wird eine Instanz der Einstellungsseite der Komponente erstellt. Die *OverlayPage* befragt diese Instanz dann nach den gewünschten Einstellungen und fügt die visuelle Darstellung der Einstellungen dann unter einem neuen Reiter hinzu. Einstellungen sind mit Bezeichnung und eingestelltem Wert anzuzeigen. Nach Beendigung des Einstellungsdialoges müssen eingetragene Werte entsprechend gespeichert werden. Dieses geschieht generisch von dem Rahmenwerk Plus für alle Komponenten.

Der Übergang von Design und Implementation ist in diesem Fall der Integration in die Eclipse-Umgebung fließend. Weiterführendes wird im Abschnitt 4.6 ab Seite 77 beschrieben.

3.7.3 Zugriff auf Werte der kombinierten Einstellungen

Einstellungen werden in den *Preferences* und *Properties* gemacht.Die vom Benutzer eingestellten Werte auch wieder aufgelöst werden.

Die Auflösung der Werte erfolgt über eine Klasse des Rahmenwerks Plus. Andere Plugins als der Übersetzer Kern können nicht losgelöst von diesem auf die Werte zugreifen. Warum das nicht geht, wird deutlich, wenn man an den Speicherplatz der Schlüssel-Wert Paare denkt. Globale Einstellungen werden in einem nur der Kompilierer-Umgebung zugänglichen Speicherplatz aufbewahrt.

Näheres dazu im Abschnitt 4.6.2.

3.7.4 Reagieren auf Einstellungsänderungen

Wenn das Rahmenwerk Einstellungen von Komponenten hinzufügen kann, müssen sie auch in vielen Fällen über Änderungen an den Einstellungen informiert werden. Komponenten müssen sich als Nachrichtenempfänger registrieren. Ein üblicher Mechanismus sieht vor, dass der Empfänger (Listener) sich aktiv registriert. Ein Plugin (z.B. ein Generator) publiziert eine Einstellungsseite und möchte über Änderungen an dieser informiert werden. Eine Abhängigkeit zum Rahmenwerk Plugin muss definiert werden. Bei Instanzierung der Einstellungsseite könnte ein Listener bei dem Rahmenwerk Plugin eingetragen werden, denn um die Einstellungsseite laden zu können, muss das Plugin auch aktiviert werden. Was nun, wenn der Generator an anderen Einstellungen interessiert ist, vielleicht betreffend des *Frontend*? Dann ist das Generator-Plugin möglicherweise noch nicht geladen, wenn Änderungen in einer anderen Rubrik gespeichert werden. Die Einstellungsseite des Generators wäre womöglich gar nicht geladen worden. Außerdem kann es sein, dass Änderungen erkannt werden sollen, ohne dass eine Einstellungsseite beigesteuert werden soll - oder dass ein Teilnehmer an Änderungsbestätigungen nicht interessiert ist, wohl aber an der Publikation von Werten.

Es ist offensichtlich, dass es sich hier bei der Publikation und dem Ereignisempfang um zu trennende Aufgaben handelt. Listener bekommen folglich auch einen eigenen Erweiterungspunkt. Um Benachrichtigungen zu empfangen, trägt sich ein Plugin als Empfänger ein und wird dann bei Änderungen informiert, unabhängig davon, ob es bereits geladen ist oder nicht, somit können auch keinerlei Benachrichtigungen versäumt werden.

Die Details der Implementierung finden sich in Abschnitt 4.6.3.

3.7.5 Konfigurationsdatei zur Ablaufsteuerung der Systemerstellung

Generatoren, die mit dem aus der Kommandozeile zu steuernden Kompilierer funktionieren, sollen auch in der neuen Kompilierer-Umgebung lauffähig sein. Bei dem Kompilierer im Batchbetrieb wird beim Aufruf eine Konfigura-

tionsdatei mit übergeben. Diese Datei ist eine XML-Datei[24], die sowohl den Ablauf der Generatoren als auch den der Komponenten der Analysephase bestimmt. Auch in der neuen KOMPILIERER UMGEBUNG wird diese Datei zur Steuerung des Ablaufes ausgelesen.

Eine solche Datei kann mehrere Konfigurationen enthalten. Welche der Konfigurationen tatsächlich angewendet werden soll, kann in der KOMPILIERER UMGEBUNG in den EINSTELLUNGSSEITEN innerhalb des eclipse-Workbench leicht eingestellt werden. Der Konfigurator, wie in Abschnitt 2.5.2 beschrieben, erstellt unter Umständen mehrere Konfigurationen für ein Ziel-System. Eine grafische Auswahl verschafft dem Benutzer den Überblick über die vergleichsweise komplex aufgebauten Konfigurationen.

Auch Plugins mit mehr Funktionalität als das Bereitstellen eines Generators werden in diese Konfigurationsdatei aufgenommen, also solche, die zusätzlich Einstellungsdialoge propagieren oder Ähnliches. Diese Konfigurationsdateien bieten eine einfache Möglichkeit, Einstellungen zwischen mehreren Beteiligten zu teilen. Dabei ist es unabhängig, ob die Benutzer die eclipse KOMPILIERER UMGEBUNG benutzen oder nicht. Tatsächlich ist dieses eine der Richtlinien bei der Erstellung der neuen RAHMENWERKS PLUS. Diese Umgebung soll den Ablauf von Generatoren in anderen Umgebungen tolerieren. Eine Verwendung einer gemeinsamen Konfigurationsdatei bietet sich hier also an.

Es kann nicht nur eingetragen werden, welche Komponenten ablaufen sollen, sondern auch Parameter für den Ablauf. Da eine Konfigurationsdatei noch nicht durch einen Wizard oder einen speziellen Editor erstellt wird, kann es bei der Eingabe leicht zu Fehlern kommen.

Eine zukünftige Erweiterung des KOMPILIERER RAHMENWERKS könnte eine Unterstützung der Konfigurationserstellung sein und wäre eine große Hilfe für den Benutzer. Dafür bietet sich eclipse an. Wenn man den *Manifest-Editor* des PDE betrachtet, kommt man zu einem Eindruck, wie ein solcher Editor aussehen könnte.

3.8 Interne Laufzeitbeobachtung des Rahmenwerk Plus

Damit Entwickler, die an dem RAHMENWERK arbeiten oder Generatoren oder andere Komponenten erstellen, nachvollziehen können, was zur Laufzeit geschieht, werden Ausgaben in einer eigens dafür erstellten Ansicht gemacht. Aus vier Kategorien kann gewählt werden, welche Ausgaben von Interesse sind. Die Auswahl geschieht über eine grafische Einstellungsseite. Diese Kategorien sind:

- **Methodenaufruf** – Wenn Methoden aufgerufen werden, wird eine entsprechende Ausgabe gemacht. Dabei können auch Parameterwerte betrachtet werden. Es ist für Kenner des Aufbaus des Plugins möglich, den Ablauf nachzuvollziehen.

[24] Mit der Namenskonvention: `cat.xml` ursprünglich - für Zwecke des RAHMENWERKS PLUS jedoch erweitert: `cat*.xml`

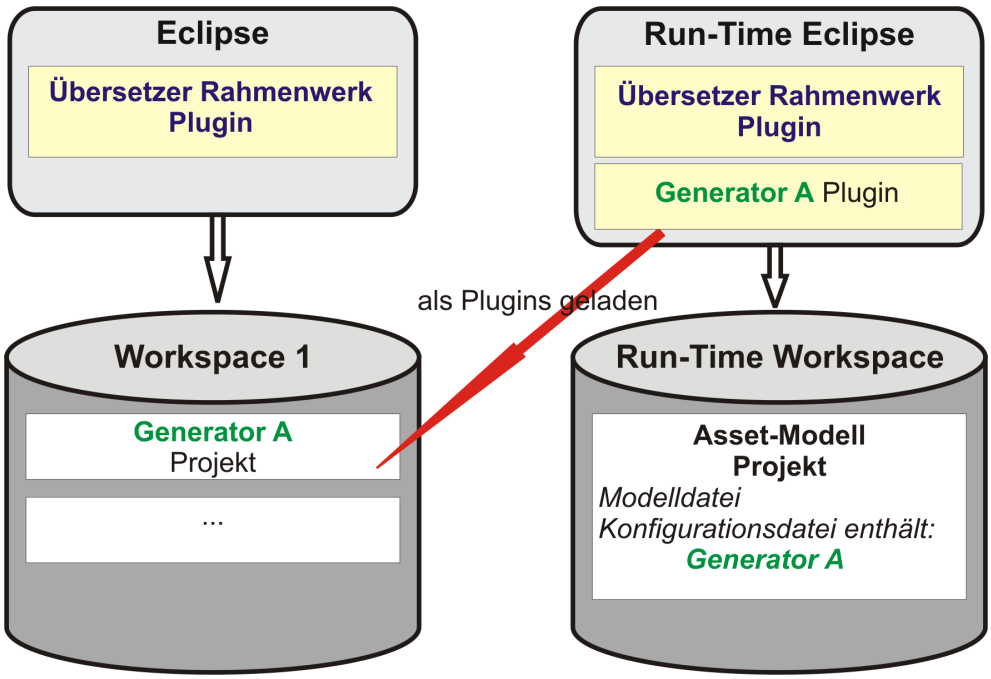


Abbildung 3.11: Der 'zwei Instanzen Plugin-Test'

- **Status** – Wenn der Benutzer über den Status der Systemerzeugung informiert werden möchte, wird dies Kategorie gewählt. Wenn einem Projekt keine Konfigurationen zugewiesen sind, wird das dem Benutzer beispielsweise angezeigt.
- **Ereignisse** – Vieles in eclipse basiert auf Ereignissen. Für Ereignisse, die innerhalb des RAHMENWERKS versendet oder wenn Komponenten des Rahmenwerks im Zuge einer Benachrichtigung angesprochen werden, wird eine Ausgabe erstellt, um den Ablauf nachvollziehen zu können.
- **Listen** – In dem Ablauf des RAHMENWERKS werden an verschiedenen Stellen Listen von Objekten erstellt. Um Fragen nachgehen zu können, ob ein Objekt mit aufgelistet ist oder nicht, werden diese Listen ausgegeben.

3.9 Komponenten-Einbindung zu Testzwecken

Erstellt ein Komponenten-Entwickler beispielsweise einen Generator, so legt er im Workspace eclipses ein Plugin-Projekt an. Das Projekt befindet sich in dem Bereich, auf den eclipse zugreift, im Workspace, ist jedoch noch nicht Teil von eclipse selbst. Damit aus einem Plugin-**Projekt** ein **Plugin** wird, muss man es in eclipse einbinden. Üblicherweise erstellt man dazu ein sogenanntes *feature* und lädt dieses dann mittels des Updatemechanismus. Das ist notwendig, wenn man eine Komponente publizieren möchte. Wenn man die Komponente jedoch in der laufenden Entwicklung testen möchte, ist das eine viel zu aufwendige

Vorgehensweise. Eclipse hat einen anderen Mechanismus vorgesehen. In der Abb. 3.11 ist dieser veranschaulicht.

Für dieses Vorgehen wird eine zweite Instanz von eclipse gestartet. Die erste Instanz eclipses referenziert den Workspace mit dem Generator-Projekt, in der Abb.: Generator A Projekt. Die zweite Instanz wird automatisch mit dem Projekt als Plugin geladen. Dieses zweite eclipse greift auf eine eigene Arbeitsumgebung zu.

Um eine Komponente, diesen Generator A, in einen Übersetzungsvorgang einzubinden, muss ein Modell-Projekt mit entsprechender Konfigurationsdatei im zweiten Workspace erstellt werden. Dort kann man nun den Generator A testen, als ob er mit dem Updatemechanismus installiert worden wäre.

Eine Änderung an dem Generator-Projekt erfordert zumeist keinen Neustart der zweiten eclipse-Instanz. Der Quellkode des Generator A wird zur Laufzeit des zweiten eclipse im Generator A Plugin durch entsprechenden Bytekode ersetzt. Man sagt dazu auch '*hot code replace*'. Damit wird das Erstellen eines Generators in der neuen Umgebung auch für Komponenten-Entwickler attraktiv.

In der zweiten eclipse-Instanz wird ein Modell mit dem Generator A übersetzt, und es entsteht beispielsweise Java-Quellkode. Dieser Kode kann dann anschließend automatisch von der eclipse Komponente JDT in Java-Bytekode umgesetzt werden. Ein Starten von Java-Klassen wird von eclipse auch unterstützt.

Die Forderung nach einer einheitlichen Entwicklungs-, Test- und Ablaufumgebung, wie in Abschnitt 2.6.2 beschrieben, wird somit erfüllt.

KAPITEL 4

Implementation

In diesem Kapitel wird erläutert wie die im Design aufgezeigten Komponenten implementiert werden. Die Umsetzung eines Konzepts und somit die entstehende Architektur sind hier sehr oft nah an der Implementierung zu erläutern, da eclipse den Aufbau durch seine Schnittstellen vorgibt.

Im Abschnitt 4.1 wird das CompilerCore Plugin auf Klassen- und Paket-Ebene vorgestellt. Es wird anhand der Klassen und zusätzlicher Dateien der Aufbau des Plugins beschrieben. Dabei werden Gruppen von Klassen und Dateien herausgenommen und deren Zuständigkeit betrachtet.

Werden Wörter in `Courier-Schrift` dargestellt, handelt es sich in diesem Kapitel um Klassen, Pakete, Methoden oder Bezeichner. Auch Ordner und Dateien werden in `Courier` dargestellt.

Das vorgestellte Plugin wird bereits von Komponenten-Entwicklern genutzt und soll auch in Zukunft gepflegt und erweitert werden. Der Ausblick (Abschnitt 6.3) zeigt mögliche Erweiterungen. Das folgende Kapitel *Implementation* dient auch als Dokumentation für Komponenten-Entwickler und solche, die am Rahmenwerk arbeiten.

4.1 Organisation des Rahmenwerk Plus im Überblick

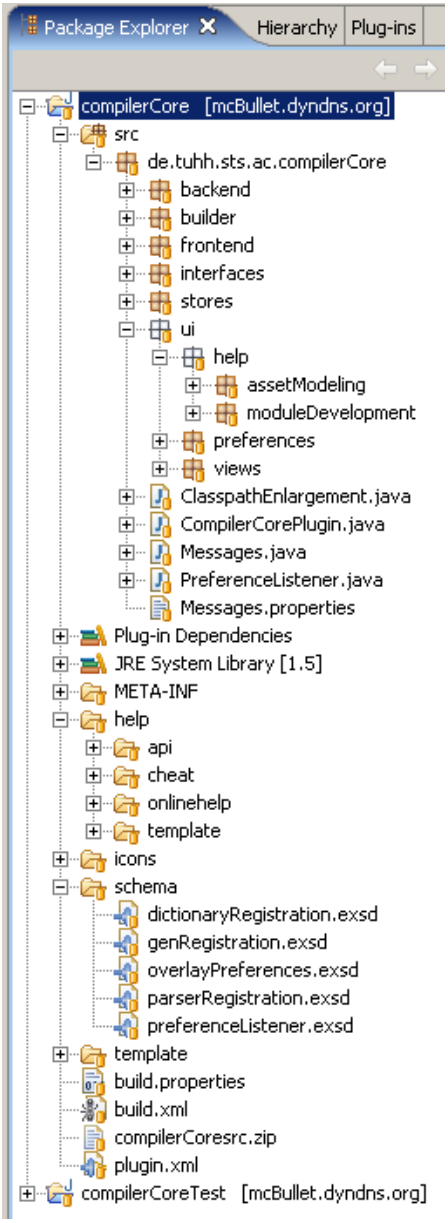


Abb 4.1: Übersicht über das Rahmenwerk-Plugin

In der Abbildung 4.1 ist das Plugin `compilerCore` zu sehen. Das Plugin beinhaltet die Kernfunktionalität des Rahmenwerks, stellt also die Ablaufsteuerung für Generatoren und andere Komponenten dar. Bevor auf die einzelnen Pakete eingegangen werden kann, ist erst einmal Generelles zum Aufbau eines Plugin in eclipse darzulegen. Insbesondere wird auf dieses Plugin eingegangen, es werden nicht alle möglichen Implementationsformen erläutert.

Dieses Plugin wurde mit dem PDE - Wizard erstellt. Automatisch wird dem Projekt eine Java-Natur zugewiesen; dieses wird durch ein kleines, blaues, hochgestelltes „J“ links neben dem Projektnamen angezeigt. Automatisch erzeugt ist der Ordner `src`, darin befindet sich der Quellkode in Form von Java-Klassen. Der Ausgabeordner `bin` ist in dieser Ansicht nicht zu sehen, wohl aber vorhanden. Genauso verhält es sich auch mit anderen Dateien, die nicht sichtbar sind, um eine gute Übersicht gewährleisten zu können. Zu nennen sind hier die Bibliotheken, die in die Distribution des Plugins eingebunden sind. Sobald die Bibliotheken in den *Plug-in Dependencies* erscheinen, werden sie an ihrem Aufenthaltsort unsichtbar. Ein Beispiel dieser Bibliotheken ist die, welche die Schnittstellen beinhaltet, gemäß denen Generatoren implementiert werden.

Es ist auch die Datei `.options` zu erwähnen. Die in der Datei zu definierenden Einstellmöglichkeiten können genutzt werden, um dem Plugin beim Start Parameter mit zu übergeben.

In der Abb. kann man noch Ordner und Klassen sehen, die mit dem Plugin ausgeliefert werden. Sie beinhalten Ressourcen wie Bilder, HTML-Seiten, Vorlagen und anderes. Dazu wird es im Folgenden noch Erklärungen geben.

4.1.1 Zeichenerklärung der Übersicht

Symbole sind hier neben den Ressource-Namen aussagekräftig. Es folgt nun eine Auflistung der Symbole, wie sie in Abb. 4.1 zu sehen sind.


Ein Projekt mit Java Quellkode. Genauer gesagt muss kein Kode enthalten sein. Dieses Symbol erscheint, sobald dem Projekt eine Java-Natur zugewiesen wurde, mehr dazu im Abschnitt 3.5.1.


Ein Ordner, entspricht dem in einem Dateisystem. Tatsächlich ist solch ein Ordner im Workspace ein „normaler“ Ordner plus einiger Metadaten, die hier jedoch nicht extra angezeigt werden.


Hierbei handelt es sich um eine Datei, die mit dem Texteditor als Standardeinstellung geöffnet wird. Sollte ein anderer Editor für das Öffnen der Datei zuständig sein, wird dessen Symbol angezeigt.


Ein Java-Paket, hier in hierarchischer Ansicht. Das Paket `backend` befindet sich beispielsweise in dem Paket `de.tuhh.sts.ac.compilerCore`.


Eine Java-Quellkode-Datei, diese Dateien enthalten Kode in Java und sind in den entsprechenden Paketen eingeordnet.


Manifestdatei, Verwendung für *plugin.xml* und `MANIFEST.MF`. Diese Dateien enthalten Beschreibungen über das Plugin.


Bibliotheken, die unter diesen Einträgen aufgelisteten Bibliotheken werden zum Start des Plugins referenziert.


Definitionen eines *extension-point schema*. Diese Beschreibungen dienen dazu, genau festzulegen, wie sich andere Plugins an dem Ablauf von diesem Plugin beteiligen können. Üblicherweise tragen die beschreibenden Dateien die Identifikation des Erweiterungspunktes im Namen. Die Datei `genRegistration.exsd` enthält zum Beispiel Informationen darüber, wie sich ein Generator bei dem compilerCore-Plugin bekannt zu machen hat, welche Parameter angegeben werden müssen und wie diese auszusehen haben.


Hiebei handelt es sich um ein Zeichen, das fast allen Dateien überlagert ist[1]. Dieses Zeichen gibt darüber Auskunft, ob die Datei mit einem CVS Repository abgeglichen ist. Das CVS-Plugin definiert dieses Symbol. CVS ist in eclipse integriert und wird zur Versionsverwaltung und Verteilung zwischen Teammitgliedern verwendet.

[1] Man spricht hier von einem *Decorator*.

4.1.2 Die Plugin - Beschreibung

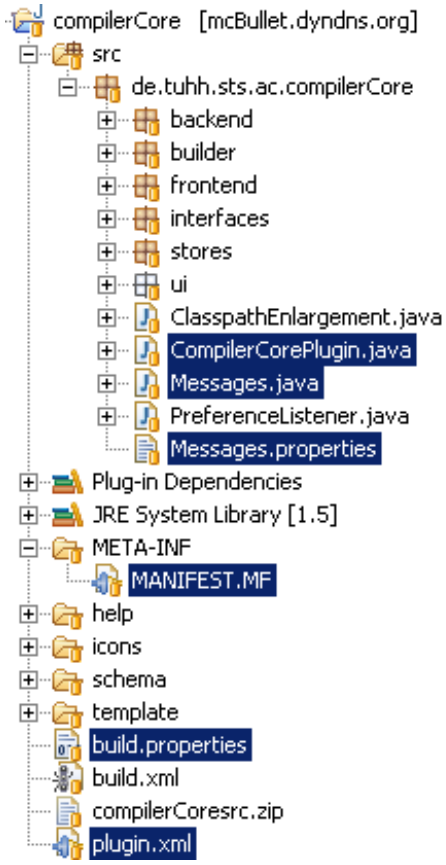


Abb 4.2: Plugin-Beschreibung

Blau hinterlegt kann man in Abb. 4.2 sehen, welche Komponenten für die Beschreibung des Plugins zuständig sind. Die Dateien MANIFEST.MF und plugin.xml werden als Manifest-Dateien bezeichnet. Sie beinhalten Beschreibungen über das Plugin. Dazu gehören Angaben über die Identifikation des Plugins, benötigte andere Plugins, Versionsnummer, zur Laufzeit einzubindende Bibliotheken und Definitionen für Erweiterungen an anderen Plugins. Es werden Erweiterungen (siehe Abschnitt 3.4.1) definiert, die die Funktionalität anderer Plugins erweitern.

In `build.properties` ist gekennzeichnet, welche der hier zu sehenden Komponenten zu dem Plugin gehören und mit diesem ausgeliefert werden sollen.

Die Klasse `CompilerCorePlugin` wird dann geladen, wenn das Plugin aktiviert wird[2]. Sie beinhaltet Methoden für den Zugriff auf Laufzeitinformationen des Plugins. Man kann sowohl plug-inspezifische Einstellungen zur Laufzeit speichern, siehe Abschnitt 3.7, wie auch auf mitgelieferte Dateien des Plugins zugreifen. Diese Klasse ist ein SINGLETON (siehe [2]) und wird immer dann benutzt, wenn eine zentrale Instanz notwendig ist.

Beim Start des Plugins wird von eclipse die Methode `start()` und bei Beendigung die Methode `stop()` aufgerufen. In diesem Fall wird die `stop()`-Methode dazu verwendet, Instanzen, die sich als Nachrichtenempfänger bei Klassen außerhalb des Plugins eingetragen haben, wieder auszutragen. Die `start()`-Methode sorgt für die Initialisierung der dynamischen Pfaderweiterung, mehr dazu Abschnitt 4.4.1.

Die „unsichtbare“ Datei `.options` wurde schon erwähnt. Darin sind die einstellbaren Parameter beschrieben, wie sie der Benutzer in einem Dialog, wie dem in Abb. 4.3 gezeigten, sehen kann. Sind die Einstellungen vorgenommen, werden diese bei der Aktivierung des Plugins einmalig ausgelesen und anderen Klassen bereitgestellt.

Andere Klassen, auch Komponenten dritter, können diese Einstellungen verwenden, um z.B. zu entscheiden, ob eine Ausgabe an den Benutzer gemacht

[2] Auch das wird in der Manifestdatei beschrieben.

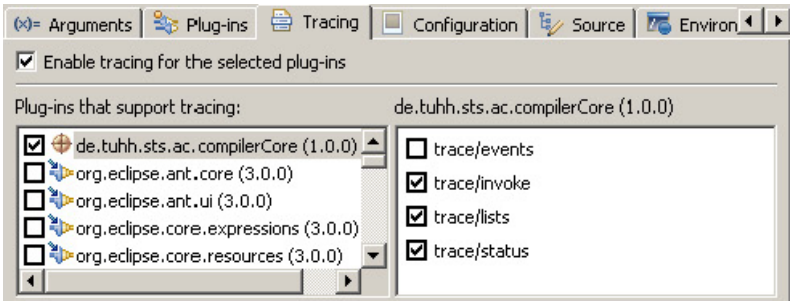


Abbildung 4.3: Starteinstellung Tracing

werden soll oder nicht. So kann dann z.B. nachvollzogen werden, welche Methoden angesprochen worden sind, welche Ereignisse ausgelöst wurden, was Listen enthalten oder welche Zustände erreicht sind.

Diese Einstellungen können über einen grafischen Einstellungsdialog zur Laufzeit geändert werden.

Die Klasse `CompilerCorePlugin` kann auch für die Bearbeitung von Ausnahmen angesprochen werden. Dabei sind zwei Arten von Benutzerinteraktion vorgesehen. Ist der Fehler der Art, dass der Benutzer nicht weiter informiert werden soll, es sei denn, er schaut explizit nach, so wird eine statische Methode aufgerufen. Die Fehlermeldung wird in den Fehler-Speicher des Workbench geschrieben und dem Benutzer via der eclipse-eigenen Ansicht *Error Log* präsentiert. Das hat den Vorteil, dass der Benutzer durch Fehler nicht penetriert wird. Der Nachteil ist ebenso offensichtlich; Fehler können vom Benutzer übersehen werden. Dieser fährt in seiner Arbeit fort, als ob alles in Ordnung wäre. In manchen Situationen ist das unzureichend, weil von dem Benutzer korrigierende Aktionen vorgenommen werden sollten. In diesem Fall kann der Benutzer mittels eines Dialogs über den Fehler informiert werden. Dazu bietet die `CompilerCorePlugin`-Klasse die Methode `handleInformUser()` an. Sie sorgt für das Öffnen eines Dialogfensters mit entsprechendem Inhalt. Wie die Methode so etwas macht, kann man im Quellkodeausschnitt B.1 im Anhang sehen.

Eine Schwierigkeit ist hier zu überwinden. Eclipse sorgt dafür, dass Dialoge nur von dem Workbench-Thread aufgerufen werden können. Von außerhalb des Thread kann man Kode in den laufenden Workbench-Thread einfügen und zur Ausführung bringen. Man kann diese Vorgehensweise im Quellkodeausschnitt im Anhang B.1 ab Zeile 16 sehen. Meistens ist solch eine komplizierte Vorgehensweise jedoch nicht nötig. Die meisten Dialoge und Wizards werden als Erweiterungen definiert und somit von eclipse aus gestartet und laufen dann im richtigen Thread ab.

In den Dateien `Messages.java` bzw. `Messages.properties` wird festgelegt, welche Zeichenketten in den Benutzeransichten zu sehen sind. Mit statischen Methoden der Klasse `Messages` kann auf diese mittels Schlüssel und Angabe von Formatparametern zugegriffen werden.

4.1.3 Dynamische Komponenteneinbindung

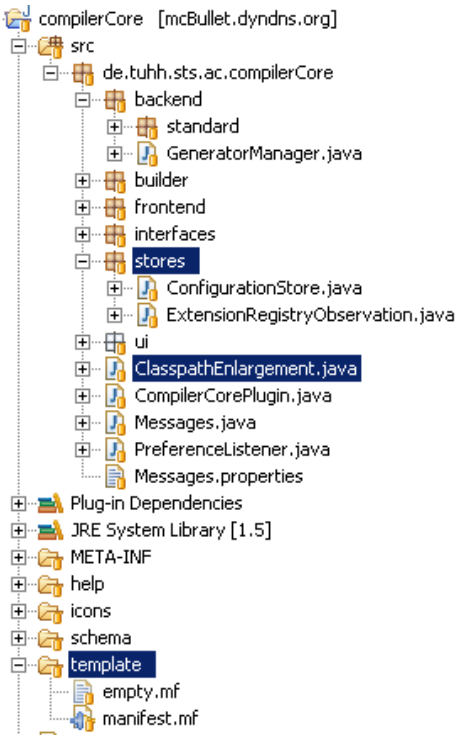


Abb 4.4: Zugriffsklassen für die Konfiguration

Drei Arten von Einstellungen sind für den Ablauf der Übersetzung entscheidend.

Es ist wichtig, welche Komponenten sich in welchem Ausmaß an dem Kompiliervorgang beteiligen möchten. Eine von eclipse verwaltete *Extension-Registry* beinhaltet Erweiterungsdefinitionen aller Plugins. Wenn es also darum geht, für einen Erweiterungspunkt Plugins zu finden, die diesen nutzen, wird die Registry angesprochen. Die Klasse `ExtensionRegistryObservation` hält Kopien von solchen Einträgen und überwacht deren Veränderungen[3].

Klassen, die für die Übersetzung wichtig sind, können u. sollten durch Plugins definiert werden. Falls dies nicht der Fall ist, gibt es auch noch die Möglichkeit, angeforderte Klassen über eine Erweiterung des Pfades zugänglich zu machen. Dafür ist die Klasse `ClasspathEnlargement` zuständig.

Auf die dynamische Erweiterung des Pfades wird im Abschnitt 4.4.1 detailliert eingegangen. Dazu ist es notwendig, ein Plugin zu generieren und zu starten. Vorlagen für die dazu notwendige Manifestdatei befinden sich im Ordner `template`.

Der genaue Ablauf der Übersetzung, welche Teile dafür verwendet werden und mit welchen Parametern dies geschieht, wird in Konfigurationsdateien bestimmt. Der Zugriff auf diese Dateien erfolgt über den `ConfigurationStore`. Die Abschnitte 3.7.5 und 4.5 beschäftigen sich mit diesem Vorgang.

Letztendlich kann man auch in der grafischen eclipse-Umgebung Einstellungen machen. Der dazu bereitgestellte umfangreiche Mechanismus wird im Abschnitt separat 4.1.7 erfasst.

4.1.4 Hilfe und Benutzerführung

Im Bereich Hilfe und Benutzerführung sind drei Arten der Benutzerführung zu unterscheiden. Es handelt sich hier um vollautomatische, über halbautomatische bis hin zu der rein manuelle Erschaffung von Ressourcen.

[3] Hier treten Veränderungen nur auf wenn ein neues Plugin installiert wird.

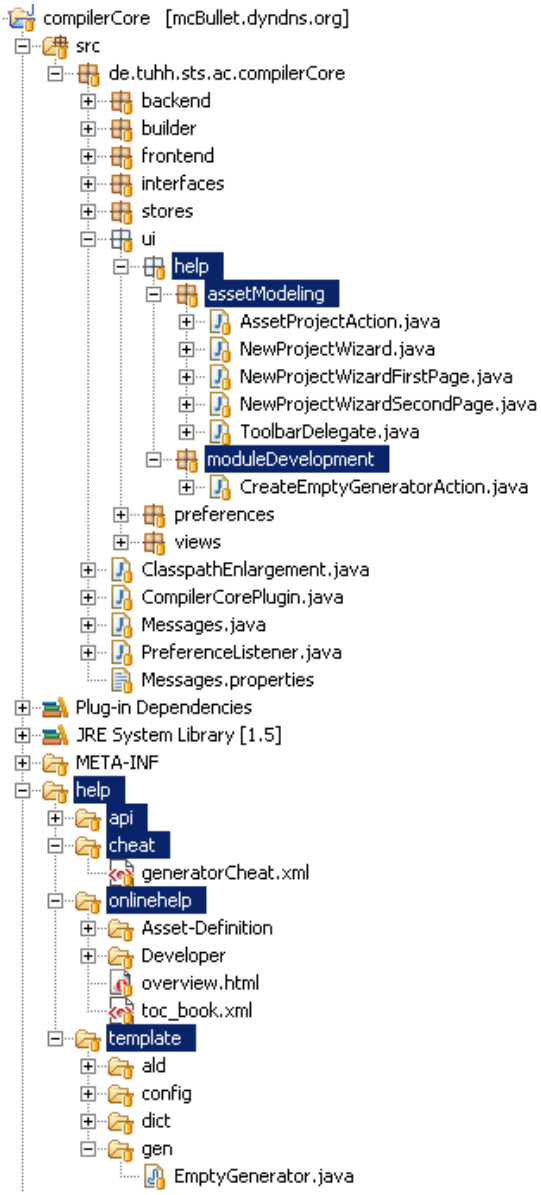


Abb 4.5: Wizard, Cheat Sheet, Hilfe

Die Hilfe ist eine Dokumentation im klassischen Sinne. Sie besteht aus Text und Grafiken. Zur Anzeige stellt eclipse einen Browser zur Verfügung. Es werden, wie bei Browsern (siehe [26]) allg. üblich, `html`- oder `pdf`-Dokumente angezeigt. Die Strukturierung der Inhalte, sowie die Seiten sind in dem Ordner `help/onlinehelp` enthalten. Für diese Art der Hilfe sind keine Aktionen vorgesehen, die irgendetwas automatisch für den Benutzer tun.

Anders dagegen *Cheat-Sheets*: Es handelt sich hierbei um eine Art der interaktiven und parameterfreien Anleitung. Der Anwender kann sich Schritt für Schritt durch einen Arbeitsgang führen lassen. Wird einerseits erwartet, dass der Benutzer, den Beschreibungen folgend, Aktionen manuell ausführt, ist es auf der anderen Seite möglich eine Aktion vom CompilerCore Plugin ausführen zu lassen. Es kann sich bei diesen Aktionen z.B. um das Öffnen eines beschriebenen Wizards, oder um das Generieren einer gesamten Projektstruktur handeln. Vorteil dieser Art der Benutzerführung ist, dass der Benutzer alle Teilschritte nachvollziehen und später auch selbständig ohne Hilfe des Cheat Sheet ausführen kann. Der Benutzer hat bei diesen Anleitungen leider keine Möglichkeit, Parameter für die Ausführung von Aktionen einzugeben.

Die Datei `generatorCheat.xml` enthält eine Anleitung für Generator-Komponenten Entwickler, siehe Abschnitt 2.5.4. Die in Java implementierten Aktionen dieser Anleitung sind nicht Bestandteil der Anleitung, sondern in der Klasse `CreateEmptyGeneratorAction.java` von überall her aufrufbar. Der Ordner `help/template` enthält notwendige Dateien für die Generierung eines Generator-Projektes. Aktionen und Cheat Sheets werden im Kapitel 5 und im Abschnitt 5.3.1 behandelt.

Bei einem Wizard wird der Benutzer durch einen komplexen Vorgang ge-

führt. Dabei können Eingaben getätigt werden, die für die Vorgehensweise der Erzeugungslogik im Hintergrund des Wizards notwendig sind. Für Asset-Modell Ersteller, siehe Abschnitt 2.5.1, wird ein Wizard zur Erstellung eines neuen Projektes angeboten. Die Erklärung dessen, was dabei getan wird, steht *nicht* im Vordergrund. Dafür muss dann die Hilfe hinzugezogen werden. Der Wizard befindet sich im Paket `help.assetModelling` und benötigt auch Vorlagen aus dem Ordner `help/template`.

Der Benutzerführung ist das Kapitel 5 gewidmet.

4.1.5 Kompilieren von Projekten

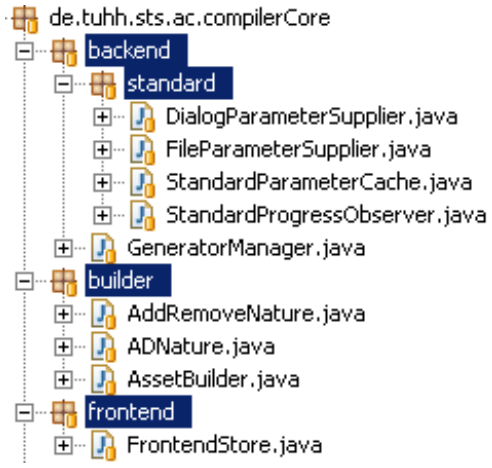


Abb 4.6: Klassen für den Kompilier-Prozess

Die zentrale Klasse für den Kompilier-Prozess ist `AssetBuilder`. Was ein Builder in eclipse ist und unter welchen Umständen er angesprochen wird, kann im Abschnitt 3.5 ausführlich nachgelesen werden. Dort beschrieben ist ebenfalls die Aufgabe einer Natur (`ADNature.java`), die eng mit dem Gebrauch von Buildern zusammenhängt.

Im Verlauf des Übersetzungs-Prozesses benötigt `AssetBuilder` für die Analysephase `FrontendStore` und dann `GeneratorManager` für die folgende Synthesephase. Auf den Ablauf der Übersetzung gehen die Abschnitte 2.3 und 4.5 ein. Im Paket `backend.standard` befinden sich Hilfsklassen für den Ablauf von Generatoren. Es handelt sich dabei um Klassen, die verwendet werden, wenn erweiternde Plugins keine eigenen Klassen jener Art liefern. Beispielsweise ist die Klasse mit dem Namen `FileParameterSupplier` ein Parameter-Lieferant, wie in Abschnitt 3.2 beschrieben, der Parameter aus der Konfigurationsdatei der laufenden Konfiguration ausliest. Näheres über diesen Vorgang wird in Abschnitt 4.5.1 ab Seite 75 beschrieben.

4.1.6 Anbieten von Erweiterungspunkten

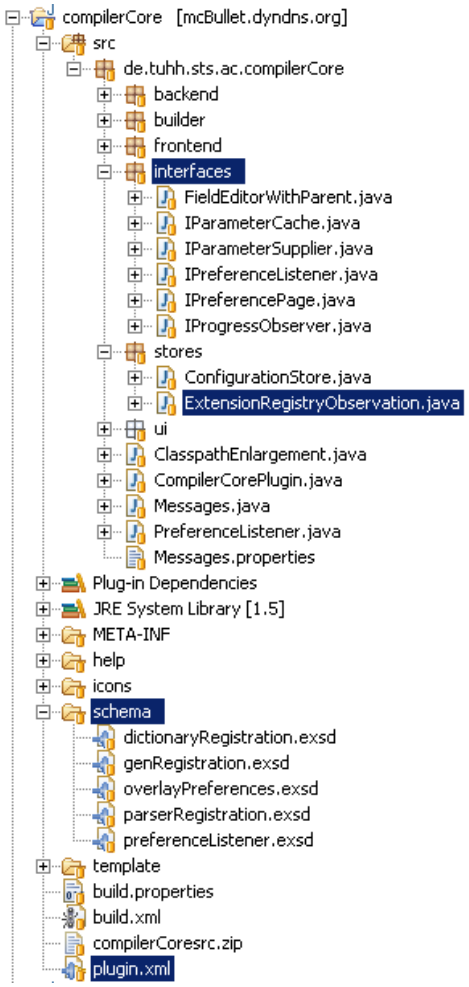


Abb 4.7: Anbieten von Erweiterungspunkten

Plugins können Erweiterungspunkte anbieten. Dazu muss man in der Manifestdatei `plugin.xml` einen *extension-point* mit einer einzigartigen Identifikation definieren. Es wird dann in einer separaten Datei beschrieben, wie *Erweiterungen* genau aussehen müssen. Ein *schema* wird definiert. Die Dateien im Ordner `schema` enthalten solche Beschreibungen. Darin ist auch definiert, welche Schnittstellen von erweiternden Klassen implementiert werden müssen. Die Schnittstellen sind in dem CompilerCore Plugin definiert u. im Paket `interfaces` zu finden.

Nimmt ein Plugin an einer Erweiterung teil, so wird zum Start von eclipse ein weiterer Eintrag in der *Extension-Registry* von eclipse vorgenommen. Auf diese Einträge wird dann über Zugriffsmethoden der Klasse `ExtensionRegistryObservation` zugegriffen. Damit können z.B. Klassen, wie sie in der Erweiterung angegeben sind, instanziert werden. Ein Beispiel für die Instanzierung durch diese Klasse wird in Abschnitt 4.5 beschrieben. Nun ist es wichtig, dass die geladenen Klassen erforderliche Schnittstellen auch implementieren, da ansonsten eine Ausnahme ausgelöst werden würde. Vorteil dieses dynamischen Ladens von Klassen ist, die freie Konfigurations- und Erweiterungsmöglichkeit. Außerdem ist vorgesehen, andere Plugins erst dann zu laden wenn Klassen aus ihnen benötigt werden. Man vergleiche dazu entsprechendes Konzept in Abschnitt 3.6.

4.1.7 Einstellungs-Dialoge

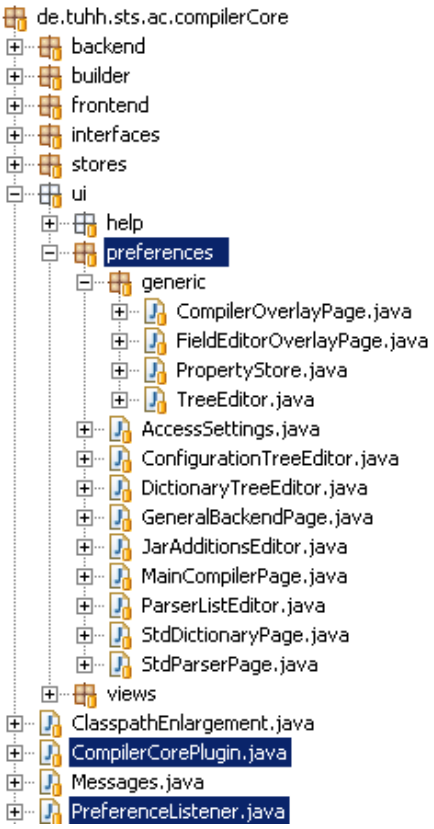


Abb 4.8: Klassen für Einstellungsdialoge

Das CompilerCore Plugin bietet einen bestimmten Service im Zusammenhang mit Einstellungsdialogen an. Diesem Konzept ist der Abschnitt 3.7 und der Implementation dann der Abschnitt 4.6 gewidmet.

Eine Unterteilung in drei Kategorien von Einstellungen wird vorgenommen. Erweiternde Klassen benennen in ihrer *Extension* eine der Zuordnungen: General, Frontend oder Backend. Die Klassen aus dem Paket `preferences` mit der Endung „Page“ nehmen selbst an dem Service teil und repräsentieren für den Benutzer sichtbare Einstellungen.

Das Subpaket *generic* beinhaltet Klassen, die mehrfach benutzt werden und eine generelle Präsentation darstellen. Insbesondere wird im Abschnitt 4.6 auf die Klassen `FieldEditorOverlayPage` u. `PropertyStore` eingegangen.

Änderungen an Einstellungen rufen Benachrichtigungen an alle Beobachter hervor. Alle Interessenten haben sich über einen Erweiterungspunkt zu registrieren. Die Verwaltung der Beobachter wird in der bereits vorgestellten Klasse `CompilerCorePlugin` vorgenommen. Auch das Plugin CompilerCore bildet hier keine Ausnahme. Wie andere Plugins auch, registriert es einen Beobachter: die Klasse `PreferenceListener`.

Über die Klasse `AccessSettings` kann auf die vom Benutzer eingestellten Werte zugegriffen werden.

```
<!-- ================== Nature-Definition ========= -->
<extension point="org.eclipse.core.resources.natures"
   id="assetDefinitionNature"
   name="Nature for Asset Definition">
   <runtime>
         <run class="de.tuhh.sts.ac.compilerCore.builder.
            ADNature"/>
   </runtime>
   <builder id="de.tuhh.sts.ac.assetDefinitionBuilder" />
</extension>
```

Kode-Beispiel 4.1: Beschreibung einer Natur

4.2 Die Natur Asset-Definition-Nature

Builder und Naturen sind im Design im Abschnitt zum eclipse Builder-Konzept 3.5, ab Seite 39 beschrieben.

Es ist für Projekte, die ein Asset-Modell enthalten, notwendig, eine Natur zu definieren:
`de.tuhh.sts.ac.compilerCore.assetDefinitionNature`
(*ADNature*).

Diese Natur wird in der Manifestdatei[4] definiert, siehe Kode-Beispiel 4.1. In der 6. u. 7. Zeile wird definiert, welche Klasse die entsprechende Schnittstelle `IProjectNature` implementiert. Der Aufbau des Manifesteintrags ist von eclipse bestimmt und ermöglicht es eclipse, zur Laufzeit entsprechende Klassen zu laden. Wird eine Natur zu einem Projekt hinzugefügt, können beliebige Verarbeitungslogiken ausgeführt werden. Z.B. kann es zum Einbinden bestimmter Bibliotheken kommen, oder eine neue Verzeichnisstruktur wird erstellt.

Durch die Implementierung der oben genannten Schnittstelle, kann eclipse Methoden aufrufen, die das Verhalten beim Hinzufügen, beziehungsweise beim Entfernen der Natur bestimmen. Die *ADNature* fügt dem, bei dem Aufruf mit übergebenen Projekt, einen Builder hinzu. Somit wird, welches Plugin auch immer die *ADNature* einem Projekt hinzufügt, immer ein Builder eingetragen. Dieses Verhalten macht sich das CompilerCore Plugin auch zu Nutze (siehe Abschnitt 5.2).

Wie schon erwähnt, kann diese Natur als Filter Verwendung finden. In dieser Implementierung wird in dem *popupmenu*[5] ein Eintrag ein- oder ausgeblendet. Der Eintrag, um die *ADNature* zu entfernen, erscheint nur, falls auch eine solche vorliegt, vgl. Kode-Beispiel 4.2. Die Zeilen 5ff zeigen, wie zu einem *popupmenu* ein Eintrag hinzugefügt wird. Im Attribut „objectClass“ (Zeile 6) wird bestimmt für welche Arten von Ressourcen der Eintrag erscheint. Die Zeilen 7 bis 10 definieren einen weiteren Filter und sind folgendermaßen zu interpretieren: nur falls die *ADNature* in dem Projekt als Natur zu finden ist, erscheint der Eintrag in dem Menü. Es scheint nicht sinnvoll, eine Natur zu de-

[4] `plugin.xml`
[5] Das beim Anwählen der rechten Maustaste auf einem Projekt erscheinende Menü.

```
<!-- ========== addADNature / removeADNature ========== -->
<extension point="org.eclipse.ui.popupMenus"
   id="nature.pop"
   name="Natures">
   <objectContribution id="de.tuhh.sts.ac.removeNature"
      objectClass="org.eclipse.core.resources.IProject">
      <filter name="projectNature"
         value="de.tuhh.sts.ac.compilerCore.
             assetDefinitionNature">
      </filter>
      ...
```

Kode-Beispiel 4.2: Entfernen der Natur Asset-Definition-Nature

finieren, nur um sie wieder entfernen zu können, deswegen soll hier ein zweites Beispiel aufgeführt werden. Projektspezifische Einstellungen für das Übersetzer Rahmenwerk können nur getätigt werden, falls es sich um ein Projekt mit einer *ADNature* handelt.

4.3 Analysephase und Fehler-Marker

Die Analyse wird von drei Komponenten erledigt. Die Verwaltung dieser Komponenten übernimmt die Klasse `FrontendStore`, die Instanzen für jedes Projekt vorhält.

Die ersten beiden Komponenten sind der Scanner und der Parser. Sie werden zusammen eingebunden und übernehmen die Aufgabe der lexikalischen und syntaktischen Analyse. Es wird hier nicht davon ausgegangen, dass Programmierer oder Benutzer diese Komponenten auf eigene Art erweitern und einbinden wollen. Für jedes Projekt kann aus einem Einstellungsdialog gewählt werden, welche Scanner/Parser Komponente man wählen möchte. Dazu werden alle verfügbaren Plugins angezeigt, die eine Erweiterung dazu definieren. Ein Standard-Paar wird mit dem Rahmenwerk Plugin installiert.

Die dritte Komponente besteht aus `Dictionary` Klassen. Diese werden dazu verwandt, Referenzen aus personalisierten Modellen aufzulösen. Dieses Auflösen kann unter Umständen individuell geschehen. Es kann sein, dass eine Datenbank dazu angesprochen wird oder auch im lokalen Dateisystem Modell-Dateien durchsucht werden. Diese `Dictionary` Elemente, von denen es auch mehrere für einen Durchlauf geben kann, können in den Konfigurationsdateien definiert und konfiguriert werden. Es gibt auch einen Erweiterungspunkt für *Dictionary Plugins*. Was für die Einbindung und Konfiguration von Generatoren gilt[6], ist auch an dieser Stelle möglich. Im Gegensatz zu der Vorgehensweise des `GeneratorManager` in der Synthesephase ist der `FrontendStore` nicht eindeutig bezüglich einer Konfiguration, sondern bezüglich eines Projektes. Vorstellbar ist somit auch ein *Dictionary Plugin*, das eine grafische Einstel-

[6]Instanzierung im Abschnitt 4.5 beschrieben.

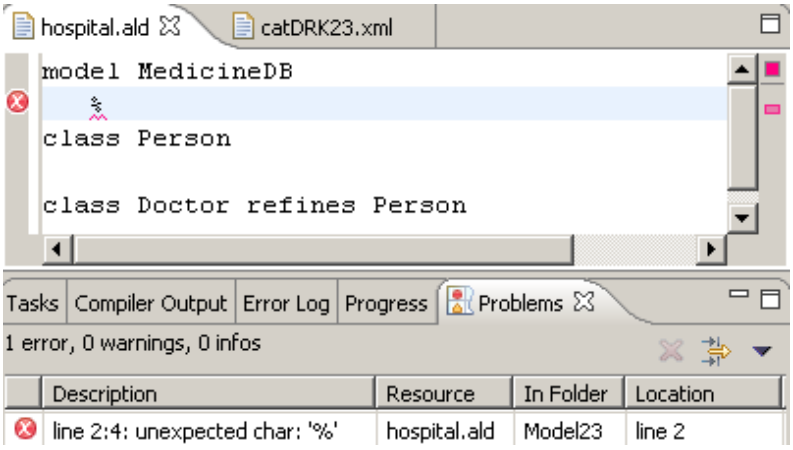


Abbildung 4.9: Fehler-Marker im Asset-Modell

lungsseite propagiert.

Scheitert die Analyse aufgrund eines fehlerhaften Modells, wird ein Marker erzeugt. Wie sich das dem Benutzer darstellt, ist in Abb. 4.9 zu sehen.

Hier wird nun beschrieben, wie solch eine Erweiterung des eclipse Marker-Mechanismus zu bewerkstelligen ist. Der Design-Abschnitt 3.5.6 ab Seite 45 legt dar, wie sich Marker im Allgemeinen dem Benutzer zeigen.

Es können eigene Marker definiert werden. Man vergleiche hierzu Kode-Beispiel 4.3. Eigene Marker können auch mit eigenen Attribut-Wert-Paaren versehen werden. Größter Vorteil eines selbst definierten Markers liegt in der Möglichkeit, nach diesen Markern suchen zu können, bzw. bestimmte Marker ein- oder auszublenden. Dieses kann aus den in eclipse enthaltenen Ansichten heraus geschehen, wie in Abb. 4.9 zu sehen ist.

Zur Erzeugung eines Markers wird die Ressource, für die ein Marker erstellt werden soll, herangezogen[7]. Ressourcen haben eine Methode, um Marker zu erzeugen. Dabei wird der Typ der Marker als Parameter übergeben. Wie sich der Marker über Sitzungen hinaus verhält, wird über seine definierten Eigenschaften, wie im Kode-Beispiel 4.3 gezeigt, definiert. Ohne den in Zeile 6 definierten Zusatz würde der Marker bei Beendigung von eclipse nicht persistent gemacht werden. Solch eine Speicherung und Wiederherstellung übernimmt eclipse dann jedoch selbstständig, ohne explizite Programmierung im Plugin.

4.4 Dynamischen Pfaderweiterung in eclipse

Die Manifestdatei[8], Kode-Beispiel 4.4, ist von der Kompilierer Umgebung generiert und wird später noch genauer betrachtet. Da die Manifestdatei der Kompilierer Umgebung selbst viel komplexer ist, wird obige an ihrer statt

[7] Auch die nur im Workspace von eclipse vorhandene Ressource: `workspace root` kann Marker haben

[8] Doppelverwendung der Begrifflichkeit von Seiten der eclipse-Architekten seit Version 3.0 für `plugin.xml` und `MANIFEST.MF`

```
<!-- ========== Marker ========== -->
<extension point="org.eclipse.core.resources.markers"
   id="cocomamarker"
   name="Problem during Asset Definition Comilation">
   <super type="org.eclipse.core.resources.problemmarker"/>
   <persistent value="true"/>
</extension>
```

Kode-Beispiel 4.3: Definition eines Markers

betrachtet. Hier wird gezeigt, wie Bibliotheken eingebunden und wie Pakete anderen Plugins zugänglich gemacht werden. In den Zeilen 8ff werden die Bibliothek und die Pakete, die andere Plugins sehen können, beschrieben. Enthält eine Bibliothek noch andere Pakete, so können diese zwar von dem einbindenden Plugin benutzt werden, sie sind jedoch für andere Plugins unsichtbar.

4.4.1 Generierung von Plugins zur Laufzeit

Seit der eclipse-Version 3.0 wird das Laden von Klassen anders als zuvor organisiert. Ein neuer Kern (siehe [11]) sorgt für ein dynamisches Auflösen von Pfaden. Basis dieses Mechanismus sind sogenannte `Bundle`-Objekte. In einem Bundle sind Informationen enthalten wie solche über Abhängigkeiten zwischen diesen oder auch Paket-Publikationen. Ein Plugin definiert ein Bundle durch eine Bundle-Manifest-Datei. Manifestdateien von Plugins werden automatisch zur Startzeit von eclipse ausgelesen. Es sind also Informationen über das Plugin bei eclipse bekannt, bevor das Plugin geladen ist (wie bei dem Entwurfsmuster Proxy, siehe [2]). Eingetragene Pakete werden von einem Bundle als bekannt angesehen. Soll nun eine Klasse aus einem Paket geladen werden, wird das Bundle entsprechend auf die angegebene Bibliothek verweisen.

Bundles können auch unabhängig von Plugins, wie sie bereits bekannt sind, installiert werden. Genau das macht sich die Kompilierer Umgebung zu Nutze. Es ist schon erwähnt worden, dass die Umgebung ein Plugin für Generatoren dynamisch generieren und einbinden kann. Das ist, um genau zu sein, nicht ganz richtig. Ein Plugin im engeren Sinne, wie es automatisch in der Startphase von eclipse analysiert wird, wird nicht erzeugt, sondern nur das, was notwendig ist, ein Bundle zu initialisieren. Dieses Bundle wird dann zur Laufzeit erstellt und in den Zustand „aktiv“ überführt. Somit steht das Bundle für das Auflösen von Pfaden zur Verfügung. Solch ein Bundle verhält sich dann wie das eines Plugins.

Dazu muss eine Manifestdatei erzeugt werden. Kode-Beispiel 4.4 ist ein Beispiel dafür. Diese Datei wurde von der Kompilierer Umgebung generiert, was im folgenden erklärt wird.

In den Zeilen 2 bis 7 wird beschrieben, wie das zugehörige Bundle heißt, welche Version davon vorliegt, welche Abhängigkeiten zu anderen Bundles bestehen und wer das Bundle zur Verfügung stellt (MF = **M**athias **F**reier).

Die Einbindung und die Verwendung einer neuen Bibliothek wird erklärt.

```
Manifest-Version: 1.0
Bundle-Name: CompilerCore Additional Jars Plug-in
Bundle-SymbolicName: de.tuhh.sts.ac.compilerCore.
    additionalJars; singleton=true
Bundle-Version: 1.0.0
Bundle-Vendor: MF
Bundle-Localization: plugin
Require-Bundle: de.tuhh.sts.ac.compilerCore
Bundle-Classpath: noPluginGenerator.jar
Provide-Package: de.tuhh.sts.ac.noPlugin
Export-Package: de.tuhh.sts.ac.noPlugin
```

Kode-Beispiel 4.4: Manifestdatei für dynamischen Import von Bibliotheken

Zuerst muss der Benutzer in einem Einstellungsdialog eingeben, an welchem Ort sich die gewünschte Bibliothek befindet[9]. Die Bibliothek – hier in dem Kode-Beispiel `noPluginGenerator.jar` – befindet sich im Allgemeinen in einem beliebigen Ordner. Der Einstellungsdialog wird geschlossen und eine Änderung der Einstellungen wird propagiert. Ein Beobachter (nach [2]) aus dem CompilerCore-Plugin liest nun die Einträge aus und überprüft das Vorhandensein der Bibliotheken. In dem Ordner, der die Bibliotheken enthält, wird ein weiterer Ordner: `META-INF` und darin eine Datei `Manifest.MF`, falls noch nicht vorhanden, erstellt. Die Bezeichnungen `META-INF` und `Manifest.MF` sind durch die OSGI-Spezifikation bestimmt. In der Manifestdatei sind Verweise auf die Bibliotheken und Informationen über deren Inhalt enthalten. Dazu muss der Inhalt der Bibliotheken zur Laufzeit zuvor analysiert werden.

Es tritt eine mögliche Fehlersituation auf. Erstellt man ein eclipse Plugin nach der OSGI-Spezifikation, beinhaltet das entsprechende Projekt auch einen Ordner `META-INF` und darin die Manifestdatei. Würde die Manifestdatei, im Zuge der Einbindung einer Bibliothek, überschrieben werden, wäre das Plugin in seiner ursprünglichen Form unbrauchbar.

An dieser Stelle ist eine Entscheidung zu treffen, wie dieses Fehlverhalten umgangen werden kann. Eine in der Implementation aufwendige Möglichkeit ist es, die Manifestdatei zu analysieren und entsprechend zu ergänzen.

Die von mir gewählte Vorgehensweise vereinfacht die Implementation. Der oben genannte Fehlerfall wird dadurch ausgeschlossen, dass die Bibliothek in ein automatisch angelegtes Projekt kopiert wird. Das Projekt hat den festen Namen `catImports`. Für den Benutzer hat diese Vorgehensweise den Nachteil, dass sich ein weiteres Projekt im Workspace befindet. Er gewinnt dadurch jedoch eine bessere Übersicht über die Bibliotheken.

In der derzeitigen Implementierung ist es nicht möglich, nur einen Teil der Pakete zugänglich zu machen, es sei denn, man manipuliert nachträglich die Manifestdatei und startet eclipse neu.

Der Ort des Ordners (`catImports`), der den Ordner `META-INF` enthält, ist dabei der eindeutige Identifikator für die Aktivierung des Bundles. Eine mehr-

[9] Mehr dazu im Abschnitt 3.7.

malige Aktivierung eines Bundles hat keinerlei weitere Auswirkungen. Damit die Bundles auch vor etwaigen Einstellungsänderungen vorhanden sind, werden sie zur Startzeit des KOMPILIERER PLUGINS mit Hilfe der in einer früheren Sitzung generierten Manifestdatei im Projekt *catImports* initialisiert.

Im Design-Abschnitt 3.6 wird beschrieben, wie eine Klasse aus einem Plugin geladen wird. Sollen Klassen aus den neu hinzugefügten Bibliotheken geladen werden, ist das Vorgehen äquivalent.

Bundles werden immer in einem `BundleContext` (Kontext) erzeugt; das neue Bundle (mit den neuen Bibliotheken) in dem Kontext des COMPILER-CORE PLUGINS. Falls der Pfad im übergeordneten Classloader nicht aufgelöst werden kann, werden alle Plugins im Kontext des COMPILERCORE PLUGINS überprüft.

In der Manifestdatei des COMPILERCORE PLUGINS muss angegeben sein, dass das dynamische Laden von Paketen erlaubt ist. Im benachbarten Bundle wird angegeben, welche Pakete zur Verfügung gestellt werden. Wie im Kode-Beispiel 4.4 ab Zeile 9 zu sehen ist, gibt dieses Bundle die zur Verfügung gestellten Pakete an. Kann der Pfad eindeutig aufgelöst werden, wird der Classloader des entsprechenden Bundles dazu benutzt, die Klasse zu instanzieren.

4.4.2 Dokumentation: Ärgernis und großes Lob

Es wurde erwähnt, dass der Kern eclipses durch das OSGI-Rahmenwerk ersetzt wurde. Verlässliche Dokumentation zu der neuen Komponentensteuerung ist leider nur ungenügend verfügbar. Zum Zeitpunkt des Verfassens dieser Arbeit[10] ist Literatur nur zu dem älteren Kern eclipses veröffentlicht.

Zumeist läuft die Recherche darauf hinaus, dass man die OSGI-Spezifikation [11] lesen muss.

Die eclipse-Foren sind jedoch eine große Hilfe. Ich bin mir darüber bewusst, dass ein Forum nicht als ernst zu nehmende Referenz angegeben werden kann, da dort im Prinzip jeder einen Beitrag leisten kann. Dennoch sind die „eclipse newsgroups“ vorbildlich. Viele Fragen werden von Mitgliedern von IBM und OTI persönlich beantwortet, die auch als Autoren von Büchern der *eclipse-series* in Erscheinung treten. Man merkt, dass die großen Firmen hinter eclipse stark daran interessiert sind, die Benutzergemeinde nicht „im Regen stehen zu lassen“. Ein großes Lob an dieser Stelle an [13].

4.5 Instanzierung und Haltung der Synthese-Komponenten

Im Design in den Abschnitten 3.1 und 3.3 wird bezüglich der Instanzierung und Lebensdauer der Synthesekomponenten das Konzept vorgestellt. Anschließend wird auf die Implementierung eingegangen.

[10] Herbst 2004

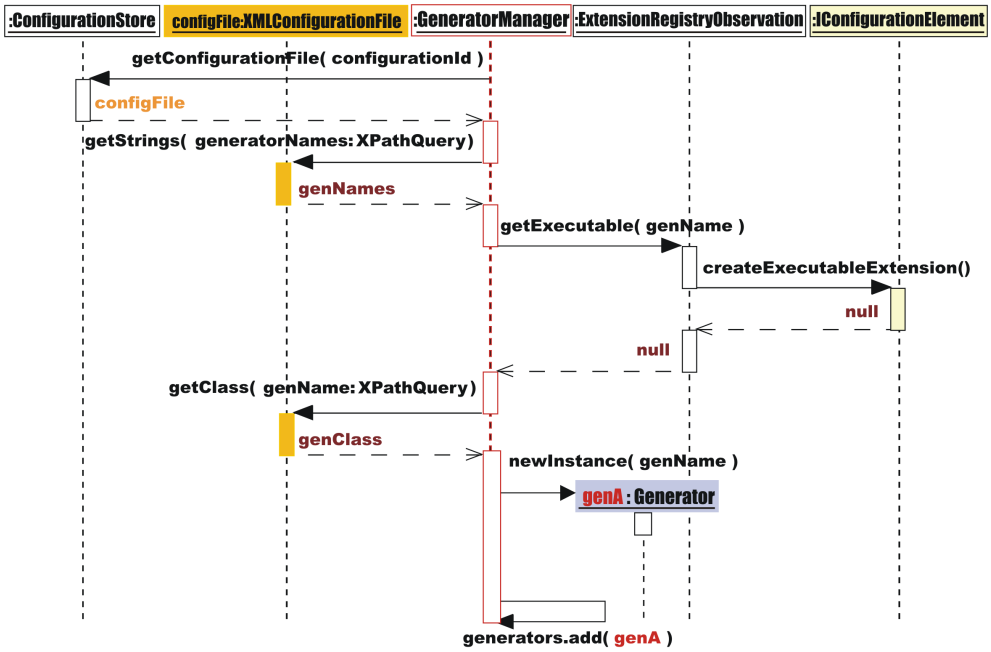


Abbildung 4.10: Instanzierung der Synthese-Komponenten

Der `GeneratorManager` ist das Kernstück der Synthesephase. In ihm enthalten sind unter anderem `Generator`en, `ParameterSupplier`[11] und in denen enthalten `ParameterCache`s[12]. Ein `ParameterSupplier` wird befragt, wenn ein Generator – mit Parametern – gestartet werden soll. Im Standardfall wird ein `ParameterSupplier` der KOMPILIERER UMGEBUNG erstellt und dem `Generator` zugewiesen. Ändert sich nichts an der Konfiguration, bleibt dieses Paar für fortwährende Benutzung erhalten. Für jeden Generator wird außerdem ein Prozess-Beobachter erstellt. Dieser *ProgressObserver* kann vom Generator-Plugin geliefert werden oder eine Standard-Komponente des RAHMENWERKS sein. Für alle gilt, dass sie dann, wenn sie gebraucht werden, entweder instanziert werden oder von einem vorherigen Durchlauf schon vorhanden sind.

Das Sequenzdiagramm 4.10 zeigt beispielhaft einen Instanzierungsvorgang eines `Generator`-Objektes, das nicht Teil eines Plugins ist.

Ein `GeneratorManager` ist genau für eine Konfiguration zuständig. Wenn eine Konfiguration das erste Mal für einen Übersetzungsvorgang benutzt wird, wird auch ein entsprechender `GeneratorManager` erstellt. Wie im Entwurfsmuster *Singleton* (nach [2]) beschrieben, wird für eine Konfiguration eine Instanz eines `GeneratorManager` erstellt. Es gibt, anders als in [2] beschrieben, für jede angewendete Konfiguration ein *Singleton*. Eine statische Methode der Klasse `GeneratorManager` erlaubt den Zugriff auf das Objekt, passend zur Konfiguration.

[11] Lieferanten für Parameter
[12] Klassen zur Zwischenspeicherung von Parametern.

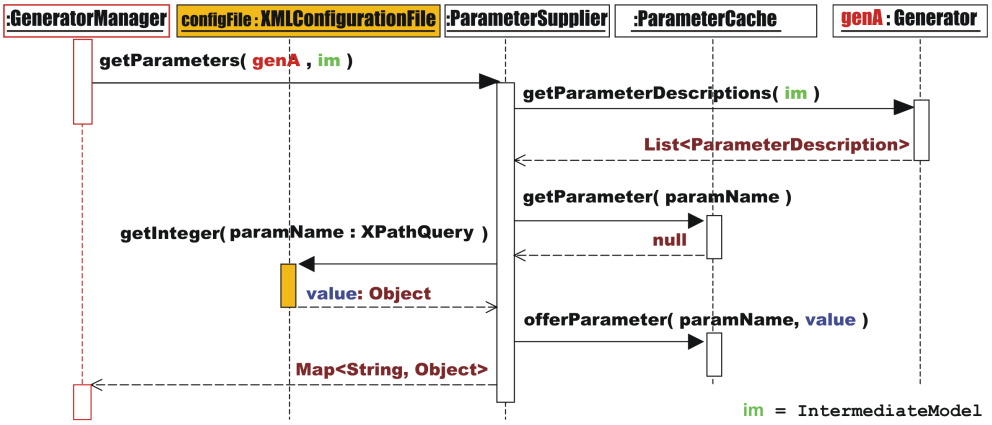


Abbildung 4.11: Ablauf der Parameterbeschaffung

Ein `GeneratorManager` führt noch eine weitere wichtige Variable, die logische Variable `valid`. Sind alle Listen von Generatoren, Parameter-Lieferanten, Caches und ProgressMonitoren[13] bereit, wird `valid` auf `true` gesetzt. Wird das Löschen vorgehaltener Werte verlangt, wird `valid false`. Das geschieht auch wenn sich etwas an der Konfigurationsdatei, der `ExtensionRegistry` oder an den RAHMENWERK-Einstellungen ändert. Die Synthese beginnt mit dem Aufruf `generatorManager.generate(zwischenModell)`. Es wird zuerst folgendes überprüft; ist `valid true`, werden die Parameter mit den vorhandenen Parameter-Lieferanten und den ebenfalls vorhandenen Generatoren aufgelöst, und die Generatoren werden gestartet. Es ist davon auszugehen, dass die meisten Parameter zwischengespeichert waren (siehe Abschnitt 3.3.2), der Zeitverlust für die Beschaffung der Instanzen und Parameter ist dann gering.

Den Inhalt einer Konfiguration liest der `GeneratorManager`, genau wie das Kommandozeilen gesteuerte Rahmenwerk, aus `XMLConfigurationFile`-Objekten heraus. Diese Objekte werden im `ConfigurationStore` des RAHMENWERKS PLUS vorgehalten. Wurde eine Generator-Kennung aus der Konfiguration ausgelesen, versucht der `GeneratorManager`, ein entsprechendes Generator-Plugin mit dieser Kennung in der `Extension-Registry` eclipses zu finden (vgl. Abschnitt 3.4.1). Zugriffsmethoden und Zwischenpeicherung von Einträgen übernimmt die Klasse `ExtensionRegistryObservation` der RAHMENWERKS. Ist ein Eintrag vorhanden, wird von eclipse ein Objekt passend zur Erweiterung erzeugt. In der Abb. 4.10 ist zwar ein solches Objekt vorhanden, es ist jedoch fehlerhaft. Dieses im normalen Ablauf eigentlich nicht vorkommende Szenario ist gewählt worden, um auch den Fall aufzuzeigen, in dem ein Generator über ein Plugin eingebunden wird. Falls die Erzeugung eines GENERATOR über den eclipse-Plugin Mechanismus nicht funktioniert, wird aus der Konfiguration die Klassenbezeichnung des zu initialisierenden `Generator` ausgelesen. Ist die Klasse bekannt, kann eine entsprechende Instanz erzeugt und im `GeneratorManager` der Liste der Generatoren hinzugefügt werden.

[13] Auf ProgressMonitore wird im Folgenden noch eingegangen.

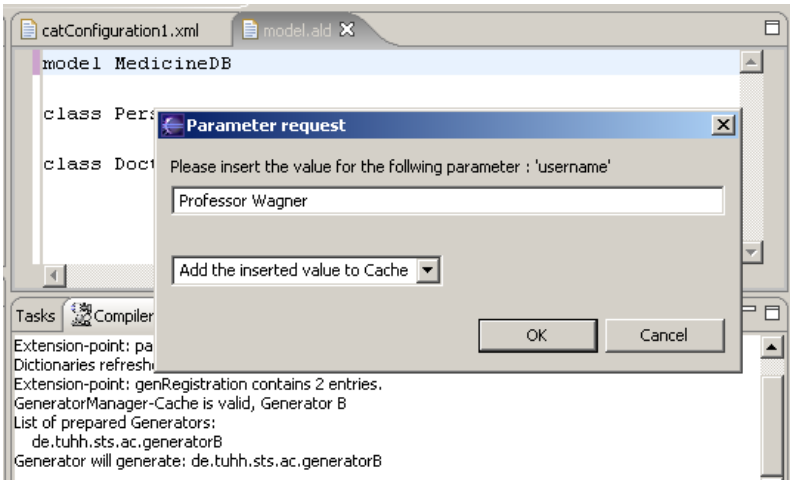


Abbildung 4.12: Dialog zur Eingabe von Parametern

Bei der Erstellung eines `ParameterSupplier`, `ParameterCache` oder `ProgressObserver` wird ähnlich vorgegangen. Es wird versucht, die Instanz der `ExtensionRegistry` nach einem Eintrag zu befragen. Ist kein solcher vorhanden, wird eine Standard-Komponente des RAHMENWERKS erzeugt.

4.5.1 Vorgehensweise der Parameterbeschaffung

Betrachtet man das Sequenzdiagramm 4.11, stellt sich der Ablauf der Parameterbeschaffung dar. Bei Anlauf der Synthesephase ist erstmals bekannt, welche Parameter von den Generatoren benötigt werden. Dann ist nämlich erst das aktuelle Zwischenmodell[14] verfügbar. Dem `ParameterSupplier` wurde bei der Initialisierung ein `ParameterCache` zugewiesen. Damit der Generator gestartet werden kann, müssen alle Parameter für den Lauf in Form eines *Dictionary* (`java.util.Map`-Objekt) bekannt sein.

Für die Beschaffung der Parameter ist der `ParameterSupplier`, im Standardfall der `FileParameterSupplier`, zuständig. Diese Implementierung sieht vor, zuerst eine Liste von Parameterbeschreibungen von dem Generator zu verlangen. Diese Beschreibungen enthalten Informationen über den Typ des Rückgabeobjektes sowie ein Defaultobjekt. Dann wird für jeden Wert der Cache befragt; falls nicht vorhanden, wird versucht, den Wert aus der Konfigurationsdatei zu lesen. Diese Konfigurationsdateien sind ebenfalls zwischengespeichert, was die Abb. nicht zeigt, im vorangegangenen Abschnitt jedoch erläutert worden ist.

Die Parameter werden also aus der Konfiguration gelesen, und falls einer nicht vorhanden ist, wird ein Dialog geöffnet und der Benutzer nach dem Parameter gefragt. Wie ein solcher Dialog sich darstellt, ist in der Abb. 4.12 zu sehen. Der Benutzer kann dann entscheiden, ob der Wert in dem Cache gespeichert werden soll oder nicht. Genau das passiert dann auch.

[14] Ausgabe der Analysephase: auch IntermediateModel genannt.

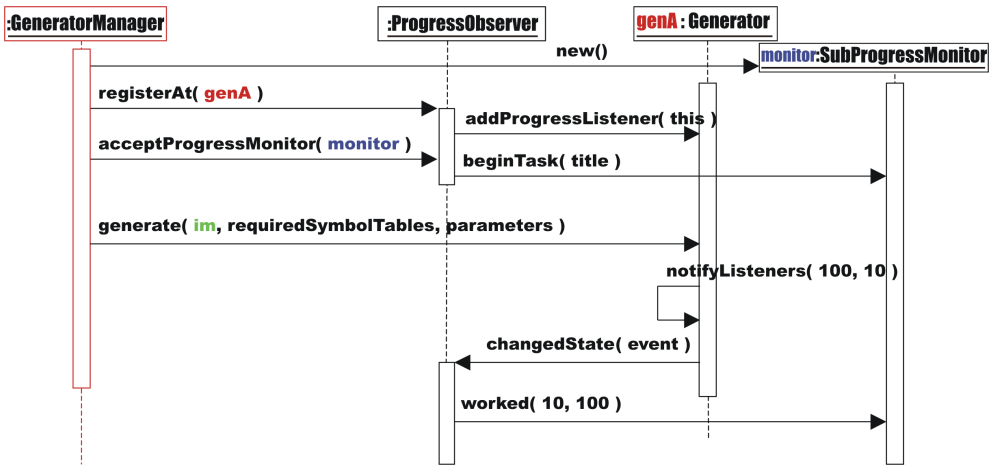


Abbildung 4.13: Laufzeitbetrachtung eines Prozess-Beobachters

Der Wert wird dem Cache angeboten. Die Liste der Parameter wird anschließend dem `GeneratorManager` zurückgeliefert und der Generator kann damit gestartet werden.

4.5.2 Observierung des Prozessfortschritts

Im Sequenzdiagramm 4.13 ist zu sehen, wie die Beobachtung des Prozessfortschritts eines `Generator` funktioniert. Im Design-Abschnitt 3.1 wurde schon beschrieben, dass Generatoren keine `ProgressObserver` als eclipse-Komponenten kennen dürfen. Vielmehr handelt es sich um Beobachter, die auch durch andere Klassen implementiert werden könnten. Im Diagramm ist eine eclipse-Komponente aufgezeigt, die für die Visualisierung des Prozessfortschritts in eclipse verantwortlich ist: `ProgressMonitor`. Solche Objekte können in `SubProgressMonitor`-Instanzen unterteilt werden. Bevor der `GeneratorManager` mit der Generierung beginnt, nimmt er eine Unterteilung für jeden `Generator` vor. Der Gesamtfortschritt der Übersetzung setzt sich aus den Stati der Unterteilungen zusammen.

Falls der Benutzer wünscht den laufenden Prozess abzubrechen, wird dieser Wunsch zu allen `ProgressMonitor`- und `SubProgressMonitor`-Instanzen propagiert. Da der `ProgressMonitor` seine zu beobachtenden Generatoren kennt, kann die Generierung abgebrochen werden. In der Generator-Schnittstelle ist das jedoch noch nicht vorgesehen, so dass das nur bei Generatoren möglich ist, die dafür entspechend erweitert worden sind.

4.5.3 Einbindung individueller Synthese-Komponenten

Die letzte der Annahmen im Design-Abschnitt 3.3.2 war, dass alle Generatoren mit dem Zwischenspeichern einverstanden sind. Das kann nicht immer gewährleistet werden.

Es kann sein, dass Caches unter individuellen Umständen ungültig werden.

```
<extension point="de.tuhh.sts.ac.compilerCore.genRegistration
    ">
  <generator responsibility="testing purpose"
      name="generatorC"
      class="de.tuhh.sts.ac.generatorC.GeneratorCBase"
      parameterSupplier="de.tuhh.sts.ac.generatorC.
          CParameterSupplier"
      parameterCache="de.tuhh.sts.ac.generatorC.
          CParameterCache"/>
</extension>
```

Kode-Beispiel 4.5: Bekanntmachung eines Generator-Plugins

Der Vorgang der Parameterbeschaffung vereinfacht sich, wenn für alle möglichen Zwischenmodelle die gleichen Parameter angefordert werden. Es entfällt dann, den Generator nach Parameterbeschreibungen zu befragen.

Aus diesen Gründen ist es notwendig, dass andere `ParameterSupplier` und `ParameterCache` für einen Generator bei der KOMPILIERER UMGEBUNG angemeldet werden können. Die Anmeldung funktioniert, wie bekannt, über den Erweiterungs-Mechanismus von eclipse. Sind individuelle Klassen registriert, werden diese anstelle der Standard-Implementierungen geladen. Betrachtet man den Ablauf der Parameterbeschaffung, kann dieser also vollkommen bis zur Rückgabe der Parameter angepasst werden. Das Kode-Beispiel 4.5 zeigt, wie ein Generator in die KOMPILIERER UMGEBUNG eingebunden werden kann. Im Abschnitt 3.7.4 wird ebenfalls das Beispiel betrachtet und ergänzend erklärt.

4.6 Einstellungen

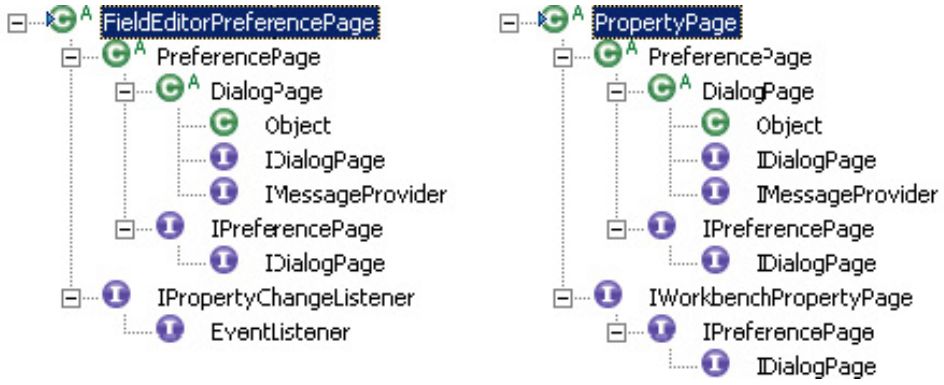
Im Folgenden wird auf die Details der Implementierung für das Einbinden von Einstellungsseiten eingegangen. In den Abschnitten 3.2 und 3.7 wurde das Konzept vorgestellt.

Preferences werden im Speicherbereich des Plugins für alle Projekte gespeichert. Dazu wird eine Klasse `PreferenceStore` verwendet, die von der Plugin-verwaltenden Klasse gestellt wird.

Properties residieren bei den Projekten. Soll herausgefunden werden, ob projektspezifische Einstellungen aktiviert sind, muss unter dem entsprechenden Projekt überprüft werden, ob der Wert mit einem bestimmten Schlüssel `true` entspricht. Da projektspezifische Einstellungen auch mit Mitgliedern einer Programmiergruppe synchronisiert werden können, werden Schlüssel-Wert-Paare in einer serialisierbaren Form gespeichert: als *Zeichenkette*.

Dass man beide Einstellungsdialoge mit gleicher Anzeige nicht doppelt implementiert, steht außer Frage. Um *Preferences* bei eclipse anzumelden, wird eine `extension`[15] in der Manifestdatei definiert und damit eine realisierende

[15] `extension point = org.eclipse.ui.preferencePages`



(a) FieldEditorPreferencePage (b) PropertyPage

Abbildung 4.14: Vererbungshierarchien im Vergleich

Klasse bekanntgegeben. Genauso verhält es sich bei *Property* Seiten[16]. Die Anordnung der Seiten und Filter für die Anzeige unterscheiden sich. Das Problem ist jedoch, dass diese beiden unterschiedlichen `extension-point` Definitionen unterschiedliche implementierende Klassen vorraussetzen. Die Instanzen werden vom eclipse-Workbench (siehe 3.4.3) geladen. Dafür implementieren *Preferences* die Schnittstelle `IPreferencePage`, und *Properties* die Schnittstelle `IWorkbenchPropertyPage`.

In [14] wird erklärt, wie solche Einstellungsseiten gemacht werden können, ohne der überaus komplexen Implementierung im JDT-Plugin zu folgen. Ein Teil dieser Struktur, wie sie im Artikel beschrieben wurde, ist erhalten geblieben, auch wenn sich Teile geändert haben. Zur Verdeutlichung des Aufbaus wird im Folgenden ein Teil des oben genannten Artikels zusammengefasst wiedergegeben. Zum Verständnis wird noch kurz erwähnt, dass eine Implementierung von der Schnittstelle `IPreferencePage` durch `FieldEditorPreferencePage` von eclipse mitgeliefert wird. Diese Implementierung vereinfacht die Erstellung von *Preference*-Seiten. Sie lässt eine beschränkte Anzahl von Elementen zu, die einfach zu integrieren sind: vergleiche hierzu den Artikel [15].

Wie zuvor beschrieben wurde, werden die verschiedenen Arten der Einstellungen an unterschiedlichen Orten gespeichert, eine `*PreferencePage` speichert automatisch seine Schlüssel-Wert Paare in einem dem Plugin zugewiesenen Teil.

Es muss, nach [14], also eine Klasse neu geschrieben werden, die den Speicherort und den Zugriff auf Werte anders implementiert. Leider funktioniert das nicht bei *FieldEditorPreferencePage*, da diese intern auf den Speicherort zugreift[17]. Man würde die *Property*-Seite komplett neu implementieren und dabei die `Field-Editoren` durch normale `SWT Widgets` ersetzen müssen. Eine andere Möglichkeit ist es, sich den Umstand zu Nutze zu machen, dass sowohl `FieldEditorPreferencePage` als auch `PropertyPage` die Klasse

[16] `extension point = org.eclipse.ui.propertyPages`
[17] Die Verwendung dieser Klasse ist im Prinzip jedoch sehr vorteilhaft.

`PreferencePage` erweitern. Dieses kann man an den Vererbungshierarchien Abb. 4.14 sehen. Die Klassen unterscheiden sich in der Implementierung von `IPropertyChangeListener` bzw. von `IWorkbenchPropertyPage`. Die Erweiterung von `PreferencePage` auf `PropertyPage` ist minimal, da die Implementation von `IWorkbenchPropertyPage` trivial ist, deswegen macht es Sinn, die Klasse `FieldEditorPreferencePage` als Basis zu nehmen.

In den Abbildungen sind Symbole zu sehen, ein C steht für eine Klasse. Ist ein hochgestellte A dabei, ist diese Klasse abstrakt, I steht für Schnittstellen (engl.: Interface).

Da *Preferences* ihre Werte an `PreferenceStore`-Objekte abgeben und ebenso Werte von diesen erhalten, muss also, wenn es sich andernfalls um ein *Properties*-Seite handelt, der neu erstellten Einstellungsseite ein Objekt vom Typ `PreferenceStore` zur Verfügung gestellt werden, das sich jedoch anders verhält. Dessen Werte werden bei den Eigenschaften der Ressource gespeichert. Diese neue Klasse erbt von `PreferenceStore` und wird in [14] `PropertyStore` genannt. Die `PropertyStore`-Klasse ist komplett, wie in [14] vorgeschlagen, in die KOMPILIERER-UMGEBUNG übernommen worden. Wenn also die *Properties* eines Projektes geändert werden sollen, wird ein `PropertyStore`-Objekt instanziert und als `PreferenceStore` (es ist ja eine Subklasse davon) den Editoren übergeben. Sollen die globalen *Preferences* geändert werden, so wird von der Plugin-Klasse `CompilerCorePlugin` eine Referenz auf den originalen `PreferenceStore` verlangt.

Die neu entstandene Klasse, die sowohl `FieldEditorPreferencePage` erweitert als auch `IWorkbenchPropertyPage` implementiert, wird im Folgenden `FieldEditorOverlayPage` genannt. Jene Klasse ist für die Zuweisung der `*Store` zuständig und verwaltet die Umschaltmöglichkeit im Dialog für projektbezogene Eigenschaften zwischen individuellen Einstellungen und den globalen.

4.6.1 Nutzung der OverlayPage

Um eine eigene Seite nach dem Muster einer `FieldEditorOverlayPage` zu gestalten, sind nur wenige Schritte notwendig. Die Einstellungsmöglichkeit ob 'projektspezifisch' oder 'global', wird von `FieldEditorOverlayPage` erstellt. Verbindung mit dem entsprechenden Store nimmt ebenfalls *OverlayPage* vor. Individuell gestaltet sich der Ort, an dem die *Preferences* gespeichert werden. Es wird eine Subklasse von `FieldEditorOverlayPage` erstellt, die auf den `PreferenceStore` des Plugins verweist. Die erweiternde Klasse wird in der KOMPILIERER-UMGEBUNG `CompilerOverlayPage` genannt.

`CompilerOverlayPage` müsste nun `FieldEditor`-Instanzen erzeugen. Da die `CompilerOverlayPage` generisch erweitert werden kann, liefern die erweiternden Klassen den Inhalt. Wenn bei *Projekt-Properties* auf projektspezifische Eigenschaften umgeschaltet wird, müssen alle Editoren aktiviert werden, bei Umschaltung auf Nutzung der *Preferences* werden alle Editoren grau unterlegt und somit inaktiv.

Innerhalb einer Einstellungsseite können Änderungen von Feldern eine Ak-

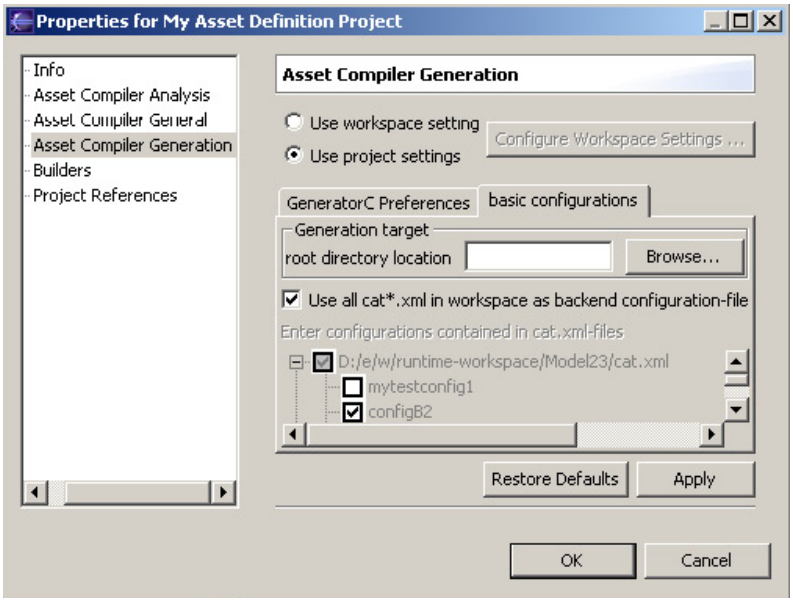


Abbildung 4.15: Projektbezogene Einstellungsseite der Kategorie Generation

tivierung oder Deaktivierung anderer Felder nach sich ziehen. Wie in Abb. 4.15 zu sehen ist, ist die Baumstruktur nicht aktiv. Entfernt man die Selektion in der Einstellung direkt darüber, so aktiviert man den Baum. Im folgenden werden nun Einstellungsseiten betrachtet und aufgezeigt, wie solche der KOMPILIERER-UMGEBUNG hinzugefügt werden können.

Um die Erweiterung der Einstellungsseite generisch bewerkstelligen zu können, sind einige Konventionen notwendig:

- Die realisierende Klasse muss die Schnittstelle `...compilerCore.interfaces.IPreferencePage` implementieren.
- Bei der Erstellung eines Reiters wird die Erstellung eines `Composite` mit `FieldEditor`en verlangt. Diese `FieldEditor`en müssen direkte Kinder eines `Composite` sein, das der `CompilerOverlayPage` bekannt zu machen ist. Manchmal ist es notwendig eigene `FieldEditor`-Typen zu erschaffen.
- Bei der Initialisierung und bei Einstellungsänderungen werden `IPreferencePage`-Instanzen informiert, sie haben dann selbstständig die Aktivierung bzw. Deaktivierung ihrer Elemente vorzunehmen.
- Voreinstellungen[18] der Werte müssen bei der Initialisierung vorgenommen werden.
- Die Schlüssel-Wert Paare globaler Einstellungen werden dann, wie zuvor beschrieben, in dem *Store* des `CompilerCore`-Plugins gespeichert.

[18] Auch *Default*-Werte genannt.

```
<extension
        point="de.tuhh.sts.ac.compilerCore.
            generatorPreferences">
    <page class="de.tuhh.sts.ac.generatorC.
        GeneratorCPreferencePage"/>
</extension>
<extension
    point="de.tuhh.sts.ac.compilerCore.preferenceListener">
    <listener class="de.tuhh.sts.ac.generatorC.
        PreferenceListener"/>
</extension>
```

Kode-Beispiel 4.6: Einbindung einer Preference-Seite

> Sie werden also nicht in den jeweiligen *Stores* der der teilnehmenden Plugins aufbewahrt[19]. Deswegen ist eine eindeutige Wahl der Schlüssel unabdingbar. Die eindeutige Identifikation des Plugins sollte darum in dem Schlüssel enthalten sein.

Man kann sich diesen Erzeugungsprozeß wie bei dem Entwurfsmuster: Schablonenmethode (nach [2]) vorstellen, bei der konkrete Seiten für Teilaspekte der Inhaltsgestaltung Methoden definieren, ohne an dem Ablauf des Seitenaufbaus beteiligt zu sein.

Auch die KOMPILIERER UMGEBUNG hat eigene Einstellungsseiten. Diese Seiten werden genauso behandelt wie alle anderen auch. Es ist also auch die Definition einer `extension` diesbezüglich notwendig. Auch wenn das zuerst ein wenig kompliziert klingt, einen `extension-point` in einen Plugin zu definieren und dann auch eine passende `extension` dazu erstellen zu müssen, liegen die Vorteile auf der Hand. Im Buch [7] führt der Autor eine Regel an: FAIR PLAY RULE. Alle, auch das eigene Plugin, haben sich an die gleichen Regeln zu halten, ansonsten würde man auf wertvolles Feedback verzichten. Wenn das eigene Plugin selbst die `extension-point`s nutzt und nicht „nach eigenen Regeln spielt“, fallen oft noch Unstimmigkeiten im Design auf. Änderungen an der Art des Ablaufs müssen so auch nur einmal vorgenommen werden und nicht für die eigene und für die Schnittstelle nach außen separat. In Abb. 4.15 sieht man einen Beispiel-Dialog. Die allgemeinen Generatoreinstellungen der KOMPILIERER UMGEBUNG sind nach den Einstellungen für den `GeneratorC` zu sehen.

Die zufällige Reihenfolge der Anordnung der Reiter ist allerdings ein Nachteil. Die realisierenden Seiten melden sich alle gleich über ihre Manifestdateien an. Beim Öffnen der Einstellungen wird nun die `ExtensionRegistry` von eclipse nach solchen Einträgen durchsucht. Die Reihenfolge der Einträge wird nicht deterministisch von eclipse erstellt. Nun lässt es sich streiten, ob zusätzliche Maßnahmen zur Sortierung der Einträge den Mehraufwand an Rechenleistung

[19] Das COMPILERCORE-PLUGIN ist als Teilnehmer und Einbindungsumgebung eine Ausnahme.

und Speicherbedarf lohnen. Um auch für diejenigen, die Erweiterungen eintragen möchten, dieses nicht noch unnötig zu komplizieren, wird davon Abstand genommen.

4.6.2 Zugriff auf Einstellungswerte

Über die Klasse `AccessSettings` können Werte aufgelöst werden. Dazu sind die Schlüssel und die Ressourcen, bei denen die Werte gespeichert sein können, mit zu übergeben. Bei der Ressource handelt es sich um ein Projekt, bei dem eventuell projektspezifische Daten gespeichert wurden. Es wird dann von `AccessSettings` überprüft, ob die globalen oder die projektspezifischen Werte Gültigkeit haben.

Ein zusätzlicher Parameter stellt die Kategorie dar, unter der ein Wert abgespeichert wurde. Dazu sind in der Klasse `AccessSettings` statische Variablen veröffentlicht, die entsprechend der Kategorie der Einstellungen gewählt werden müssen[20].

4.6.3 Beobachter von Einstellungsänderungen

Beobachter haben die Schnittstelle *IPreferenceListener* zu implementieren. Eine Benachrichtigung enthält die Art der Einstellung, wie in der Klasse *AccessSettings* definiert.

Es wird an dieser Stelle das Beispiel mit dem *GeneratorC* (vergleiche Abschnitt 3.7.2) wieder aufgegriffen. Der *GeneratorC* hat eine Seite mit einem Einstellungswert. Wird der Wert geändert, wird ein `IPreferenceListener` implementierendes Objekt instanziert. Der *GeneratorC* hat z.B. einen eigenen Cache für Generierungsparameter implementiert und kann diesen jetzt nach der Benachrichtigung invalidieren. Mehr Informationen zu der Verwendung von Caches für Generatorparameter, sind im Abschnitt 4.5.3 zu finden.

4.6.4 Auslesen der Konfigurationsdatei

Wie im Abschnitt 3.3 über die Lebensdauer von Objekten im Rahmenwerk Plus gefordert, werden die entsprechenden Konfigurationsdateien nicht bei jeder Systemerzeugung wiederholt ausgelesen.

Es gibt die Klasse `ConfigurationStore`, die zwei Aufgaben hat. Es wird ein Zugriff auf die Klassen zum Auslesen der Konfigurationsdateien bereitgestellt, und Änderungen an Konfigurationsdateien werden überwacht und propagiert. Für den Zugriff auf die Konfigurationsdateien wird eine Bibliothek aus dem originalen Rahmenwerk verwendet. Unter Angabe der Kennung einer Konfiguration[21] wird eine zwischengespeicherte Instanz, die den Zugriff ermöglicht, herausgesucht. Sollten Änderungen an Konfigurationsdateien die Invalidierung der Zwischenspeicher notwendig machen, wird dieses getan und propagiert, bevor eine neue Systemerzeugung startet (siehe Abschnitt 3.5.4).

[20] Beschreibung der drei Kategorien : General, Frontend und Backend in Abschnitt 3.7.2
[21] Eine Konfigurationsdatei hat mehrere Konfigurationen.

4.7 Automatisierte Tests

Unter *Test-Driven Development* (TDD) versteht man eine Programmier-Vorgehensweise, die automatisierte Tests vor die Implementation stellt. Oftmals Teil der weitgreifenderen Entwicklungsmethodik *Extreme Programming* (XP), z.B. [17], findet TDD, auch davon losgelöst, sinnvoll Verwendung. Auch wenn XP seine Fürsprecher hat, siehe [19], herrscht in der Literatur kein Konsens, ob diese Art des Programmierens zu bevorzugen ist. XP ist Limitationen unterworfen (siehe [20]).

TDD soll zu Programmen führen, die weniger Fehler aufweisen als phasenorientierte Ansätze, wie sie z.B. in [3] beschrieben sind. Es soll außerdem die Lesbarkeit von Quellkode erhöht werden. Beide Aspekte werden noch vertieft. In den beiden Büchern [18] und [7] – eine in eclipse integrierte Testumgebung wird in diesen Büchern zur Erläuterung herangezogen – wird beispielsweise aufgezeigt, dass im Endeffekt weniger Arbeit und Stress durch die zusätzliche Implementation von Tests resultieren. Eine wichtige Differenzierung ist hier notwendig. Die hier besprochenen Tests haben **nicht** die Aufgabe, folgende **nicht-funktionale** Anforderungen zu testen wie z.B.:

- *Leistung* - (performance) Geschwindigkeit der Verarbeitung, Ressourcenverbrauch
- *Belastbarkeit* - (stress) Skalierbarkeit des Systems
- *Benutzbarkeit* - (usability) Handhabbarkeit des Systems für den Benutzer.

Es handelt sich vielmehr um automatisierte Tests, die in einem bestimmten Testszenario zur Laufzeit der Tests Kode **funktional** überprüfen.

Automatisierte Tests sind auf Betrachtungen bzgl. Semantikbeschreibungen in Programmiersprachen zurückzuführen. Seit den 60er Jahren werden Vor- und Nachteile von Pre-, Post und Side-Conditions in Programmiersprachen diskutiert. Exemplarisch wird an dieser Stelle auf [21], [22] und Programmiersprachen wie VDM, Z und Object Z verwiesen. Automatisierte Tests testen ausschließlich zur Laufzeit und sind auf explizit programmierte Testszenarien beschränkt. Die erfolgreich getesteten Programme können demnach in nicht getesteten Szenarien versagen. Vorteilhaft ist jedoch, dass das zu testende Programm vor Beginn eines Tests auch in keinem unvorhergesehenen Zustand ist. Bei Semantikbeschreibungen in Programmiersprachen können solche unvorhergesehenen Zustände fälschlicherweise zu Fehlermeldungen führen, da ein Teil eines Programms auf verschiedenen Wegen erreicht werden kann.

Im Folgenden wird ein Beispiel aus der Implementierung des Übersetzer-Rahmenwerks angegeben.

```
1 import java.util.ArrayList;
  import java.util.Arrays;
  public static ArrayList<String> parseString(String iDs) {
      return Arrays.asList(iDs.split(SEPARATOR));
5 }
```

Kode-Beispiel 4.7: Beispiel einer fehleranfälligen Implementierung

4.7.1 Szenario für einen sinnvollen automatisierten Test

Da sich Benutzeransichten nur bedingt testen lassen, habe ich darauf weitestgehend verzichtet[22]. Weil die UI-Komponenten von eclipse aufgerufen und verwaltet werden und zwingend in einem eigenen Thread aufgerufen werden, ergeben sich für automatisierte Tests Komplikationen. Benutzereingaben dürfen ebenfalls *nicht* zur Ausführung notwendig sein. Ein selbst implementierter Auswahlbaum, wie er in Abb. 4.15 zu sehen ist, speichert selektierte Werte in einer Zeichenkette. Es gibt also eine Methode zur Konvertierung von einer Baumstruktur in eine Zeichenkette und eine, die aus einer Zeichenkette eine Liste von Objekten macht. Um aus einer Zeichenkette eine Liste zu machen, wurde Kode wie im Auszug 4.7 geschrieben. Ein Fehler offenbart sich bei der Benutzung erst, wenn folgender Fall auftritt: die entsprechende Liste wird verlangt, obwohl nur eine leere Zeichenkette vorliegt. Die referenzierte Implementierung gibt nun eine Liste mit einem Wert zurück, der leeren Zeichenkette. Das macht beim vorliegenden Design keinen Sinn. Wenn für die nächste Systemerzeugung keine Konfigurationen ausgewählt wird, darf die Liste der Konfigurationen nicht die Länge 1 haben, sondern 0. Die Implementierung wird um eine Abfrage ergänzt, ob es sich nicht um eine leere Zeichenkette handle.

Ein Fehlerfall wird jedoch nicht so schnell offensichtlich. Fügt man auf anderem Wege als über die Benutzerschnittstelle, also durch direkten Zugriff auf die gespeicherten Werte, eine Konfiguration der Zeichenkette hinzu, kann die Eingabe nicht der Norm entsprechen. Sind so Leerzeichen in der Zeichenkette enthalten, bleiben diese in der Liste erhalten. Der Test getriebene Ansatz offenbart solche Schwachstellen. Im Anhang B.2 findet man die Testklasse. In den Zeilen 21ff wird eine Zeichenkette der zu testenden Methode übergeben. Im Folgenden offenbarte sich, dass einer der zurückgelieferten Einträge nicht dem Erwarteten entspricht. Bei dem zweiten Eintrag wurde die gewünschte Identifikation inklusive einem Leerzeichen zurückgegeben. Dies ist zum Test auf Gleichheit von Zeichenketten unbrauchbar.

Anhand dieses Beispiels wird deutlich, wie der durch Tests getriebene Ansatz den Entwickler diszipliniert. Man wird dazu gezwungen, alle Möglichkeiten der Eingabe und der Ausgabe zu überdenken, bevor die eigentliche Implementierung erfolgt. Bei phasenorientierten Ansätzen findet das teilweise im Design statt.

[22] Interessant ist jedoch, dass zu SWT (siehe 3.4.3) automatisierte Tests vorhanden sind.

4.7.2 Weitere Vorteile automatisierter Tests

Zwei weitere Vorteile entstehen durch die Definition automatisierter Tests.

- **Dokumentation**. Automatisierte Tests dokumentierten die Intention der zu testenden Klassen. Durch die Tests ist zumeist beispielhaft vorgegeben, welche Ausgaben auf bestimmte Eingaben zu erwarten sind. Außerdem werden in den Test idealerweise auch Seiteneffekte mit überprüft. Wie im Beispiel des vorangegangenen Abschnitts sichtbar wird, kann man in der Testklasse (Kode-Beispiel B.2 im Anhang) sowohl die Überprüfung von `null` als Eingabeparameter, als auch als Ausgabe sehen. Ein `null` als Eingabe ist erlaubt. Niemals darf jedoch `null` als Ausgabe auftreten.

- **Veränderungen an Methoden bleiben im Rahmen der Annahmen**. Fehler in der Programmierung treten immer auf, der Kode muss korrigiert werden. Es kann auch sein, dass Funktionen in ihrer Funktionalität erweitert oder eingeschränkt werden. Es ist in solchen Fällen sicherzustellen, dass alle Referenzen auf die geänderte Methode weiterhin funktionstüchtig bleiben. Der Aufruf einer Methode ist immer mit gewissen Voraussetzungen verknüpft. Es werden Annahmen über das Verhalten einer Methode gemacht. Wenn sich dieses ändert kann es zu Fehlern kommen. In der API-Dokumentation wird die Funktionalität einer öffentlichen Methode in Prosa dargelegt. Eine Änderung der Dokumentation wird ein Aufrufer nicht gewahr. Werden allerdings vom Aufrufer Tests implementiert, die das Verhalten von Methoden überprüfen, wird eine Änderung der Interaktion mit der Zielmethode einen Fehler in den Tests auslösen. Der eine Methode Verändernde kann demnach in dem Fall, dass alle Test fehlerlos passieren, davon ausgehen, dass die Neuimplementierung keine Folgefehler nach sich zieht, wenn die Testfälle die gesamte Spezifikation abdecken.

Beide für den Einsatz von Test sprechenden Argumente sind besonders in dem vorliegenden Projekt stark zu gewichten. Die Kompilierer-Umgebung ist ein Teil eines größeren Projektes, das die Mitarbeit mehrerer erfordert. Wie es die Natur der Vorgehensweise in einer Universität entspricht, werden die Personen, die daran arbeiten, häufig wechseln. Studenten werden für ein viertel oder ein halbes Jahr daran arbeiten und dann nicht mehr verfügbar sein.

Man stelle sich vor:

– Ein Komponenten-Entwickler (Student) definiere eine Komponente. Dann ändere sich eine Methode des Rahmenwerks und die Komponente funktioniere nicht mehr. Eine der Annahmen, die der Komponenten-Entwickler gemacht hat, treffe nicht mehr zu. –

Bei der Verwendung von automatisierten Tests wird demjenigen, der die Methode im Rahmenwerk ändert, idealerweise deutlich, dass die Änderung eine der früher gemachten Annahmen verletzt. Durch Betrachtung der entsprechenden Testmethode kann herausgefunden werden, auf welche Annahmen zu

achten ist. Korrektive Maßnahmen sollten derart vorgenommen werden, dass die geänderte Methode auch allen alten Anforderungen weiterhin entspricht.

4.7.3 Aufbau der Testumgebung

Drei Komponenten sind für die hier implementierten automatisierten Tests notwendig.

- Die **JUnit Testumgebung** ist eine Ablaufunterstützung von Tests, die eine weitere eclipse-Instanz startet und schließt.
- Ein **Test Project Fixture** ist ein Testscenario, bei dem bestimmte Einstellungen und Ressourcen gegeben werden.
- Die **Test-Klassen und -Methoden** enthalten die eigentlichen Tests selbst.

Im Abschnitt 4.7.1 wird ein Beispiel für Tests aufgeführt. Alle Tests vorzustellen, würde den Rahmen sprengen. Im Falle eines konkreten Bedarfs kann in den Testklassen selbst nachgeschaut werden, deren Aufbau denen der zu testenden Klassen gleicht. Die Testklassen befinden sich in einem getrennten Projekt: `compilerCoreTest`. Auf diese Weise kann man die Tests allen zugänglich machen. Testklassen können den Entwicklern helfen, ihre Komponenten zu implementieren, auch wenn sie keinen Zugriff auf den Quellkode der Übersetzer Umgebung haben. Testklassen können auch dazu verwendet werden, dynamisch eingebundene Komponenten durch ihre Schnittstellen zu testen. Im späteren Betrieb ist es nicht erwünscht, die langsamen Tests immer mit einzugliedern. Darum ist es sinnvoll, sie in einem getrennten Projekt unterzubringen.

Es sind im Paket `de.tuhh.sts.ac.test.compilerCore` Klassen und Pakete enthalten, die dem Aufbau des CompilerCore Plugins entsprechen. Testklassen orientieren sich in der Namensgebung an den zu testenden Klassen, indem deren Name ein „Test“ angehängt wird. Für die Klasse `ClasspathEnlargement` gibt es also eine entsprechende Testklasse mit dem Namen `ClasspathEnlargementTest`.

4.7.4 JUnit Testumgebung

JUnit ist eine Form der xUnit-Testumgebungen[23]. XUnit wurde in über 30 Programmiersprachen portiert. Im Anhang B.2 wird die Klasse TestCase aus den *JUnit-Rahmenwerk* erweitert. Geerbte Methoden wie `assertNotNull()` erlauben dann die Überprüfung, ob ein Objekt nicht `null` ist. Dieses Rahmenwerk allein reicht aber hier nicht zum Test aus. Um Projekte zu testen, die sich gerade in der Entwicklung befinden, wir eine zweite Instanz von eclipse gestartet, bei der die Projekte der *Host-Instanz* als Plugins eingebunden werden. Dieser Aufbau ist in Abb. 4.16 zu sehen. Auf der linken Seite sieht man

[23] Ursprüngliche Implementierung in Python, *J* für Java, also die Java-Implementierung des xUnit

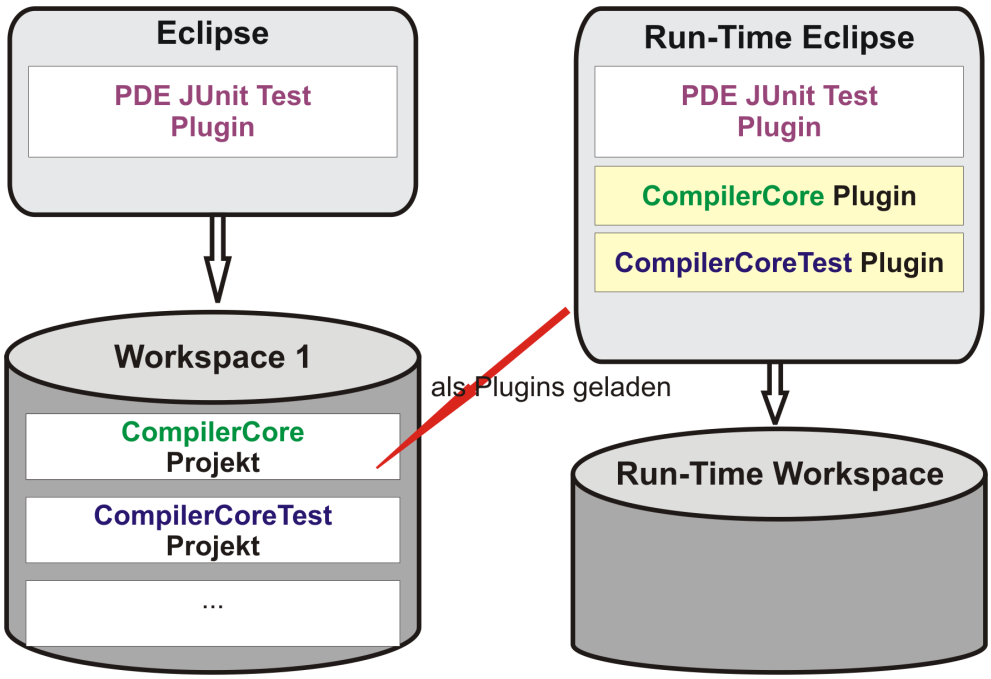


Abbildung 4.16: Die Testumgebung PDE-JUnit

die Instanz von eclipse, in der man arbeitet. Es werden Plugin-Projekte im Workspace erstellt. Mit Hilfe des PDE wird eine neue Instanz gestartet. Die Projekte (auch das die Tests enthaltende) werden als Plugins mit der neuen Instanz geladen. Es kann angegeben werden, welche Testklassen ausgeführt werden sollen, auch die Auswahl von allen in dem Testprojekt enthaltenen ist möglich. Nach Ausführung der Tests schließt sich die neue eclipse-Instanz automatisch, und eine Ansicht zur Testevaluation wird dem Anwender gezeigt.

In der Abb. 4.16 ist zu sehen, dass der Workspace leer ist. Das sind aber keine realen Testbedingungen. Im Workspace fehlen Projekte (siehe Anwendungsfall im Abschnitt 2.5). Im folgenden Kapitel wird darauf eingegangen.

4.7.5 Erstellen von Testszenarios

In Abb. 4.17 wird der nun vollständige Aufbau mit einem Testprojekt[24] aufgezeigt. Im Testprojekt, im Workspace auf der rechten Seite, sind nun all jene Elemente, die für einen Test notwendig sind, vorhanden. Soll z.B. die dynamische Pfaderweiterung der KOMPILIERER UMGEBUNG getestet werden, so muss auch eine Bibliothek vorhanden sein, die nun mit eingebunden werden kann. Diese wird dann im *Fixture* mit berücksichtigt und im Projekt, hier rot: *Test Fixture*, gespeichert. Es ist wichtig, für alle Testläufe einen konsistenten Anfangszustand zu haben. Nur dann kann auch davon ausgegangen werden, dass bei einem Fehler in einem Test auch gerade geänderter Kode der verursachende ist. Nach Beendigung des Testlaufs muss das Projekt wieder gelöscht werden.

[24] Auch Fixture genannt.

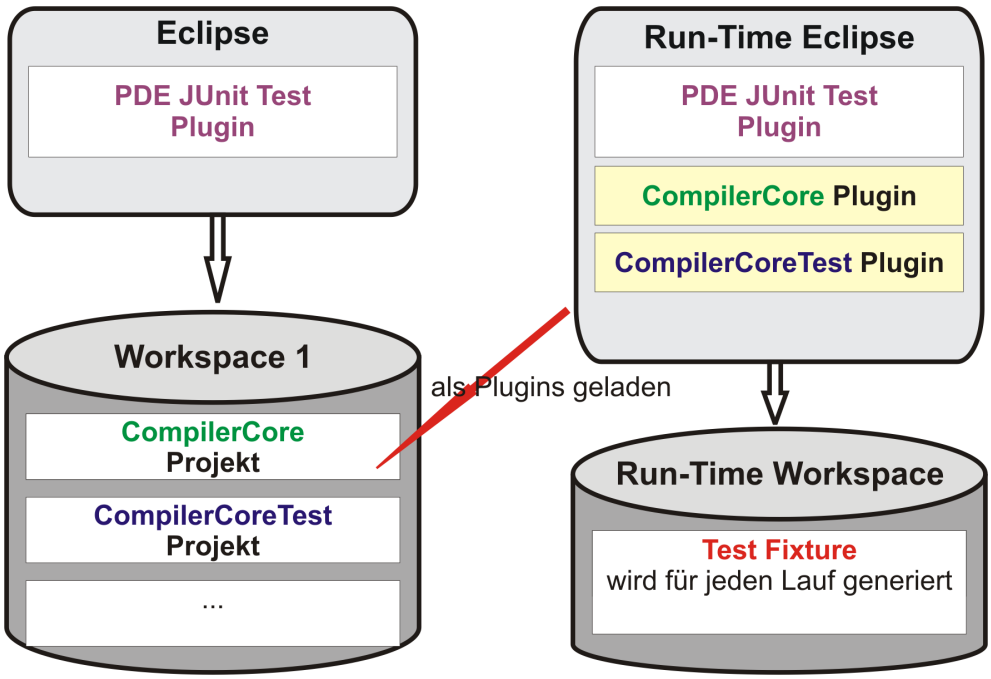


Abbildung 4.17: Testumgebung mit Fixture

Das für die Tests verantwortliche Plugin kann auch von anderen Entwicklern für ihre Aufgaben genutzt werden. Die Erstellung eines passenden *Fixture* ist zumeist das größte Problem. Dabei kann nun eine anpassbare Klasse aus dem Test-Plugin benutzt werden. Unter Übergabe einer Asset-Modelldatei und einer Konfiguration wird ein *Fixture* erstellt und auf Verlangen dann auch gelöscht.

KAPITEL 5

Benutzerführung und Inbetriebnahme

Die Möglichkeit, komplizierte Aktionen mit Hilfe von Dialogen und Wizards ausführen zu können, ist im Abschnitt 4.1.4 schon erwähnt worden. Der Benutzer wird durch den Dialog zu Eingaben animiert, die Aktionen hervorrufen. Nachteil dieser Wizards ist nur zu oft, dass der Benutzer nicht nachvollziehen kann, was hinter den Kulissen passiert. Bei reinen Hilfetexten dagegen sind Erklärungen vorhanden, es passiert jedoch auch nichts automatisch.

In diesem Kapitel soll auf die verschiedenen Arten der Benutzerführung anhand der Realisierung in der Kompilierer Umgebung eingegangen werden. Zu unterscheiden sind dabei 3 Arten; Wizards, Cheat Sheets und Hilfe. Wizards und Cheat Sheets, sowie Menüs und Toolbars haben ein Element gemeinsam, nämlich Aktionen. Eclipse definiert hier einen Typ, der vom Ort des Aufrufs unabhängig ausgeführt werden kann. Zum Verständnis ist ein Abschnitt der Erklärung vorgesehen.

Die Benutzerführung ist nicht Teil des Designs, da sie keine funktionale Erweiterung darstellt. Komplexe Vorgänge werden lediglich erläutert oder automatisiert. Es wurde darauf geachtet, viele Möglichkeiten der Benutzerführung vorzustellen, um eine spätere Erweiterung, aufgrund des bereits Implementierten, einfach zu gestalten.

In dem Abschnitt zur Inbetriebnahme wird diskutiert, welche Faktoren für die Benutzerakzeptanz der Kompilierer-Umgebung wichtig sind. Die Kompilierer-Umgebung wurde in Betrieb genommen.

5.1 Erreichbarkeit von Befehlen und Aktionen

Wenn sich in eclipse ein Menü öffnet, werden je nach Kontext verschiedene Aktionen angezeigt. Eclipse instanziert ausführende Objekte erst, wenn sie tatsächlich benötigt werden. Solange eine deskriptive Beschreibung ausreicht, ist es nicht notwendig, eine solche Instanz zu laden. Solch ein Vorgehen nennt

```
<action id="de.tuhh.sts.ac.project.removeNature"
  enablesFor="1"
  label="Remove Asset Definition Nature"
  tooltip="Remove the Asset Definition Nature if exits"
  class="de.tuhh.sts.ac.compilerCore.builder.AddRemoveNature"
    >
</action>
```

Kode-Beispiel 5.1: Definition einer Aktion in einer Manifestdatei

man *lazy loading*. Benötigte Beschreibungen zur Anzeige in Menüs werden in der Konfigurationsdatei des jeweiligen Plugins zur Verfügung gestellt. Daraus wird ein generisches Proxy-Objekt im Workbench erstellt [7]. Man vergleiche hierzu das Entwurfsmuster: Proxy [2]. Ob ein Element erscheint oder nicht, ob es aktiv oder deaktiviert ist, wird ebenfalls in der Konfiguration beschrieben und fordert das Laden der die *Aktion* repräsentierenden Klasse nicht.

Eventuell verweisen mehrere Proxy-Objekte auf ein gemeinsames Element, genauer auf ein Objekt, das die Schnittstelle `IAction` implementiert. Es handelt sich hier um einen *Befehl*, beziehungsweise einen *KonkreterBefehl*, wie im Entwurfsmuster Befehl [2] beschrieben. `IAction` definiert die Methode `run()`, um den Befehl auszuführen. Zusätzlich stellt ein Befehl folgendes zur Verfügung.

- Alle notwendigen Informationen, um eine Aktion anzuzeigen, sind im Befehl enthalten. Das beinhaltet Titel, Bild, Tooltip und den aktiv-inaktiv Status.
- Wenn der Zustand der Aktion sich ändert, werden Beobachter informiert.
- Ein Befehl kann von vielen Grafik-Elementen gleichzeitig benutzt werden. In Abb. 5.1 wird ein Ausschnitt der Laufzeitobjekte einer eclipse-Instanz sichtbar gemacht. Ein Menü und eine Werkzeugleiste verweisen indirekt auf die **selbe** Aktion.

Ein Befehl wird durch ein Objekt repräsentiert. Man sieht in Abb. 5.1, dass der Befehl *SaveAction* durch eine Kapselung von den verschiedenen UI-Widgets erreicht werden kann. In die andere Richtung funktioniert die Kommunikation über den Beobachter-Mechanismus. Wie im Entwurfsmuster Beobachter [2] beschrieben, ist die `IAction` das Subjekt und die umhüllende Klasse der Beobachter. So wird z.B. nach Ausführung des Befehls der *enabled-Zustand* propagiert. Nach dem Ausführen des Befehls zum Speichern wird das entsprechende Feld in der Werkzeugleiste ebenso wie im Menü grau.

5.2 Der Wizard New-Asset-Project

Ein Wizard kann aus mehreren Seiten bestehen. In der Abb. 5.2(a) ist die erste Seite des Wizards zur Erstellung eines neuen Projektes für Asset-Modell-

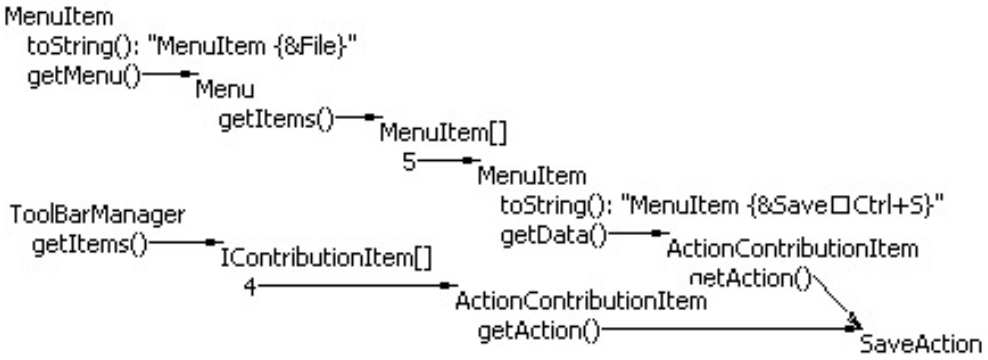


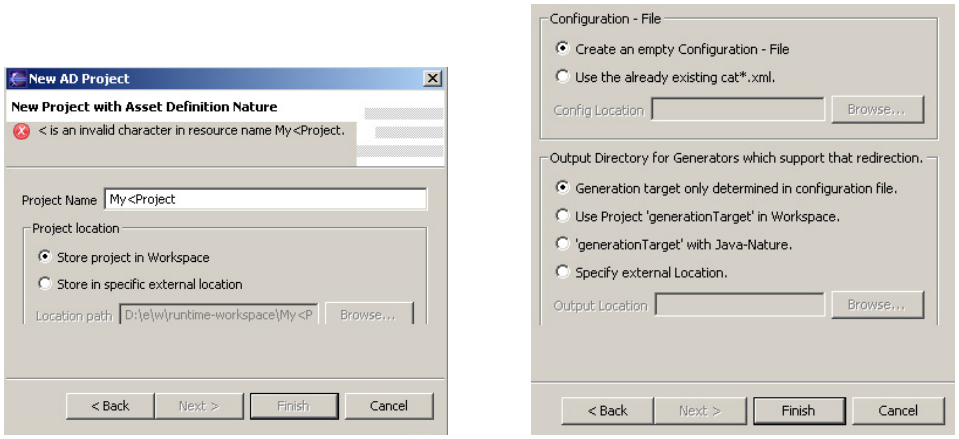
Abbildung 5.1: Menü und Werkzeugleiste teilen eine Aktion

Ersteller[1] zu sehen. Wird ein Projektname eingegeben, so ändert sich der grau hinterlegte „Location Path“ entsprechend. Stammordner ist dabei das Wurzel-Verzeichnis des Workspace. Man kann auch an anderer Stelle ein neues Projekt erstellen. Diese Art Zielortbestimmung ist auch schon von dem „New Java Project Wizard“ bekannt. Man beachte auch die Fehlerkontrolle während der Eingabe. Sollte ein ungültiges Zeichen oder ein ungültiger Pfad eingegeben werden, kann der Wizard ohne Korrektur lediglich abgebrochen und nicht fortgesetzt werden. Eine Fehleranzeige erscheint, wie im oberen Abschnitt in Abb. 5.2(a) sichtbar.

Ein zweiter Dialog erscheint nach Auswahl von „Next“. In Abb. 5.2(b) ist die zweite Seite zu sehen. Es kann für das neue Projekt eine Konfigurationsdatei ausgewählt werden. Diese Datei wird dann in das neue Projekt kopiert. Das anzugebende Ausgabeverzeichnis ist identisch mit der Einstellung im Einstellungsdialog in Abb. 4.15 unter „root directory for generationTarget“. Falls ein solches Ziel noch nicht existiert, wird es erstellt. Dieser Eintrag für das Projekt hat die Konsequenz, dass alle Einstellungen dieser Kategorie projektbezogen aufgelöst werden. Wird ausgewählt, dass das Zielprojekt ein Java-Projekt sein soll, wird die *Java-Natur* hinzugefügt. Dass ein Java-Builder für das Projekt eingetragen wird, erledigt das bereits vorgestellte JDT-Plugin automatisch. So wird es auch im Abschnitt 4.2 beschrieben.

Soviel zu dem, was der Benutzer sehen kann. Um einen Wizard in den Workbench einzugliedern, muss eine Erweiterung definiert werden. Wizards werden vom Workbench aufgerufen. Das bedeutet auch, dass es erschwert ist diese UI-Komponenten durch automatisierte Tests zu kontrollieren (siehe Abschnitt 4.7). Solche Wizards sind generell von verschieden Orten aus aufrufbar, je nachdem, wo Erweiterungspunkte definiert sind. Die Wizards laufen dann im gleichen Thread wie der Workbench ab. Das ermöglicht die Anzeige von Seiten, ohne den Schutzmechanismus des Workbench, wie in Abschnitt 4.1.2 beschrieben, umgehen zu müssen. Alle weiteren Aktivitäten des Workbench sind unterbunden, so dass Änderungen am Workspace während der Bearbeitungszeit der Wizards nicht von anderer Stelle aus geschehen.

[1] Vergleiche Abschnitt 2.5.1



(a) Seite I (b) Seite II

Abbildung 5.2: Zwei Seiten des *New Asset Project Wizard*

Der Wizard sammelt auf seinen zwei Seiten Parameter. Diese vom Benutzer gemachten Einstellungen werden dann zur Konfigurationen einer bestimmte Aktion genutzt. Die Aktion `AssetProjectAction` wird dann aufgerufen. Die eigentliche Tätigkeit wurde aus dem Wizard ausgelagert. Das hat den Vorteil, dass diese Aktion auch von anderer Stelle aus ausgeführt werden kann. Eine Taste in der eclipse Symbolleiste *Toolbar* macht genau das. Nach Drücken dieser Taste wird ein Modell-Projekt mit automatischer Namensgebung und Standardeinstellungen erstellt.

5.3 Interaktive Anleitung zum Generatorbau: Cheat Sheet

Eclipse stellt *Cheat Sheets* (interaktive Anleitungen) zur Verfügung, um Benutzer durch den Entwicklungsprozess von Anwendungen zu führen[2]. Jede Anleitung hat zum Ziel, ein gelungenes Ende eines Entwicklungsprozesses zu erreichen. Die Unterteilung in mehrere Abschnitte ist üblich. Solche Abschnitte werden dann vom Benutzer der Reihe nach geöffnet, wobei die Einleitung initial bereitsteht. Für einen bestehenden Workspace wird protokolliert, wie weit eine Anleitung geöffnet wurde, um den Fortschritt eines Entwicklungsprozesses über Sitzungen hinaus nachvollziehen zu können. Der Anwender kann dann, wenn er das nächste Mal diese Seite öffnet, dort weitermachen, wo er letztes Mal aufgehört hat. Es handelt sich bei diesen Anleitungen nicht nur um reine Texte. Befehle aus Eclipse können durch Anklicken angesprochen werden, wie zum Beispiel das Initialisieren eines Wizards, das Erstellen eines Projektes oder der Wechsel zu einer bestimmten Perspektive. Auch Verweise auf bestimmte weiterführende Erklärungen der eclipse-Hilfe können eingebunden sein.

[2]Hauptquelle: „Working with cheat sheets“ aus [16]

```
1 <!-- ================== cheatsheet ================== -->
  <extension id="generator.cheatsheet"
    name="cheat-sheet for building new generators"
    point="org.eclipse.ui.cheatsheets.cheatSheetContent">
5   <cheatsheet id="compilerCore.generatorCheatSheet"
      contentFile="help/cheat/generatorCheat.xml"
      name="building cocoma-generator">
      <description>
          CheatSheet for building new CoCoMa-Generators
10    </description>
    </cheatsheet>
  </extension>
```

Kode-Beispiel 5.2: Einbindung einer interaktiven Anleitung

5.3.1 Erstellen eines Cheat Sheet

Bekannterweise wird eclipse dadurch erweitert, dass vorhandene *extension-point*s benutzt werden. Das Kode-Beispiel 5.2 zeigt eine solche Erweiterung. In der in Zeile 6 wird jene XML-Datei beschrieben, die den Inhalt des Cheat Sheet widerspiegelt[3]. Eine solche XML-Datei besteht nun im Wesentlichen aus zwei Arten von Elementen: zuerst die Einleitung, dann die einzelnen Arbeitsschritte[4]. Ein Arbeitsschritt hat vier Elemente: eine Überschrift, einen Verweis auf ein Hilfe-Thema aus der eclipse-Hilfe, und er kann auch eine automatische Aktion anbieten. Dann folgt der den Arbeitsschritt beschreibende Text. Die Zeilen 8-10 beschreiben, was im Menü des Workbenchs angezeigt wird, bevor das eigentliche Cheat Sheet geladen wird.

5.3.2 Ausführbare Aktionen in Cheat Sheets

An diese Stelle soll nun gezeigt werden, wie solch ein Arbeitsschritt aussehen kann. In Kode-Beispiel 5.3 ist der Aufbau gezeigt. Man beachte, dass Cheat Sheets im UI-Thread ablaufen, alle Plugins und realisierende Klassen sind bekannt. Wird nun eine *action* definiert, so wird zuerst das Plugin, welches die realisierende Klasse birgt, benannt. Daraufhin die Klasse selbst, die die Schnittstelle `IAction` zu implementieren hat. Optional auch `ICheatSheetAction`, nämlich dann, wenn auch Parameter mit übergeben werden. Wie in Zeile 6 von Kode-Beispiel 5.3 zu sehen ist (die Kode-Zeile 5 umschließt hier zwei gedruckte Zeilen), wird hier ein Parameter mit übergeben. Dazu muss ein Attribut mit der Bezeichnung: `paramN` hinzugefügt werden[5]. Einen näheren Einblick über Aktionen im Allgemeinen bekommt man im Abschnitt 5.1.

[3] Der Pfad ist relativ zum Plugin-Installationspfad angegeben.
[4] *intro* und *item*
[5] N wird fortlaufend durch Natürliche Zahlen ohne Null substituiert.

```
<item title="Create your Generator class"
   href="/org.eclipse.jdt.doc.user/gettingStarted/qs-9.htm"
   skip="true">
   <action pluginId="de.tuhh.sts.ac.compilerCore"
         class="de.tuhh.sts.ac.compilerCore.ui.help.
            moduleDevelopment.CreateEmptyGeneratorAction"
         param1="generatedGenerator"/>
   <description>
         Textual description...
   </description>
</item>
```

Kode-Beispiel 5.3: Aktionen in Cheat Sheets

5.4 Die eclipse Online-Hilfe

Eclipse beinhaltet einen eigenen Browser. Werden Dokumentationen referenziert, kann dieser geöffnet werden und versorgt sowohl Benutzer wie auch Entwickler mit Informationen. In wie weit besonders Entwickler von dieser integrierten Hilfe profitieren, wird im Analyse-Kapitel 2 deutlich.

Hilfedokumente werden in eclipse über den Erweiterungsmechanismus eingebunden. Das bedeutet auch, dass Komponenten-Entwickler der Asset-Hilfe Dokumente hinzufügen können. Andere Dokumentationen, auch entfernte, die nur über einen Webserver erreicht werden, können referenziert werden.

Die Hilfe für Modell-Ersteller und Komponenten-Entwickler ist in eben diese beiden Bereiche getrennt. Der Modell-Definierer soll nicht mit den Implementierungsdetails verwirrt werden. Die Dokumentationen sind sämtlich in englischer Sprache verfasst, da diese Sprache in der SW-Entwicklung allgemein üblich ist.

5.5 Inbetriebnahme

Der einheitliche Installations- und Updatemechanismus von eclipse wird für die Installation dieses Plugins verwendet. Dazu wird eine URL eingegeben, unter der dann ein sogenanntes *feature* angeboten wird. Ein *feature* ist eine Ansammlung von Plugins. In diesem Fall wird sowohl das Rahmenwerk Plus als auch Beispiel-Generatoren unter einer Internet-Adresse angeboten. Letztgenannte Generatoren sollen die Benutzung der vorhandenen Mechanismen demonstrieren. Die Implementation eines Parameter-Lieferanten, die Publikationen einer Einstellungsseite, etc. werden so beispielhaft aufgezeigt.

Eclipse benutzt die JVM und verspricht, auf verschiedenen Betriebssystemen lauffähig zu sein. Die Installation eines Plugins ist von verschiedenen Rahmenbedingungen beschränkt. So funktioniert das vorliegende Plugin nur, wenn die Java-Version 1.5 installiert ist. Die Beta-Version von Java 1.5 verursacht Fehler. Die Installation setzt außerdem eine SDK-Version 3.0.0 von eclipse voraus. Das JDT-Plugin wird an verschiedenen Stellen mitbenutzt (siehe Abschnitte 3.4 und 5.2).

Ein wichtiges Kriterium im Bezug auf die Qualität von Software ist die Benutzerakzeptanz. In der Phase der Inbetriebnahme wird deutlich, inwieweit die angebotene Software den Ansprüchen der Anwender entspricht.

Die Plugin-Version im Alpha-Stadium berücksichtigte beispielsweise nicht, dass die Groß- und Kleinschreibung in einigen Dateisystemen Beachtung findet. Bei Tests auf einem Windows-Dateisystem konnte kein Fehler gefunden werden. Während der Inbetriebnahme auf einem Linux-System wurden Ressourcen nicht gefunden. Das schlägt sich negativ auf die Benutzerakzeptanz nieder.

Programmierer treten auf unterschiedliche Art an ein neues Werkzeug heran. Es ist *nicht* generell davon auszugehen, dass Benutzer nach einigen Fehlversuchen entsprechende Dokumentationen lesen. Vielmehr sinkt die Akzeptanz des Produktes, wenn Vorgänge nicht für sie intuitiv funktionieren. Da gewohnte Vorgehensweisen von Benutzer zu Benutzer unterschiedlich sind, und sich untereinander teilweise widersprechen, kann nur versucht werden, einem möglichst großen Kreis an Benutzern zu genügen.

Das neue Plugin schreibt dem Benutzer eine bestimmt Art der Benutzung vor. Um aus einem Modell ein System erstellen zu können, ist es beispielsweise notwendig, ein *Asset-Definition Projekt* zu erstellen, welches das Modell enthält. Wie im Abschnitt über das Builder-Konzept 3.5 deutlich geworden ist, macht das auch aus Sicht der Implementation Sinn, den Ablauf so zu gestalten. Benutzer, die diese Hintergrundinformationen nicht haben, mögen vielleicht versuchen, eine Modelldatei losgelöst von einem solchen Projekt zu übersetzen, und reagieren ablehnend, weil die Übersetzung so nicht funktioniert. Die Bereitschaft, seine eigene Arbeitsweise einem vorhandenen Werkzeug anzupassen, ist von Benutzer zu Benutzer sehr unterschiedlich.

Teile der Funktionalität werden vom Anwender auch entsprechend der eigenen Aufgabe beurteilt. So möchte der eine Benutzer einen Generator schreiben, der Java-Quellkode erzeugt, und ein anderer solchen, der beispielsweise Ausdrücke in einer SQL-Syntax generiert. Ersterer möchte nun im Wizard für ein neues *Asset-Definition Projekt* angeben, dass ein Java-Projekt als Generierungsziel automatisch erzeugt wird, der zweite möchte lieber eine andere Art von Projekt haben. Es ist leider nicht machbar, für alle potentiellen Anwendungsmöglichkeiten hier Unterstützung zu bieten. Die Auswahl ist sogar sehr begrenzt, damit die Funktionalität nicht zu Lasten der Übersichtlichkeit geschaffen wird. Ist die für den speziellen Anwender wichtige Funktionalität nicht vorhanden, so wird die Vorgehensweise mit dem Werkzeug teilweise als umständlich empfunden.

Dass Plugins durch den Update-Mechanismus eclipses problemlos erneuert werden können, schlägt sich positiv auf die Benutzerakzeptanz nieder. Es hat sich hierbei als hilfreich herausgestellt, die Benutzer über neue Versionen und die Beseitigung von Fehlern zu informieren. Damit die Benutzer in der Integrationsphase nicht das Interesse verlieren, ist eine schnelle Reaktionszeit von der Meldungen von Fehlern bis zur Korrektur wichtig.

Das Plugin wird von Komponenten-Entwicklern genutzt und wird weiterentwickelt werden.

KAPITEL 6

Schluss

In diesem Kapitel wird festgestellt, dass das entstandene Plugin den in der Analyse dargelegten Bedürfnissen entspricht oder zumindest nicht erfüllte zukünftig erweitert werden können. Es hat sich als lohnende Entscheidung erwiesen, eclipse als Integrationsplattform zu wählen.

6.1 Zusammenfassung

Für die Erweiterung des bestehenden Rahmenwerks werden Benutzerrollen von Personen, die mit dem Modell-Übersetzer arbeiten (siehe Abschnitt 2.5) herausgestellt. Diese Rollen, sowie aus deren Arbeitsprozessen resultierende Anforderungen an notwendige Programmartefakte, führen zum Rahmenwerk Plus.

Das konzeptionelle Design geht auf einen Ablauf des Rahmenwerks mit Beobachtern ein. Diese Beobachter (siehe Abschnitt 3.1) ermöglichen eine lose Kopplung der Komponenten zum Rahmenwerk, so dass das Rahmenwerk austauschbar bleibt. Im Abschnitt 3.2 wird eine Verallgemeinerung der Parameterbeschaffung für das Ansprechen von Generatoren vorgestellt. Darin enthalten ist ein Konzept zur Einbindung von grafischen Einstellungsseiten. Die Lebensdauer von Prozessen und die Zwischenspeicherung von Instanzen und Parametern wird in dem Abschnitt 3.3 behandelt.

Anschließend wird vom konzeptionellen Design, nach Evaluation möglicher Technologien, auf eine Architektur auf Basis der Entwicklungsumgebung eclipse übergegangen. Eclipse stellt eigene Konzepte vor (siehe Abschnitte 3.4 bis 3.8), die die Einbindung eines Rahmenwerks ermöglichen.

Die Integration der Komponenten und deren Ablaufsteuerung ist vollständig durch ein eclipse-Plugin realisiert. Grafische Elemente wie Dialoge, Ansichten und Einstellungsseiten erlauben eine Interaktion des Rahmenwerks und der Komponenten mit dem Benutzer.

Im Abschnitt 4.1 werden die verschiedenen Funktionsgruppen des Plugins aufgezeigt. Das RAHMENWERK PLUS ermöglicht die Integration von Komponenten der neuen erweiterten Form, als auch die von bereits bestehenden.

Mit der Integration des Rahmenwerks in eclipse ist es möglich geworden, Komponenten innerhalb **eines** Programms programmieren, einbinden und testen zu können. Während der Programmierung einer Komponente kann der Quellkode automatisch in ein Zweitsystem eclipses portiert und zur Ausführung gebracht werden (siehe Abschnitt 3.9). Von der Komponente dann generierte Quellkode-Module können automatisch von entsprechenden Kompilierern weiterverarbeitet werden, so dass Fehler in der Generierung unter Umständen gleich offensichtlich werden.

Eine interne Laufzeitbeobachtung (siehe Abschnitt 3.8) ermöglicht es den Entwicklern nachzuvollziehen, was innerhalb des RAHMENWERKS geschieht. Zusätzlich hilft eine angepasste Umgebung für automatisierte Tests (siehe Abschnitt 4.7) bei der Eingrenzung von Fehlern in der Komponenten-Entwicklung.

Es werden verschiedene Arten der Hilfestellung für die Benutzer realisiert (siehe Kapitel 5). Dabei wird darauf geachtet, möglichst viele Konzepte zur Benutzerführung zu verwenden. Der entwickelte Prototyp wird bereits von Komponenten-Entwicklern verwendet und wird weiterentwickelt werden. In dem Abschnitt 5.5 wird beschrieben, welche Erkenntnisse in Bezug auf die Benutzerakzeptanz durch die Inbetriebnahme gewonnen wurden.

6.2 Bewertung

Im Abschnitt 2.6 über die Anforderungen an das System wird wünschenswerte Funktionalität beschrieben. Zur Erinnerung werden die Forderungen hier nochmals aufgelistet und anschließend wird dazu Stellung genommen.

Forderungen von **Asset-Modell Erstellern**:

- Übersichtliche Darstellung von Dokumentationen
- Editoren, die die Modell-Definition erleichtern
- Anzeige von Fehlern im Editor
- Grafische Editoren
- Einfache Konfigurationsauswahl
- Installationsunterstützung der Komponenten
- Übersetzung auf Knopfdruck oder gar automatisch im Hintergrund
- Überblick über den Prozessfortschritt
- Abbrechen der laufenden Systemerzeugung
- Voreinstellungen und Ablauf der Weiterverarbeitung

- Einfaches Starten des fertigen Systems

Forderungen von **Kompilierer-Komponenten Entwicklern**:

- Einheitliche Entwicklungs-, Test-, und Ablaufumgebung
- Persistenzmechnismus von Einstellungen mit grafischer Oberfläche
- Einbindung von Dokumentationen
- Einfache Integration eines Generator-Prototyps
- Kommunikation über Ansichten und Dialoge
- Unterstützung für Entwicklung in der Gruppe
- Publikation durch Updatemechanismus mit Versionierung

Die Forderung nach einer **übersichtlichen Darstellung von Dokumentationen** ist durch eine erweiterbare, in eclipse integrierte, Hilfe erfüllt (siehe Abschnitt 5.4). Die Hilfedateien sind nicht im Detail ausgearbeitet, bieten jedoch einen Einstieg in die Arbeit mit dem Plugin. Komponenten-Entwickler können diese Hilfe durch eigene Beiträge erweitern. Dadurch wird gewährleistet, dass Dokumentationen einheitlich dargestellt werden können.

Editoren, die die Modell-Definition erleichtern, sind noch nicht enthalten. Eclipse bietet jedoch die Möglichkeit, nachträglich Editoren integrieren zu können. Der Plugin-Mechanismus erleichtert die Integration von vorhandenen Parsern im Editor. Dadurch kann schon während des Tippens eine Fehleranalyse stattfinden und Fehler können früh angezeigt werden.

Die **Anzeige von Fehlern im Editor** ist bereits integriert (siehe Abschnitt 4.3), es werden jedoch nur wenige Fehler erkannt. Das liegt an dem frühen Stadium der Parser-Entwicklung. Wird ein Fehler gefunden endet der Vorgang des Parsens. Die vom Parser zurückgelieferten Fehlermeldungen sind nicht immer aussagekräftig genug, um einen Fehler im Quellkode anzeigen zu können.

Grafische Editoren sind, wie andere Editoren auch, nachträglich erweiterbar.

Durch die Einstellungsdialoge zur Generierungsphase ist eine **einfache Konfigurationsauswahl** realisiert. Die Abb. 4.15 auf Seite 80 zeigt eine entsprechende Ansicht.

Die **Installationsunterstützung der Komponenten** ist komfortabel in eclipse integriert. Generatoren, Analyse-Komponenten und das Rahmenwerk Plus-Plugin werden über den eclipse-Installationsmechnismus installiert und aktualisiert. Dem Anwender wird eine Übersicht über die Versionen der Komponenten gegeben.

Bezüglich der **Übersetzung auf Knopfdruck oder gar automatischer im Hintergrund** wird der Builder-Mechanismus eclipses genutzt (siehe Abschnitt 3.5). Der Benutzer hat die Wahl, ob die Systemerzeugung automatisch

nach Speicherung der Asset-Modell-Datei oder auf Anfrage hin ausgeführt werden soll. Der Vorgang findet im Hintergrund statt, so dass der Benutzer weiterarbeiten kann. Unter Umständen kann die Interaktion des Benutzers mit den Komponenten zur Übersetzungszeit notwendig sein. Dies geschieht dann über grafische Dialoge (siehe Abschnitt 4.5.1).

Der **Überblick über den Prozessfortschritt** wird durch Beobachter realisiert und durch eclipse-Komponenten visualisiert. Ein Fortschrittsbalken zeigt dem Anwender an, wie weit die Systemerzeugung durchlaufen ist und welche Komponente zum jeweiligen Zeitpunkt gestartet wurde. Der Benutzer kann das **Abbrechen der laufenden Systemerzeugung** beantragen. Zwischen dem Start einzelner Komponenten kann immer abgebrochen werden. Innerhalb eines Generators nur, wenn dieser in ein Plugin mit entsprechender Unterstützung eingebunden ist.

Bezüglich der **Voreinstellungen und** dem **Ablauf der Weiterverarbeitung** bietet die KOMPILIERER-UMGEBUNG komfortable Einstellungsmöglichkeiten. Der Wizard zur Erstellung eines neuen Asset-Projektes (siehe Abschnitt 5.2) erlaubt zusätzlich die automatische Erstellung eines Ausgabe-Projektes. Java Quellkode kann z.B. nach seiner Generierung durch die ÜBERSETZER-UMGEBUNG automatisch vom Java-Kompilierer weiterverarbeitet werden. Die grafischen Einstellungsseiten geben dem Benutzer die Möglichkeit, den KOMPILIERER übersichtlich zu konfigurieren. Hinzuzufügende Komponenten können dynamisch Seiten propagieren (siehe Abschnitte 3.7 und 4.6).

Das **einfache Starten des fertigen Systems** ist durch die in eclipse gegebenen Funktionen möglich. Z.B. kann durch die Erstellung von Ant-Skripten (siehe Abschnitt 2.7.1) dieser Vorgang automatisiert werden. Das muss jedoch je nach Zielsystem ermöglicht werden. Eine Erweiterung durch Generator-Entwickler ist möglich, bisher jedoch noch nicht realisiert.

Zu den Konzepten von eclipse gehört es, eine **Einheitliche Entwicklungs-, Test-, und Ablaufumgebung** zu bieten. Komponenten-Projekte einer eclipse-Instanz werden zu Plugins einer zweiten (siehe Abschnitt 3.9). Die Entwicklung von Komponenten wird durch eclipse durch seine Funktionen unterstützt (siehe Abschnitt 2.7).

Die Abschnitte 3.7 und 4.6 stellen einen **Persistenzmechnismus von Einstellungen mit grafischer Oberfläche** vor. Diese Einstellungsseiten sind für Erweiterungen offen. Komponenten dritter können unkompliziert Einstellungen publizieren und auslesen.

Die **Einbindung von Dokumentationen** ist, wie oben beschrieben, geschehen. Schnittstellen zum originalen Rahmenwerk und zum RAHMENWERK PLUS sind dokumentiert. Komponenten-Entwickler können die Dokumentationen vervollständigen.

Eine **einfache Integration eines Generator-Prototyps** wird durch ein *Cheat Sheet* angeleitet (siehe Abschnitt 5.3). Ein Generator-Plugin wird einfach und schnell erzeugt. Dabei vollzieht der Entwickler die einzelnen Schritte nach und kann sie später auch ohne das *Cheat Sheet* durchführen.

Durch eine Erweiterung von Prozess-Beobachtern kann eine **Kommunika-**

tion über Ansichten und Dialoge zwischen Komponenten und Benutzer ermöglicht werden (siehe Abschnitte 3.1 und 4.5.2). Beobachter können Generator-Meldungen in beliebigen Ansichten visualisieren. Eine Anpassung der Generator-Schnittstelle wäre in Zukunft jedoch vorteilhaft, um diesen Vorgang durch reichere Benachrichtigungen der Beobachter zu vereinfachen.

Eine **Unterstützung für Entwicklung in der Gruppe** ist in eclipse enthalten. Hilfreich ist das auch auch für Plugins, die Klassen für automatisierte Tests enthalten (siehe Abschnitt 4.7).

Die **Publikation durch** einen **Updatemechanismus mit Versionierung** ist ebenfalls durch eclipse gegeben. Das PDE-Plugin hilft bei der Publikation von Plugins durch *features* (siehe Abschnitt 5.5).

Anhand der Bewertung der Umsetzung der einzelnen Anforderungen wird deutlich, inwiefern das Plugin hilft. Oben werden die Nachteile jedoch nicht aufgeführt, die man sich mit diesem Plugin 'erkauft'.

Zwei Dinge fallen hier besonders ins Auge.

Eclipse belegt im Vergleich zu dem originalen Rahmenwerk mehr Arbeitsspeicher zur Laufzeit. Da eclipse von Kompilierer-Komponenten-Entwicklern zweifach gestartet wird, werden 130 + 40 MB an Arbeitsspeicher benötigt.

Die Arbeitsweise eines Benutzers, sei er Entwickler oder Modell-Ersteller, hat sich eclipse und dem Plugin anzupassen. Möchte man beispielsweise eine Ressource dem eclipse Workspace hinzufügen, so sollte man den 'import-Wizard' von eclipse nutzen. Kopiert man in dem Dateibaum des Betriebssystems und wechselt dann zu eclipse, ist die Ressource dort nicht sichtbar, solange bis der Workspace aktualisiert wurde. Es bedarf also der Eingewöhnung in eclipse und in das neue Rahmenwerk Plus.

Einiges an Funktionalität konnte bislang nicht in allen Ausprägungen realisiert werden. Das Rahmenwerk Plugin ist jedoch so gestaltet, dass bewusst viele Möglichkeiten von eclipse benutzt worden sind, um die Anwendung darzulegen. Eine Erweiterung und Vervollständigung sollte vorhandene Komponenten als Vorbild nehmen können. Modell-Ersteller werden beispielsweise durch einen Wizard bei der Erzeugung ihres Projektes unterstützt, Komponenten-Entwickler durch ein Cheat Sheet.

Eclipse hat durch seine gute Erweiterbarkeit und besonders durch sein *Builder Konzept* überzeugt und sich somit als eine gute Wahl erwiesen. Bereits in eclipse enthaltene Funktionalität vervollständigt das Bild über die herausgestellten Anforderungen hinaus.

Alles in allem kann von einer gelungenen Entwicklung gesprochen werden, auch wenn der Kreis der Benutzer und somit der Tester im aktuellen alpha-Stadium noch sehr klein ist.

6.3 Ausblick

Im Folgenden werden einige Punkte genannt, die als eigenständige zukünftige Projekte verstanden werden können.

- **Entwicklung eines Modell-Editors.** Funktionen wie Autovervollständigung, Syntaxhervorhebung, automatische Fehlerkorrektur, etc. könnte ein speziell an die Asset-Definitionsprache angepasster Editor bereitstellen. Die Erstellung eines Editors für eclipse wird in [7] beschrieben.

  Auch ein grafischer Editor wäre denkbar. Eine grafische Repräsentation der Asset-Definitionssprache muss dafür erarbeitet werden. Nach Beendigung dieser Arbeit steht die Aufgabe aus, diese Darstellung durch einen Modell-Editor in eclipse einzubinden. Das *Graphical Editor Framework*, eine Entwicklung der eclipse-Gemeinschaft, kann dazu Verwendung finden (siehe [12]).

- **Erweiterung der Wizards zur Erstellung von speziellen Dateien,** wie Konfigurationsdateien oder Parameter-Lieferanten oder ganzen Kompilierer-Komponenten-Plugins. Der PDE „New-Project-Wizard“ mit seinen vielfältigen Optionen ist ein Beispiel dafür.

- **Ein Editor zur Bearbeitung von Konfigurationen.** Man könnte den Manifest-Editor des PDE als Vorbild nehmen (siehe auch [5] und [12]). Es sollte möglich sein, aufgrund der Parameteranforderungen eines Generators, Auswahlfelder bereitstellen zu können. Damit könnte sichergestellt werden, dass Parameter vor der Benutzung kontrolliert werden.

- **Ein Werkzeug zum Importieren verfeinerbarer Modelldateien.** Bislang muss man sich Modelldateien in das Asset-Projekt importieren. Eine Übersicht oder einen Zugriff auf entfernte Bibliotheken gibt es noch nicht.

- **Erstellung von Tutorials.** Es sind viele Arbeitsprozesse vorgestellt worden. Anwender sollten durch entsprechende Cheat Sheets angeleitet werden.

- **Vervollständigung der Hilfsdokumente u. Internationalisierung.** Bisher sind Hilfsdokumente nur in englischer Sprache verfasst. Eine Internationalisierung des Plugins und der Hilfe kann leicht bewerkstelligt werden, da Zeichenketten im UI, der Sprache entsprechend, aufgelöst werden.

- **Auto-Vervollständigung und -Fehlerkorrektur in der Asset-Definitionssprache.** Um den Vorgang der Modell-Erstellung zu unterstützen, könnte ein entsprechender Editor, auch textbasiert, erstellt werden. Die lexikalische- und syntaktische Analyse kann in den Editor integriert werden.

- **Erweiterung der automatisierten Tests zur Eingrenzung von Fehlern bei der Erstellung von Kompilierer-Komponenten.** Momentan können böswillige oder nicht den Konventionen entsprechende Komponenten den Übersetzer lahmlegen. Automatisierte Tests können dazu benutzt werden, solche Schwachstellen zu entdecken.

- **Verbesserung der Analyse durch Erweiterung des Parsers.** Dies ist eine Aufgabe aus dem Bereich der klassischen Kompilierer-Entwicklung. Die semantische Analyse sollte umfassender werden, um Fehler bei der Systemerzeugung vermeiden zu können. Außerdem sollten die Fehlermeldungen, die der Parser liefert, detailliertere Informationen über den Ort und die Art des Fehlers beinhalten. Beim Auffinden eines Fehlers bricht der Parser seine Arbeit ab. Somit wird nur *ein* Fehler ausgegeben. Der Parser ist dahingehend zu erweitern, dass Fehler toleriert werden können und der Parse-Vorgang weitergeführt werden kann.

- **Nutzung der inkrementellen Übersetzung.** Die inkrementelle Übersetzung (vorgestellt im Abschnitt 3.5.2) wird von dem Java-Kompilierer erfolgreich praktiziert. Es ist erst einmal festzustellen, ob eine solche Vorgehensweise in diesem Fall der Systemerzeugung überhaupt möglich ist. Falls dem so ist, müssen die Generatoren der inkrementellen Generierung angepasst werden.

Das COCoMa-Gesamtprojekt umfasst noch viel mehr Aufgaben. Die soeben aufgelisteten sind aus der vorliegenden Arbeit heraus entstanden und tragen dazu bei, ein vollständiges Bild einer Systemerzeugungs-Umgebung zu entwerfen.

Literaturverzeichnis

[1] Hans-Werner Sehring: *Konzeptorientierte Inhaltsverwaltung - Modell, Systemarchitektur und Prototypen.* Diss., TU Hamburg Harburg, http://www.sts.tu-harburg.de, 2004.

[2] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Entwurfsmuster.* Programmer's Choice. Addison Wesley, Deutsche Ausgabe 2004.

[3] Helmut Balzert: *Lehrbuch der Software-Technik.* Spektrum Akademischer Verlag, 1998.

[4] G. Booch, I. Jacobson, J. Rumbaugh: *The Unified Modelling Language User Guide.* Addison Wesley, 1999.

[5] J. D'Anjou, S. Fairbrother, D. Kehn, J. Kellerman, P. McCarthy, S. Shavor: *The Java Developer's Guide to Eclipse.* Addison Wesley, 2003.

[6] Berthold Daum: *Java-Entwicklung mit Eclipse 2.* dpunkt.verlag, 2003.

[7] Erich Gamma, Kent Beck: *Contribution to eclipse: Principles, Patterns and Plug-ins.* the eclipse series. Addison Wesley, 2004.

[8] F. Budinsky, R. Ellersick, T. J. Grose, Ed Merks, D. Steinberg: *Eclipse Modeling Framework.* the eclipse series. Addison Wesley, 2003.

[9] Peter Friese: *Incremental Project Builder in Eclipse.* iX Magazin für professionelle Informationstechnik, Ausgabe 8, August 2004.

[10] Frank Gerhardt, Christian Wege: *Eclipse als Basis für Rich-Client-Anwendungen.* iX Magazin für professionelle Informationstechnik, Ausgabe 7, Juli 2004.

[11] OSGi Alliance: *OSGi Specification.* http://www.osgi.org.

[12] eclipse community: *eclipse home.* http://www.eclipse.org/.

[13] ECLIPSE NEWSGROUPS: *newsgroup*. http://eclipse.org/newsgroups/.

[14] BERTHOLD DAUM: *Mutatis mutandis - Using Preference Pages as Property Pages*. Eclipse Corner Article, Oktober 2003.

[15] RYAN COOPER: *Simplifying Preference Pages with Field Editors*. Eclipse Corner Article, August 2002.

[16] IBM OTI: *Eclipse Workbench User Guide*.

[17] KENT BECK: *Extreme Programming*. Programmer's Choice. Addison Wesley, 2003.

[18] KENT BECK: *Test-Driven Development By Example*. Signature Series. Addison Wesley, 2003.

[19] *Alliance for Agile Programming*. http://www.agilealliance.org.

[20] R. FRANCE, B. RUMPE, D. TURK: *Limitations of Agile Software Processes*. Conference on eXtreme Programming and Agile (XP 2002), 2002.

[21] R. W. FLOYD: *Assigning meanings to programs*. Technischer Bericht, 1967.

[22] C. A. R. HOARE: *An axiomatic basis for computer programming*. Technischer Bericht, 1969.

[23] APACHE PROJECT: *Apache Ant*. http://ant.apache.org/.

[24] JETBRAINS: *IntelliJ IDEA*. http://www.jetbrains.com/idea/.

[25] SUN MIRCROSYSTEMS: *NetBeans*. http://www.netbeans.org/.

[26] WORLD WIDE WEB CONSORTIUM: *XML und HTML Spezifikation*. http://www.w3.org/.

ANHANG A

Beispielansichten

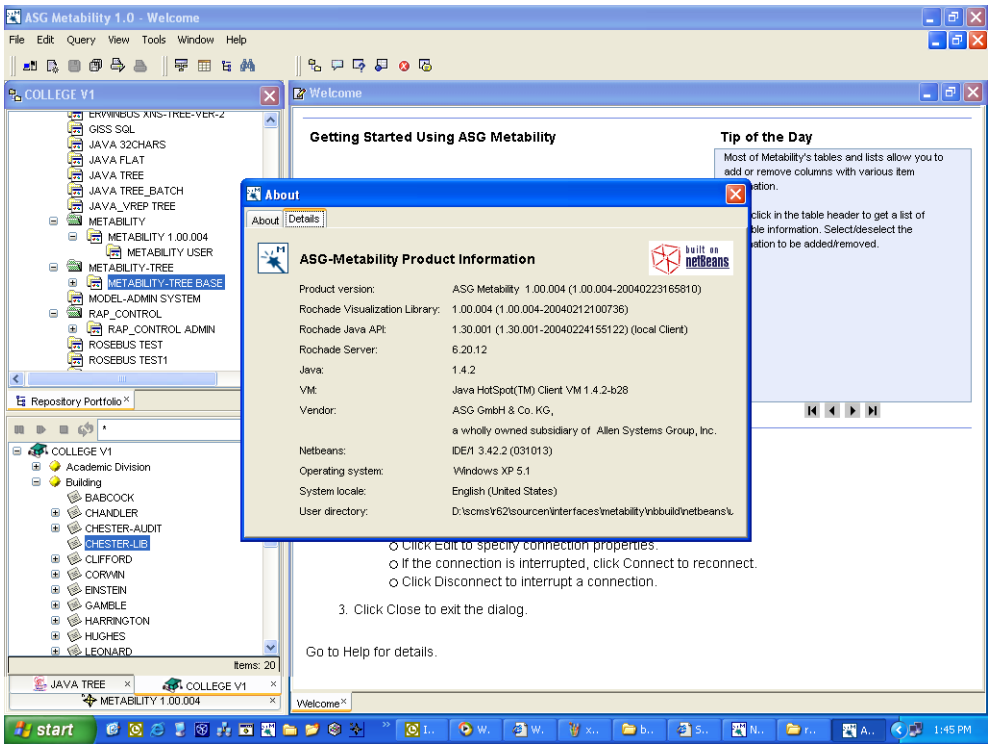


Abbildung A.1: Beispielansicht von NetBeans

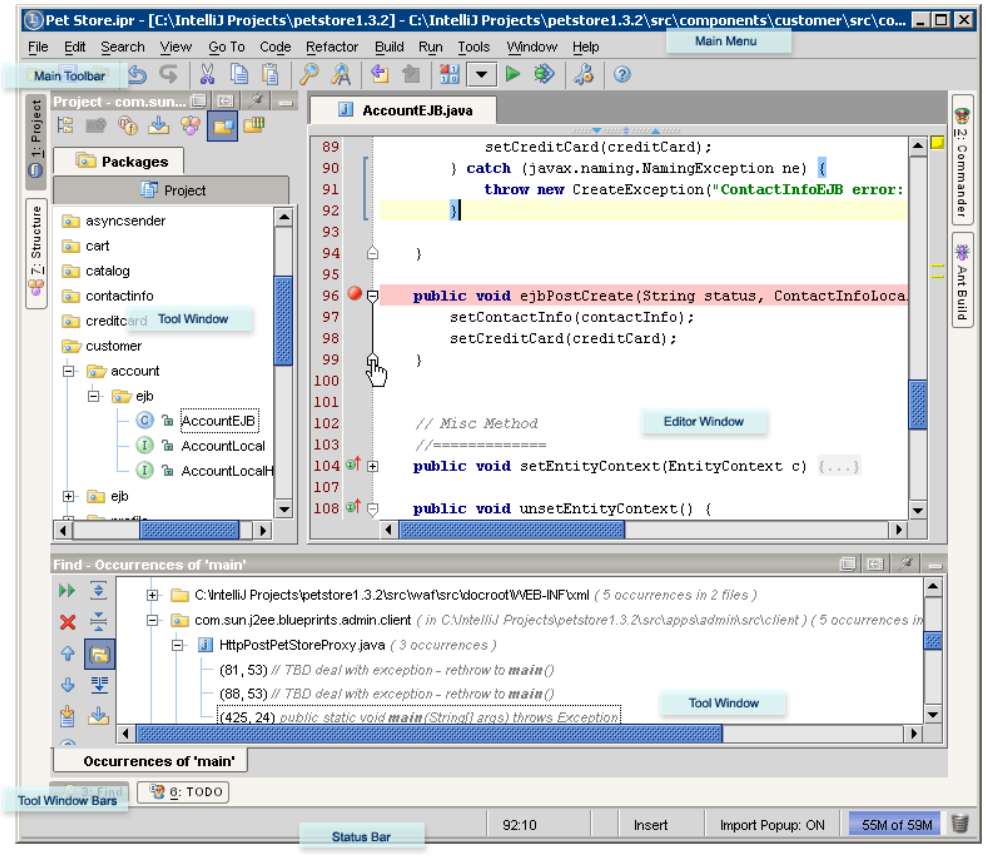


Abbildung A.2: Beispielansicht von IntelliJ IDEA

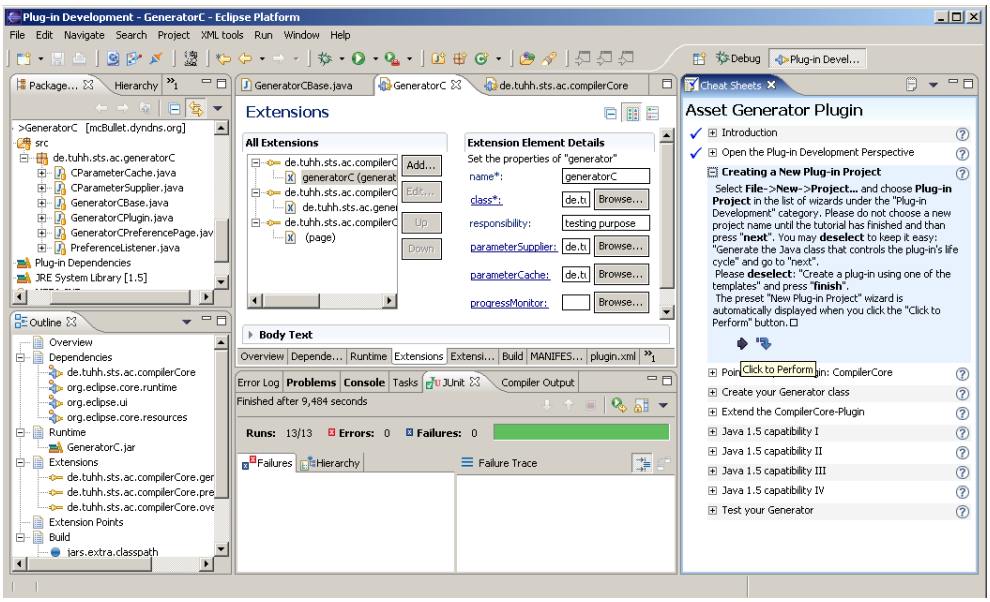


Abbildung A.3: Eclipse mit Übersetzer-Komponenten

ANHANG B

Quellkodeausschnitte

```
1 import org.eclipse.jface.dialogs.MessageDialog;
  import org.eclipse.swt.widgets.Display;

  /**
5  * Informs the user with an error-message window
   * and calls {@link #handleException(String, String,
     Exception)}.
   * @param title title of the window
   * @param message error message to be diplayed
   * @param exception may be null, otherwise its contents will
     be added
10 */
  public static void handleInformUser(String title, String
     message, Exception exception){
          if(exception == null) exception = new Exception();
          final String finalTitle = title;
          final String finalMessage = message;
15        final String finalException = exception.toString();
          Display.getCurrent().syncExec(new Runnable(){
                  public void run() {
                          MessageDialog.openError(null,
                              finalTitle, finalMessage+
                              finalException);
                  }
20        });
          handleException(title, message, exception);
  }
```

Kode-Beispiel B.1: Präsentation von Ausnahmen als *Message-Dialog*: `CompilerCorePlugin.java`

```
1 import java.util.ArrayList;

  import junit.framework.TestCase;
  import de.tuhh.sts.ac.compilerCore.preferences.generic.
      TreeEditor;
5 import de.tuhh.sts.ac.test.compilerCore.
      CompilerCoreTestPlugin;

  public class ParseNamesTest extends TestCase {
    public void testParseNames() throws Exception{
          if(CompilerCoreTestPlugin.traceInvokation)
10          System.out.println("ParseNamesTest.testParseNames()
                ");
          //TreeEditor.parseString
          // null
          assertNotNull("isNull T1",
            TreeEditor.parseString(null));
15        // ""
          assertNotNull("isNull T2",
            TreeEditor.parseString(""));
          assertEquals("size!=0 T2",
            0, TreeEditor.parseString("").size());
20        //  "aConf"+TreeEditor.SEPARATOR+" bConf
          ArrayList<String> twoNames =
            TreeEditor.parseString("aConf"+TreeEditor.SEPARATOR
                +" bConf");
          assertNotNull("isNull T3", twoNames);
          assertEquals("size!=2 T3",
25          2, twoNames.size());
          assertTrue("Doesn't contain aConf ", twoNames.
              contains("aConf"));
          assertTrue("Doesn't contain bConf ", twoNames.
              contains("bConf"));
    }
  }
```

Kode-Beispiel B.2: Test einer Methode zur Auflistung von Identifikationen: `ParseNamesTest.java`