Quelltextgestützte Softwaremigration

Diplomarbeit

Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Softwaretechnik

Verfasser:

Fabian Dittberner

Hasselkamp 33
21502 Geesthacht
E-Mail: fabian.dittberner@dinamixx.info
Matrikelnr.: 5014104

Erstbetreuer:
Dr. Wolf-Gideon Bleek

Zweitbetreuer:
Dr. Axel Schmolitzky

Hamburg, 2007

Programmieren und Schriftstellerei
haben sehr viel gemein.
Und doch gibt es einen wesentlichen Unterschied:
Computer sind die weitaus schlimmeren Kritiker.

- F.D.

Für meine liebe
Kommilitonin und Freundin
Laalak

Danksagung

Ich möchte mich hier noch einmal bei denen bedanken, die mich bei der Ausarbeitung der Diplomarbeit unterstützt haben. Ich bedanke mich bei Dr. Wolf-Gideon Bleek, der mich als Erstbetreuer durch diese Arbeit begleitet hat. Außerdem bedanke ich mich bei Dr. Axel Schmolitzky für seine hilfreichen Tipps und Anregungen vor allem zu Beginn meiner Arbeit und für seine Zweitbetreuung. Ich möchte mich bei Alla Remfert bedanken, die im Zuge ihrer Studienarbeit das Plug-In benutzt und getestet hat. Ihre Anmerkungen und Verbesserungsvorschläge halfen sehr bei der Entwicklung der Software. Schließlich möchte ich mich noch besonders bei meinen Kollegen Holger Bohlmann und Andreas Havenstein bedanken, die meine Arbeit Korrektur lesen mussten und mir sehr dabei geholfen haben, das Gesamtbild der Arbeit abzurunden.

Inhaltsverzeichnis

Verzeichnis der Abbildungen, Tabellen und Beispiele **6**

1. Einleitung **7**

1.1. Hintergrund 7
1.2. Motivation 7
1.3. Gliederung 8
1.4. Konventionen 8

2. Grundlagen für die Migration im *WebMig*-Projekt **10**

2.1. Verschiedene Arten von Migration 10
2.2. Das *WebMig*-Projekt 11
2.3. Benutzungskontext 13
2.3.1. Die unterschiedlichen Arten von Entwicklern 13
2.3.2. Nutzung des Quelltexts als Grundlage 14
2.3.3. Die Nutzbarkeit aus einer IDE heraus 14

3. Konzepte für die Quelltextmigration im *WebMig*-Projekt **16**

3.1. Voraussetzungen 16
3.2. Zerlegung und Zuordnung des Quelltexts 16
3.3. Zuordnungsarten 21
3.4. Eindeutigkeit und Änderungsverfolgung 24

4. Die Unterstützung der Quelltextmigration als Plug-In **29**

4.1. Anforderungsermittlung 29
4.2. Benutzungsmodell 30
4.2.1. Szenarios 31
4.2.2. Darstellung der Segmentierungsinformationen 34
4.3. Technische Realisierung des Plug-Ins 37
4.3.1. Einbinden von Funktionalität in *Eclipse* 37
4.3.2. Umsetzung des Segmentmodells 39
4.3.3. Eindeutigkeit und Änderungsverfolgung 46
4.3.4. Umsetzung des Benutzungsmodells 49
4.3.5. Erstellen von Migrationsprojekten 51
4.3.6. Segmentieren und Zuordnen von Segmenten 54
4.3.7. Markieren von Segmenten als obsolet 59
4.3.8. Analyse des Migrationsfortschritts 60
4.3.9. Umsetzung zusätzlicher Funktionen 62
4.4. Testen der Anwendung 64

5. Auswertung **66**

5.1. Nutzen der Segmentierung 66
5.2. Einbettung des Plug-Ins 68

6. Zusammenfassung und Ausblick **69**

6.1. Zusammenfassung 69
6.2. Ausblick 70

Anhang A: Installation des Plug-Ins **72**

Anhang B: Benutzung des Plug-Ins **73**

Anhang C: Kommentarmarkenreferenz **75**

Literaturverzeichnis **76**

Verzeichnis der Abbildungen, Tabellen und Beispiele

Abb. 2.1.: *Eclipse*-Architektur 12
Abb. 3.1.: Beispiele für Segmente und Kopplungen als Modell für eine Zerlegung und Zuordnung des Quelltexts 17
Tab. 3.2.: Bewertung von Verbindungen und Segmenten als Modell für eine Zerlegung des Quelltexts 18
Abb. 3.3.: Zuordnung verschachtelter Segmente im a) Ursprungs- und b) Zielsystem 20
Abb. 3.4.: Die einfache Zuordnung von Ursprungs- zu Zielsegment 21
Abb. 3.5.: Zuordnung von einem Ursprungssegment zu n Zielsegmenten 22
Abb. 3.6.: Zuordnung von verteilter Ursprungsfunktionalität zu einem Zielsegment 23
Abb. 3.7.: Zuordnung verteilter Funktionalität im Ursprungssystem zu verteilter Implementierung im Zielsystem 23
Abb. 3.8.: Auswirkungen von Quelltextänderungen auf die Eindeutigkeit von Segmenten 24
Abb. 3.9.: Änderungen am Ursprungssystem bei migrierter Funktionalität 26
Bsp. 3.10.: Änderungsverfolgung in Segmenten 26
Bsp. 3.11.: Änderungen an Funktionalität ohne Änderungsverfolgung 27
Bsp. 3.12.: Änderungen an Funktionalität mit Änderungsverfolgung 27
Abb. 4.1.: Übersichtsszenario für die Migrationsanalyse 31
Abb. 4.2.: *Eclipse*-Plattform mit *extension points* 37
Abb. 4.3.: Implementierung einer IAction am Beispiel der AddSegmentAction 38
Abb. 4.4.: Die Segmentklasse 39
Abb. 4.5.: Die SegmentUtility-Schnittstelle 43
Abb. 4.6.: Die Schnittstelle für das Auslesen von Segmentierungsinformationen 43
Abb. 4.7.: Klassendiagramm der SMPEditor-Klasse 44
Bsp. 4.8.: Registrierung eines Editors (im Plug-In-Manifest) 45
Abb. 4.9.: Die Schnittstelle der Klasse SegmentIDCreationPolicy 47
Abb. 4.10.: Die Schnittstelle der Klasse MD5FingerprintCalculator 48
Tab. 4.11.: Aktionen/Ansichten und ihre Lokalisierung in *Eclipse* 51
Abb. 4.12.: Das Klassendiagramm für den NewMigrationProjectWizard 53
Abb. 4.13.: Das Klassendiagramm für die NewMigrationProjectSourceWizardPage 53
Bsp. 4.14.: Einbinden eines Wizards (im Plug-In-Manifest) 54
Abb. 4.15.: Das Klassendiagramm der unterschiedlichen *Segmentierer*-Klassen 56
Abb. 4.16.: Das Klassendiagramm der ISegmentApplier-Schnittstelle und ihrer Implementierungen 57
Bsp. 4.17.: Einbinden von Kontextaktionen (im Plug-In-Manifest) 58
Abb. 4.18.: Schnittstelle und Implementierungen von ISegmentObsoleteMarker 59
Bsp. 4.19.: Einbinden der Kontextaktion zum Markieren obsoleter Segmente (im Plug-In-Manifest) 59
Abb. 4.20.: Die Klasse MigrationProgressCollector 60
Abb. 4.21.: Die Klassen ViewPart und MigrationProgressView 61
Bsp. 4.22.: Einbinden eines *views* (im Plug-In-Manifest) 61
Abb. 4.23.: Die Schnittstelle der FingerprintUpdater-Klasse 62
Bsp. 4.24.: Einbinden von Kontextaktionen im Editor(im Plug-In-Manifest) 62
Bsp. 4.25.: Das Klassendiagramm der Schnittstelle ISourceSegmenter und der *Java*-Implementierung 63
Bsp. 4.26.: Einbinden von Kontextaktionen auf Ressourcen (im Plug-In-Manifest) 64
Abb. A.1.: Konfigurationsdialog von *Eclipse* zum Aktivieren von Menüeinträgen 72
Tab. C.1.: Übersicht über die Segmentkommentare 75

1. Einleitung

1.1. Hintergrund

Bei der Entwicklung moderner Softwaresysteme stehen neben der Umsetzung fachlicher Konzepte genauso eine lange Nutzdauer sowie eine damit einhergehende hohe Wartbarkeit im Vordergrund. Bestehende Softwaresysteme werden bei Bedarf um neue und benötigte Technologien erweitert. Alte Technologien werden abgelöst, da sie nicht mehr oder nur eingeschränkt die benötigte Funktionalität bieten oder nicht mehr vom ursprünglichen Hersteller unterstützt werden. Schließlich kann die Umstellung auf eine kostengünstigere und ökonomischere Lösung der Beweggrund für einen Wechsel auf alternative Technologien sein.

Das setzt bei vielen Systemen eine Migration der Software auf eine neue technologische Basis voraus, um das vorhandene System nicht auf einer anderen Basis komplett neu entwickeln zu müssen. Dafür wird schrittweise das vorhandene System auf die neue technologische Basis übertragen und angepasst, ohne dass die ursprüngliche Funktionalität verloren gehen darf.

Im Zuge des *WebMig*-Projekts am Fachbereich Informatik der Universität Hamburg wird derzeit eine Software zur Unterstützung verteilter Projektarbeit namens *CommSy* von seiner ursprünglichen Technologiebasis (*PHP*) auf eine neue Technologie (*Java*) migriert.

1.2. Motivation

Das *WebMig*-Projekt verfolgt einen pragmatischen Ansatz bei der Migration. Dabei wird die Migration des *CommSy* von einer Technologie in die andere Schritt für Schritt anhand des Quelltexts vorgenommen. Um die einzelnen Teile des *CommSy* in das neue System zu übertragen, werden die Anforderungen bzw. die vorhandene Funktionalität von den Entwicklern aus dem Quelltext herausgelesen und auf Basis der neuen Technologie nachprogrammiert. Dieses Vorgehen bei der Migration bringt einige Probleme mit sich. Durch den Umfang der *CommSy*-Software ist inzwischen nur noch in groben Grenzen erkennbar – wenn überhaupt – welche Teile der ursprünglichen Funktionalität bisher in das neue System migriert worden sind. Die mangelnde Übersicht macht es den Entwicklern schwer, Aussagen darüber treffen zu können, wie weit die Migration inzwischen fortgeschritten ist. Außerdem gestaltet sich das Bestimmen noch zu migrierender Systemteile zunehmend schwieriger, vor allem in den Teilen der Software, in der schon viel Migrationsarbeit geleistet wurde und in denen unter Umständen nur noch verstreut liegende Funktionalität zu migrieren ist.

Ein weiteres Problem ergibt sich daraus, dass das *CommSy* parallel zur Migration selbst noch weiterentwickelt wird. Das im Einsatz befindliche System wird gewartet und erweitert. Mit fortschreitender Migration steigt somit auch die Wahrscheinlichkeit, dass Funktionalität, die schon längst auf die neue Technologie migriert wurde, im ursprünglichen System nachträglichen Änderungen und Erweiterungen unterliegt. Die fehlende Übersicht über den Stand der Migration erschwert es zudem, die von Änderungen betroffenen Systemanteile schon migrierter Funktionalität zu identifizieren.

Das Ziel dieser Arbeit ist es, für die genannten Probleme eine Lösung zu entwickeln, die einem Entwicklerteam die Möglichkeit gibt, den Fortschritt der Migration in einer integrierten Entwicklungsumgebung (*Integrated Development Environment*, kurz *IDE*) zu verfolgen und zu steuern. Diese Lösung ist als Erweiterung (ein so genanntes *Plug-In*) der IDE gedacht. Entsprechend dem im *WebMig*-Projekt gewählten Vorgehen bei der Migration soll das Plug-In – das *Software Migration Plug-In* – speziell die Arbeit im Quelltext geeignet unterstützen und erweitern.

Um dieses Ziel zu erreichen, werden zuerst Konzepte erarbeitet, anhand derer sich der Quelltext der beiden, an der Migration beteiligten Projekte geeignet einteilen und in Beziehung zueinander setzen lässt. Aus dieser Beziehung werden Erkenntnisse über den Stand der Migration gewonnen. Den Entwicklern wird während der Arbeit an der Migration im Quelltext angezeigt, welche Teile des *CommSy* schon migriert wurden und welche noch zu migrieren sind. Dazu ist es notwendig, dass die Entwickler Werkzeuge an die Hand bekommen, um Systemteile in beiden Systemen identifizieren und verknüpfen zu können, in denen sich dieselbe Funktionalität befindet.

Die fortlaufenden Änderungen am *CommSy* sind ein weiterer Aspekt, der bei der Ausarbeitung der Lösung betrachtet wird. Die Lösung bietet dafür eine Möglichkeit an, den Entwicklern des *WebMig*-Projekts nachträgliche Änderungen im Ursprungssystem mitteilen zu können. Schon migrierte Systemteile können darüber geprüft werden, ob nachträgliche Änderungen Korrekturen oder Erweiterungen des Zielsystems nach sich ziehen.

Anhand der in dieser Arbeit entworfenen Lösung wird untersucht, welche Schwierigkeiten sich bei der schrittweisen Migration von Quelltext zu Quelltext ergeben. Das betrifft vor allem das Finden einer sinnvollen Einteilung der Quelltexte hinsichtlich ihrer funktionalen Bedeutung sowie der Art und Weise, damit geeignet Beziehungen zwischen Ursprungs- und Zielsystem herzustellen. Ein weiterer Punkt dabei ist die zweckdienliche Darstellung der daraus gewonnenen Informationen in der Entwicklungsumgebung, vor allem im Quelltext selbst.

1.3. Gliederung

Das zweite Kapitel dient als Einführung in das *WebMig*-Projekt, insbesondere in das Vorgehen bei der Quelltextmigration. Dazu wird die Quelltextmigration eingangs im Kontext anderer Migrationsformen dargestellt und von diesen abgegrenzt. Ferner enthält es eine kurze Darstellung der beiden Technologien, zwischen denen die Migration stattfindet. Außerdem werden die im Projekt verwendete Entwicklungsumgebung *Eclipse* [Eclipse 2007] sowie der jeweilige Arbeitsbereich der in beiden Teams arbeitenden Entwickler vorgestellt. Die Einführung dient dazu, den Kontext des *WebMig*-Projekts zu schildern, in dem sich diese Arbeit bewegt.

Im dritten Kapitel werden grundlegende Ideen und Konzepte für eine Unterstützung der Quelltextmigration entwickelt und vorgestellt. Diese sollen es ermöglichen, auf Basis einer Einteilung und Verknüpfung der Quelltexte der beiden Systeme Aussagen über Zustand und Fortschritt der Migration zu gewinnen. Die Konzepte bilden die Grundlage für die Umsetzung einer die Migration unterstützenden Softwarelösung.

Das vierte Kapitel beschreibt den Weg vom Ermitteln der Anforderungen bis hin zur technischen Realisierung. Der Beschreibung der Anforderungsermittlung folgt der Entwurf eines Daten- und eines Benutzungsmodells, die die in Kapitel 3 entwickelten Konzepte umsetzen. Anhand von *Szenarios* (vgl. [Züllighoven 1998], S.611-612) werden die Arbeitsschritte eines Entwicklers beschrieben, um in der Entwicklungsumgebung Ursprungs- und Zielprojekt zu verknüpfen und Informationen hinsichtlich des Migrationsfortschritts zu gewinnen. Die Modelle sind die Basis für die technische Realisierung des Plug-Ins in der Entwicklungsumgebung *Eclipse*. Für einzelne Punkte der Umsetzung werden alternative Lösungsansätze vorgestellt und diskutiert sowie die Wahl der jeweils im Plug-In umgesetzten Alternative begründet.

Das fünfte Kapitel wertet die Ergebnisse aus, die sich bei der Ausarbeitung des Plug-Ins ergeben haben. Dazu gehört, inwieweit durch die vorgestellte Lösung die Migration im Quelltext unterstützt wird und ob die Lösung die gewünschte Übersicht über die Migration liefern kann.

Das sechste Kapitel fasst die einzelnen Schritte von der Ausarbeitung der grundlegenden Konzepte über Entwurf der Modelle bis hin zur Realisierung der Softwarelösung noch einmal zusammen. Ferner gibt es einen Ausblick darauf, was durch die Arbeit nicht abgedeckt wird und welche weiteren Möglichkeiten sich daraus für eine Unterstützung der Quelltextmigration ergeben.

1.4. Konventionen

Der Text dieser Arbeit folgt einigen Formatierungskonventionen. Dabei werden besondere Begriffe, die aus dem technischen oder fachlichen Kontext der Arbeit stammen, bei der ersten Einführung im Text kursiv dargestellt (zum Beispiel *Workspace* oder *Segmentierung*). Technische Eigennamen und Zitate werden durchgehend kursiv geschrieben (zum Beispiel *Java*

und *Eclipse*). Die Bezeichner von Klassen und Schnittstellen werden in serifenloser Schrift aus dem übrigen Text hervorgehoben (zum Beispiel org.eclipse.swt.graphics.Color).

In den Kapiteln, in denen die Arbeit mit dem Plug-In im Zusammenhang mit dem Benutzer bzw. der Benutzerin beschrieben wird – also den an der Migration beteiligten Entwicklern -, werde ich der Einfachheit halber als Referenz auf BenutzerInnen des *Plug-Ins* nur noch von dem Benutzer sprechen, um eine unnötige Aufblähung des Textes durch konsequente und jeweils vollständige Verwendung der korrekten weiblichen und männlichen Ansprache zu vermeiden. Die Wahl der männlichen Form ist dabei rein willkürlich und stellt keine besondere Präferenz der einen bzw. Ablehnung der anderen Form seitens des Autors dar, sondern dient nur der Vereinfachung des Texts. Eine Ausnahme bilden die Textpassagen, in denen ich auf die Benutzerin Alla Remfert verweise, die das in dieser Diplomarbeit entwickelte Plug-In im Zuge ihrer eigenen Studienarbeit benutzt und getestet hat.

Der Begriff *Benutzer* wurde zudem zur deutlichen Abgrenzung zum *Entwickler* gewählt. Dies wird an vielen Stellen der Arbeit als notwendig erachtet, um Missverständnissen vorzubeugen, da die *Entwickler* des *CommSy* und *JCommSy* zwar gleichzeitig *Benutzer* des Plug-Ins sind, aber das Plug-In nicht selbst entwickeln. Trotzdem handelt es sich bei den am *WebMig*-Projekt beteiligten Entwicklern sowie den Benutzern des Plug-Ins im Wesentlichen um denselben Personenkreis.

2. Grundlagen für die Migration im *WebMig*-Projekt

2.1. Verschiedene Arten von Migration

Nach Kremers und van Dissel [Kremers 2000] ergibt sich die Notwendigkeit einer Migration aus den wachsenden Anforderungen an ein Softwaresystem, die erfüllt werden müssen, sowie den technischen Möglichkeiten, die ausgeschöpft werden sollen. Dazu wird ein vorhandenes Softwaresystem in ein neues System überführt, das die neuen Anforderungen erfüllt. Diese Überführung kann dadurch bewerkstelligt werden, dass aus dem Ursprungssystem alle Anforderungen ermittelt werden, die das neue System erfüllen muss. Diese Art der Migration entspricht einer Neuentwicklung; beim ermittelten Anforderungskatalog könnte es sich genauso gut um das Anforderungsdokument handeln, das für das Ursprungssystem erstellt und anhand dessen das System gebaut wurde. In der Softwareentwicklung werden noch weitere Formen der Migration unterschieden.

Ein Beispiel für eine Migration ist die Aktualisierung einer ERP-Software[1] wie SAP. Wird diese von SAP R/2 auf SAP R/3 aktualisiert, so sind weitreichende technische sowie funktionale Änderungen zu berücksichtigen. Bei einer solchen Migration liegt das Hauptaugenmerk auf der Schulung der Benutzer in der Benutzung der neuen Software und der Migration der Bestandsdaten auf die neue technologische Basis.

Ein weiteres Beispiel stellt die Migration von Datenbeständen innerhalb einer Datenbank oder von einer Datenbanksoftware auf eine andere dar. Im Unterschied zur Softwaremigration beschränkt sich diese Migration nur auf die Inhalte und die Strukturen einer Datenbank. Bei so genannten agilen Projekten (vgl. [Cockburn 2005], S.165-167) werden dem Auftraggeber regelmäßig neue Versionen der in der Entwicklung befindlichen Software übergeben. Diese können in Teilen schon produktiv eingesetzt werden, bevor das Gesamtpaket fertig ist. Daher ist es häufig notwendig, dass die Produktivdaten des Auftraggebers auf die neueren Versionen der Software übertragen werden. Das Hauptaugenmerk bei dieser Form der Migration liegt auf Datenerhaltung sowie dem Einhalten bestehender Konsistenzbedingungen.

In Abgrenzung zu den beiden oben genannten Beispielen von Migration handelt es sich im Kontext des *WebMig*-Projekts um eine Migration der Software auf Quelltextbasis von einer Technologie auf eine andere. Das heißt, dass die komplette Funktionalität des *CommSy* auf der Basis des Quelltexts ermittelt und im *JCommSy* implementiert wird, sich im Quelltext des *JCommSy* wieder findet. Ausgenommen ist hierbei *CommSy*-spezifische, genauer gesagt *PHP*-spezifische Funktionalität. Diese ist technologisch motiviert und wird für das Funktionieren im *JCommSy* nicht benötigt. Für die quelltextgestützte Migration gibt es unterschiedliche Ansätze. So beschreibt Cordy in [Cordy 2006] die automatisierte Migration von Quelltexten einer Sprache in Quelltexte einer anderen Sprache anhand *TXL*-gestützter[2] Transformationen. Dabei werden die Transformationen von Kontrollstrukturen in *TXL* beschrieben. Der Quelltext des Ursprungssystems wird geparst, der entstandene Parse-Baum anhand der *TXL*-Beschreibungen transformiert und der transformierte Parse-Baum dann wieder in die Zielsprache zurückgeparst.

Beim *WebMig*-Projekt wird diese Überführung von Hand durchgeführt. Funktionalität wird dabei Stück für Stück im Zielsystem in *Java* nachprogrammiert, so dass das Ursprungsprojekt schrittweise in die neue Technologie überführt werden kann. Als Ausgangspunkt für die Funktionalität im Ursprungsprojekt dient der dazugehörige Quelltext. Technologiespezifische oder ungenutzte Teile des Ursprungsprojekts müssen dabei identifiziert und von der Migration ausgenommen werden. In dieser Arbeit wird mit einer geeigneten Einteilung und Zuordnung des Quelltexts beider Projekten ein pragmatischer Ansatz untersucht, diese manuelle Überführung zu unterstützen und kontrollierbar zu machen.

---

[1] Enterprise Resource Planning, ERP: Software zur Planung der Verwendung von Unternehmensressourcen.

[2] Turing eXtender Language, TXL, Programmiersprache.

2.2. Das *WebMig*-Projekt

Im Zuge des *WebMig*-Projekts wird an der Universität Hamburg das vom Arbeitsbereich Softwaretechnik entwickelte *CommSy* von der bisherigen *PHP*-basierten Technologie auf *Java* migriert. Das *CommSy* „*ist ein webbasiertes System zur Unterstützung von vernetzter Projektarbeit*“ [Commsy 2007] und ermöglicht die Zusammenarbeit innerhalb eines Projekts über Internetbrowser. Um die Technologiebasis zu modernisieren und auf Industriestandards zu setzen, wird dieses Projekt auf *Java* und die dazugehörige Servertechnologie migriert. Der Name des Zielprojekts lautet *JCommSy*.

Die Technologie im Ursprungsprojekt – *PHP*

Das im Einsatz befindliche *CommSy* wird seit 1999 auf der Basis von *PHP* [PHP 2007] vom Arbeitsbereich Softwaretechnik im Fachbereich Informatik der Universität Hamburg entwickelt. *PHP* wird dabei serverseitig interpretiert, um zum Beispiel dynamische Webseiten zu erzeugen, die als Antwort auf Anfragen aus dem Internet generiert werden. Entwickelt wurde *PHP* 1995 von Rasmus Lerdorf, der eine Reihe von *Perl*-Skripten als „personal home page tools“ zusammenstellte (vgl. [Sebesta 2006], S. 110-111).

PHP ist eine in *HTML* eingebettete, serverseitig interpretierte Skriptsprache. Daher sind *PHP*-Skripte aufgebaut aus Teilen von Antwortcode (in *HTML*), einzelnen Funktionen und Variablen sowie ganzen Klassen. *PHP* vermischt dabei Elemente prozeduraler, reflektiver und objektorientierter Programmiersprachen.

Die Technologie im Zielprojekt – *Java*

Das neue *JCommSy* basiert auf der von *Sun Microsystems* entwickelten *Java*-Technologie [Sun 2007]. *Java* ist eine objektorientierte Programmiersprache. In *Java* verfasste Quelltextdateien beinhalten somit Klassenbeschreibungen mit den enthaltenen Methoden, Funktionen und Variablen. Des Weiteren werden im Zielsystem *JSP* (*Java Server Pages*) für die Beschreibung der Benutzungsoberfläche verwendet. Diese werden ähnlich den *PHP*-Skripten im Ursprungssystem serverseitig interpretiert und sind aus Funktionen und reinen *HTML*-Anteilen aufgebaut.

Die Entwicklungsumgebung – *Eclipse*

Beide Systeme werden derzeit in der *Open Source*-Entwicklungsumgebung *Eclipse* entwickelt. *Eclipse* wurde ursprünglich von *IBM* entwickelt und Anfang 2004 für die *Open Source*-Entwicklung freigegeben. Die Entwicklungsumgebung und die Gemeinschaft, die sich um das *Open Source*-Projekt gebildet hat, beschreiben deren Gründer so:

> *“Eclipse is an open source community whose projects are focused on providing a vendor-neutral open development platform and application frameworks for building software. The Eclipse Foundation is a not-for-profit corporation formed to advance the creation, evolution, promotion, and support of the Eclipse Platform and to cultivate both an open source community and an ecosystem of complementary products, capabilities, and services.” (vgl. [Eclipse 2007])*

Eine ähnliche Beschreibung findet sich in dem *Eclipse*-Buch von McAffer und Lemieux, in der speziell die in dieser Arbeit zu Grunde gelegte Entwicklungsumgebung Erwähnung findet:

> *„First and foremost, Eclipse is an open source community of people building Java™ based tools and infrastructure to help you solve your problems. The most obvious output of the community is the Eclipse Java integrated development environment (IDE).” (vgl. [McAffer 2005])*

Die Erweiterbarkeit von *Eclipse* beruht auf der Plug-In-Architektur dieser Software. Für das *CommSy*-Projekt sowie für das *JCommSy*-Projekt bedeutet das in erster Linie, dass sowohl das Ursprungs- wie auch das Zielsystem in *Eclipse* weiterentwickelt werden können, da entsprechende Werkzeuge und Editoren als Plug-Ins vorhanden sind und eingebunden werden können. Für eine Migrationsunterstützung bietet sich daher ebenso die Entwicklung einer Erweiterung an, die neben den in *Eclipse* vorhandenen Werkzeugen in die IDE eingebunden werden kann.

Hinter *Eclipse* steht eine ähnliche Philosophie wie seinerzeit bei der Entwicklung von *SmallTalk*. Benutzer sollten die von ihnen benutzte Software selber gestalten, ausbauen und umändern können. *Eclipse* bietet dazu den so genannten Plug-In-Mechanismus. Damit unterscheidet sich *Eclipse* im Ansatz von *Rich-Client*-Anwendungen, die die meiste Funktionalität in einem großen Anwendungsblock zur Verfügung stellen und bei denen die Plug-In-Schnittstelle nur ein Anhang an den Anwendungsteil darstellt.

Im Gegensatz zu klassischen Entwicklungsumgebungen, die oftmals als abgeschlossene Softwarepakete vertrieben werden und dem Entwickler wenige bis gar keine Erweiterungsmöglichkeiten bieten, baut sich *Eclipse* selbst fast vollständig aus Erweiterungen auf. Bis auf einen kleinen Kern (*run-time kernel*, vgl. [Beck 2003], S. 16), der das Registrieren, Laden und Ausführen von Erweiterungen (Plug-Ins) verwaltet, besteht die gesamte Entwicklungsumgebung selbst aus Erweiterungen. Es gibt Quelltext-Editoren für verschiedene Programmiersprachen, Werkzeuge für Dateinavigation und –verwaltung, Suchfunktionen, Versionierungswerkzeuge, Datenbankverwaltung etc.

Die *Eclipse*-Entwicklungsumgebung gliedert sich in die drei in Abb. 2.1. dargestellten Schichten. Wesentlich dabei ist die *Eclipse*-Plattformkomponente, die den Kern sowie grundlegende Komponenten der Benutzungsschnittstelle enthält. Darauf bauen die eigentliche IDE - der so genannte *Java Development Toolkit* – sowie die Plug-In-Entwicklungsumgebung auf.

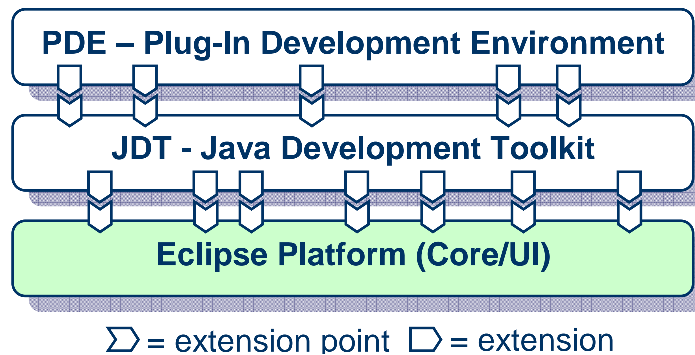


Abb. 2.1.: *Eclipse*-Architektur (angelehnt an [Beck 2003], S. 5)

Wie in der Abbildung beispielhaft dargestellt, erfolgt der Zugriff der auf der *Eclipse*-Plattform aufbauenden Schichten über das Einklinken von *extensions* in *extension points*. Diese stellen die Schnittstellen zwischen den Schichten in *Eclipse* dar und werden detaillierter in Kapitel 4 bei der Beschreibung der technischen Realisierung betrachtet. Wesentlich dabei ist, dass die Plattformschicht die interne, programmiersprachen-neutrale Infrastruktur definiert und das Dateisystem kapselt (vgl. [Beck 2003], S. 5). Das JDT arbeitet vollständig auf den durch die Plattform zur Verfügung gestellten Schnittstellen. Zugriffe auf das Dateisystem sind zwar aus den Klassen des JDT prinzipiell möglich, aber durch die vorgegebene Architektur von *Eclipse* untersagt. Das oberhalb des JDT befindliche PDE nutzt und erweitert die Komponenten des JDT, um die Entwicklung von Plug-Ins in *Eclipse* zu unterstützen.

2.3. Benutzungskontext

Für die Handhabung einer Migrationsunterstützung im *WebMig*-Projekt lässt sich ein Satz von Anforderungen ermitteln. Ein Teil der Anforderungen ergibt sich aus Voraussetzungen, die der Benutzungskontext mit sich bringt. Dies sind die Anforderungen, die zum Beispiel aus der unterschiedlichen Arbeit der Entwicklerteams des *CommSy*- und *JCommSy*-Projekts entstehen. Dazu kommen Anforderungen, die sich aus der Benutzung von *Eclipse* als Entwicklungsumgebung heraus ergeben und eng mit dem sonstigen Verhalten der IDE gekoppelt sind. *Eclipse* gibt für Erweiterungen einige Richtlinien vor, um das Gesamtbild und Verhalten der Anwendung für den Benutzer möglichst gleichförmig zu halten. Die Anforderungen reflektieren diese Richtlinien, damit sich die Migrationsunterstützung nahtlos in die übrige IDE eingliedert und die Arbeit der Benutzer nicht behindert wird.

Um den Kontext und die sich daraus ergebenden Anforderungen in Erfahrung zu bringen, wurden einige Benutzer der Migrationsunterstützung ebenso wie Benutzer der Entwicklungsumgebung befragt. Weitere Anforderungen ergaben sich aus den Kommentaren, die während der Entwicklung über Benutzbarkeit und Präsentation der Migrationsunterstützung hereinkamen. Die sich daraus ergebenden Benutzerprofile werden im folgenden Kapitel beschrieben.

2.3.1. Die unterschiedlichen Arten von Entwicklern

Für die Ausarbeitung einer Lösung zur Migrationsunterstützung im *WebMig*-Projekt wurden zwei Arten von Entwicklern unterschieden. Beiden Arten von Entwicklern ist gemein, dass es sich jeweils um Entwickler in einem der beiden Softwareprojekte handelt – dem ursprünglichen *CommSy*, das migriert wird, sowie dem neuen *JCommSy*, in das migriert wird. Es gibt aber deutliche Unterschiede zwischen beiden Arten von Entwicklern, was die Nutzung der Migrationsunterstützung betrifft. Diese Unterschiede werden im Folgenden kurz aufgezählt:

1) *JCommSy*-Entwickler
    a. entwickelt im *JCommSy*-Projekt, also in dem Projekt, in das das ursprüngliche *CommSy*-System migriert wird.
    b. nutzt die Migrationsunterstützung, um einen Überblick darüber zu bekommen, welche Teile des Ursprungssystems schon migriert wurden und welche noch offen sind
    c. unterteilt Quelltext des Ursprungs- und Zielprojekts anhand von Quelltextabschnitten, um mit ihrer Hilfe die Migration zu systematisieren. Dazu ordnet er die Abschnitte aus dem Ursprungsprojekt Abschnitten des Zielprojekts zu, in denen dieselbe Funktionalität implementiert ist.
    d. arbeitet somit im Quelltext beider Projekte. Obwohl der *JCommSy*-Entwickler hauptsächlich im *JCommSy*-Projekt entwickelt, fügt er durch seine Unterteilungsarbeit dem Quelltext im *CommSy*-Projekt Informationen hinzu.

2) *CommSy*-Entwickler
    a. pflegt das Ursprungssystem *CommSy* und entwickelt es weiter. Dazu gehören zum Beispiel Erweiterungen oder Fehlerkorrekturen des im Einsatz befindlichen *CommSy*.
    b. ist nicht mit der Entwicklung des *JCommSy* beschäftigt; er hat nur Änderungen am *CommSy* zu verantworten und im Überblick zu behalten. Ebenso hat der *CommSy*-Entwickler nicht dafür zu sorgen, dass Änderungen am *CommSy* an das *JCommSy* weitergegeben werden.
    c. arbeitet derzeit ausschließlich im Quelltext des Ursprungssystems. Die Einteilung des Ursprungssystems durch den *JCommSy*-Entwickler ist für ihn nicht von Interesse. Die zusätzliche Menge an Informationen, die durch diese Einteilung im Quelltext erscheint, beeinträchtigen sogar die Übersichtlichkeit des Quelltexts und können vom *CommSy*-Entwickler somit als störend empfunden werden.

Aus Gesprächen mit *JCommSy*-Entwicklern wurde deren Arbeits- und Sichtweise erarbeitet. Dabei wurde besonders Wert darauf gelegt, Möglichkeiten zu finden, eine Migrationsunterstützung derart in den Entwicklerarbeitsplatz zu integrieren, dass sie parallel zur Entwicklungstätigkeit genutzt werden kann. Die Anforderungen der oben beschriebenen *CommSy*-Entwickler ergaben sich aus den Rückmeldungen, die nach der Einführung des Plug-Ins eingingen. Diese betrafen zum Beispiel die Ausprägung der Migrationsinformationen in den Quelltexten der beiden Systeme, in diesem Fall besonders in den Quelltexten des *CommSy*. Mithilfe der Rückmeldungen wurde versucht, die Sichtweise eines *CommSy*-Entwicklers einzunehmen, der das Plug-In nicht nutzen wird, aber in dessen Projekt die zusätzlichen Migrationsinformationen des Plug-Ins erscheinen. Hauptaufgabe dieser Sicht auf das Plug-In besteht darin, den *CommSy*-Entwickler nicht durch die zusätzlichen Quelltext-Informationen, die für die Migrationsunterstützung im *CommSy*-Quelltext notwendig sind, in seiner Arbeit zu beeinträchtigen.

Dadurch, dass sich das Plug-In im Quelltext beider Systeme bewegt, wird der *CommSy*-Entwickler, der ansonsten wenig mit der Migration ins *JCommSy* zu tun hat, indirekt zu einem Plug-In-Nutzer. Um dies zu berücksichtigen, wird bei der Konzeption des Plug-Ins eine Möglichkeit gesucht, entweder mit einer möglichst geringen Menge an für die Migration notwendigen Informationen auszukommen, oder diese vollends vor einem *CommSy*-Entwickler zu verbergen. Letzteres bedingt, dass der *CommSy*-Entwickler dann das Plug-In und dessen erweiterte Editoren nutzen müsste, um die zusätzlichen Migrationsinformationen ausblenden zu können.

Aus diesen beiden Sichtweisen ergibt sich eine grundsätzliche Anforderung für eine Migrationsunterstützung: für einen Teil der Entwickler stellt das Werkzeug die gesamte Funktionalität zur Verfügung. Für den anderen Teil der Entwickler verbirgt das Werkzeug möglichst viele der migrationsspezifischen Informationen oder hält deren Umfang möglichst gering.

2.3.2. Nutzung des Quelltexts als Grundlage

Für die Migration von *PHP* nach *Java* bzw. *CommSy* nach *JCommSy* wird eine einheitliche Grundlage benötigt, auf der unterstützende Werkzeuge arbeiten können. Eine solche Grundlage bildet bei den beiden Projekten die gemeinsam nutzbare, zentrale Quelltextbasis, das so genannte *Repository* (von engl. *repository* für Lager, Quelle). Für die beiden verwendeten Sprachen bietet sich wiederum der Quelltext an. Dieser enthält die Funktionalität der beiden Systeme in menschenlesbarer Form (im Gegensatz zum Beispiel zu den kompilierten Klassen des Zielsystems *JCommSy*) und erlaubt von daher die direkte Arbeit auf den die Funktionen definierenden Texten. Des Weiteren ist die Quelltextbasis für beide Systeme dieselbe, so dass die Entwickler des *JCommSy* auf der aktuellen Fassung des Ursprungsprojekts arbeiten können.

Da bei der Migration von *CommSy* nach *JCommSy* die Programmiersprache wechselt, handelt es sich bei dieser Migration streng genommen um eine Neuentwicklung [Kremers 2000]. Der wesentliche Unterschied zur Neuentwicklung besteht indes darin, dass die grundlegende Funktionalität durch das bestehende und im Einsatz befindliche *CommSy* detailliert vorgegeben ist. Daher wird die Migration anhand einer unmittelbaren Überführung von Ursprungsfunktionalität in die Zielfunktionalität erreicht. Während zwei Projekte, die anhand desselben Anforderungskatalog entwickelt werden, im Endeffekt völlig unterschiedliche Strukturen hervorbringen können, wird bei dieser direkten Quelltext-zu-Quelltext-Migration die Struktur des Ursprungssystems übernommen. Das heißt, dass ein Quelltextabschnitt aus dem Ursprungssystems wieder auf einen Quelltextabschnitt im Zielsystem abgebildet wird.

2.3.3. Die Nutzbarkeit aus einer IDE heraus

Eng mit der Nutzung der Quelltexte von Ursprungs- und Zielprojekt ist die grundlegende Anforderung verbunden, die Migrationsunterstützung in die Entwicklungsumgebung *Eclipse*

einzubetten. Diese Anforderung ergibt sich aus der Sicht auf das *WebMig*-Migrationsprojekt als schrittweise Überführung von Funktionalität aus einem Ursprungssystem in ein Zielsystem. Dabei wird der Ansatz verfolgt, dass der Quelltext des Ursprungssystems Stück für Stück in ein und derselben Entwicklungsumgebung in das Zielsystem und dessen Sprache überführt wird.

Die Entwicklungsumgebung *Eclipse* bietet für beide verwendeten Programmiersprachen Editoren und weitere Werkzeuge. Die Einbettung in die Entwicklungsumgebung ermöglicht daher eine Form der Unterstützung, die direktes Arbeiten auf den Quelltexten von Ursprungs- und Zielsystem bietet und somit die schrittweise Migration von Quelltextdatei zu Quelltextdatei erleichtert.

3. Konzepte für die Quelltextmigration im *WebMig*-Projekt

Im Folgenden werden die Konzepte erarbeitet und ausformuliert, die der Quelltextmigration im *WebMig*-Projekt zugrunde liegen bzw. dazu dienen können, diese zu systematisieren. Die Konzepte bilden dann die Grundlage für die nachfolgende Realisierung einer Migrationsunterstützung.

3.1. Voraussetzungen

Die Forderung nach einer Migrationsunterstützung für das *JCommSy*-Projekt entstand aus der Notwendigkeit heraus, einen Überblick über den Fortschritt der Migration zu bekommen. Dabei war maßgebend, dass das Projekt schon seit einiger Zeit migriert wurde und den Entwicklern des *JCommSy* keine Unterstützung zur Verfügung stand, mithilfe derer sie feststellen konnten, welche Teile des Ursprungsprojekts schon migriert waren und welche noch offen waren.

Verschärft wird das Problem der fehlenden Übersicht über den Migrationsfortschritt noch dadurch, dass das Ursprungsprojekt *CommSy* während der Migration einer Weiterentwicklung unterliegt. Die Entwickler des *CommSy*-Projekts sind fortwährend damit beschäftigt, Erweiterungen und Fehlerkorrekturen am *CommSy* vorzunehmen, was die Arbeit der *JCommSy*-Entwickler zusätzlich erschwert. Selbst Projektteile, deren Migration ein Entwickler noch unmittelbar im Blick hat, können schon durch Änderungen am Ursprungsprojekt an Funktionalität gewonnen oder verloren haben.

Im Wesentlichen besteht also die Hauptaufgabe einer im Zuge dieser Arbeit entwickelten Migrationsunterstützung darin, zwei Projekt-Repositories miteinander zu synchronisieren, so dass den Entwicklern des Migrationsprojekts Möglichkeiten an die Hand gegeben werden, den Fortschritt der Migration im Überblick zu behalten sowie Änderungen am Ursprungsprojekt erfassen zu können. Hinzu kommt, dass durch die technologischen Unterschiede der beiden Systeme einige Teile des Ursprungssystems nicht in das Zielsystem übernommen werden, da es sich um technologiespezifische Anteile handelt. Dementsprechend brauchen die Entwickler ein Mittel, nicht zu migrierende Systemanteile kenntlich zu machen und so von noch zu migrierender Funktionalität unterscheiden zu können.

3.2. Zerlegung und Zuordnung des Quelltexts

Um zwei Projekte hinsichtlich der Migration zu synchronisieren, wird eine Möglichkeit benötigt, die Quelltexte der beiden Projekte miteinander in Beziehung zu setzen. Dabei gilt es, ein Modell zu finden, mit der sich die Quelltexte beider Projekte zerlegen lassen, um dann eine Verbindung zwischen Ursprungs- und Zielsystem herzustellen. Dieses Modell beschreibt, auf welche Art Quelltext zerlegt und die so gewonnenen Einheiten einander zugeordnet werden können. Für die Migrationsanalyse wird dieses Modell dann benutzt, um in beiden Systemen Abschnitte zu identifizieren und einander zuzuordnen, die dieselbe Funktionalität implementieren.

Zerlegung und Zuordnung lassen sich dabei unterschiedlich modellieren. Denkbar wären Einheiten, im Folgenden als *Segmente* bezeichnet, die die Quelltextabschnitte einfassen und um für die Migration benötigte Informationen erweitern. Eine weitere Möglichkeit bieten Einheiten, die jeweils durch eine Verbindung von zwei einander zugeordneten Quelltextabschnitten definiert sind. Die Verbindungseinheiten werden hier als *Kopplung* bezeichnet. Beide Alternativen bieten unterschiedliche Möglichkeiten, die für die Migrationsanalyse nötigen Informationen aus ihnen abzuleiten. Daher werden die beiden Alternativen - Kopplung und Segment - als grundlegende Einheiten der Zerlegung und Zuordnung von Quelltext im Folgenden genauer untersucht.

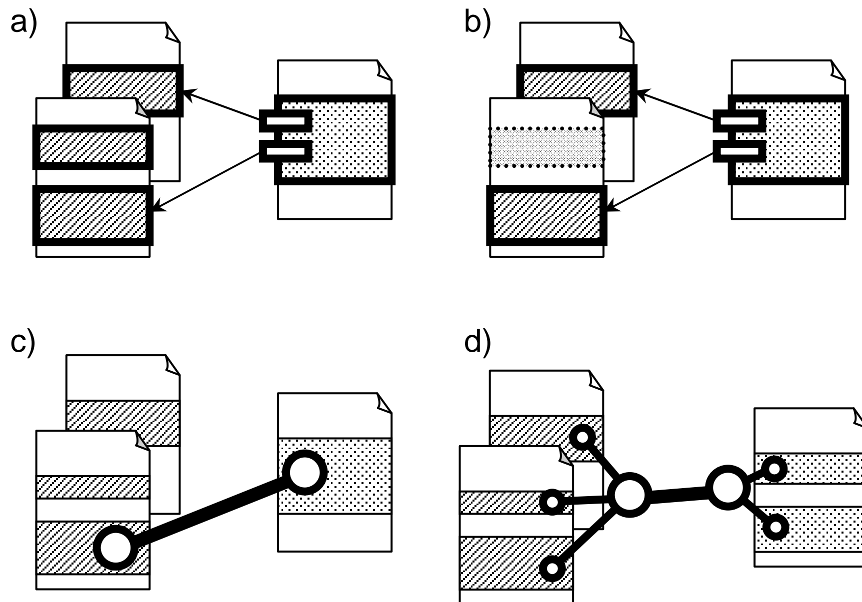


Abb. 3.1.: Beispiele für Segmente und Kopplungen als Modell für eine Zerlegung und Zuordnung des Quelltexts

Einige Beispiele für die beiden betrachteten Modelle sind in Abb. 3.1. aufgeführt. Die Beispiele zeigen in vereinfachter Darstellung die Besonderheiten der beiden Modelle. Es werden jeweils Abschnitte im Quelltext von Ursprungsprojekt (links) und Zielprojekt (rechts) einander zugeordnet. Das Beispiel in Abb. 3.1. a) zeigt die Einteilung der Quelltexte in Segmente. Die Informationen über die Zuordnung von Segmenten zueinander werden an den Segmenten abgelegt (als kleine Rechtecke am Segment dargestellt), ebenso wie zusätzliche Informationen wie zum Beispiel der Zustand der Segmente. Die Richtung der Zuordnung ist für dieses Beispiel nicht relevant, wurde aber gemäß der Umsetzung im Plug-In vom Ziel zum Ursprung angegeben. Der Grund dafür – Reduzieren der Datenmenge - wird ausführlich in Kapitel 4.3. im Abschnitt über die technische Realisierung des Plug-Ins beschrieben. In Abb. 3.1. b) wird ein Segment, welches nicht in das Zielsystem überführt wird, als obsolet markiert (im Beispiel grau hinterlegt und mit Punktrahmen versehen).

Das Beispiel in Abb. 3.1. c) stellt eine Verbindung zweier Quelltextabschnitte über eine Kopplung dar. Die Kopplung enthält Informationen darüber, welcher Abschnitt sich jeweils an einem ihrer Endpunkte befindet. So können zum Beispiel Anfang und Ende des Abschnitts im Quelltext in einem Endpunkt abgelegt werden. Die Kopplung enthält darüber hinaus Informationen über die Zustände der durch sie verbundenen Abschnitte. Die Endpunkte der Kopplung sind zudem als Ursprungs- und Zielpunkt gekennzeichnet. Durch diese Kennzeichnung wird aus der Kopplung eine gerichtete Verbindung, aus der sich die Richtung der Migration ableiten lässt. Im letzten Beispiel in Abb. 3.1. d) wird die Besonderheit der Kopplung als Modell insbesondere in Hinblick auf die Zuordnung von Quelltextabschnitten zueinander gezeigt. Die Kopplung bietet eine einfache Möglichkeit, *n:m-Zuordnungen* (vgl. Kapitel 3.3.) darzustellen. Die Endpunkte der Kopplung verweisen dann nicht mehr auf einen einzigen Quelltextabschnitt, sondern auf eine Menge von Abschnitten, die in beiden Systemen dieselbe Funktionalität umfassen.

Verfügbare Information	Kopplung als Einheit	Segment als Einheit
Zuordnung	Ja	Ja
Obsolete Systemteile	Nein	Ja
Zusammengehörigkeit	Ja	Eingeschränkt
Änderungen	Ja	Ja
Unterscheidung nicht betrachtet / nicht migriert	Nein	Ja

Tab. 3.2.: Bewertung von Kopplungen und Segmenten als Modell für eine Zerlegung des Quelltexts

In Tab. 3.2. ist für beide Modelle für eine Auswahl von Informationen angegeben, ob diese sich an der jeweiligen Einheit wiederfinden lassen. Diese Auswahl wurde so getroffen, dass der Vergleich der beiden Modelle hinsichtlich der Verfügbarkeit einiger Schlüsselinformationen Anhaltspunkte liefert, welches der beiden Modelle für eine Umsetzung im Plug-In geeigneter ist. Der Auswahl lassen sich noch weitere Punkte hinzufügen, zum Beispiel die Richtung der Zuordnung, allerdings wurde die Auswahl der Übersichtlichkeit halber auf einige wenige Punkte beschränkt, die für die Zerlegung und Zuordnung von Quelltextabschnitten als wesentlich betrachtet werden können.

Die Zuordnung von Ursprungs- und Zielfunktionalität zueinander lässt sich sowohl an einer Kopplung als auch an einem Segment festmachen. Dafür benötigt eine Kopplung lediglich Kenntnis über die einander zugeordneten Bereiche an ihren beiden Enden. Ein Segment kann mit Informationen versehen werden, welches andere Segment mit ihm verbunden ist.

Um Systemteile als obsolet, also als nicht zu migrieren kennzeichnen zu können, werden die entsprechenden Segmente mit einer besonderen Kennzeichnung versehen. Bei einer Kopplung ist eine solche Kennzeichnung nicht möglich, da sie nur als Zuordnung zwischen mindestens zwei Abschnitten existiert. Es wäre eine Sonderform der Kopplung notwendig, die nur einen Endpunkt (den zu kennzeichnenden Abschnitt) besitzt und ausschließlich der genannten Kennzeichnung dient.

Im folgenden Kapitel werden unterschiedliche Arten von Zuordnung untersucht, die sich in der Anzahl der einander zugeordneten Quelltextabschnitte unterscheiden. Für die Betrachtung der Tauglichkeit von Kopplungen und Segmenten als Modell für die Zuordnung ergibt sich an dieser Stelle die Fragestellung nach der Zusammengehörigkeit von Einheiten. Das bedeutet in diesem Zusammenhang, dass sich die an der Migration einer bestimmten Funktionalität beteiligten Quelltextabschnitte als zusammengehörig identifizieren lassen. Werden Kopplungen verwendet, so lassen sich die Endpunkte jeweils auf Mengen von Endpunkten erweitern, so dass diese Form der Einteilung flexibel ist hinsichtlich der Anzahl einander zugeordneter Quelltextabschnitte. Bei Segmenten gestaltet sich diese Erweiterung als schwierig. Bei komplexen Verknüpfungsgeflechten verteilen sich die Zuordnungsinformationen auf unterschiedliche Segmente, da ein Segment jeweils nur die ihm direkt zugeordneten Segmente kennt. Die Zusammengehörigkeit ergibt sich daher nur implizit aus den Zuordnungen der einzelnen Segmente zueinander und kann nicht direkt an einem Segment abgelesen werden. Die Zusammengehörigkeit von Segmenten hinsichtlich einer bestimmten Funktionalität lässt sich daher nur eingeschränkt an den Segmenten selbst festmachen.

Das Erkennen von Änderungen an zugeordneten Quelltextabschnitten lässt sich über beide Formen realisieren. Die Kopplung erhält dafür Informationen über den Zustand der Quelltextabschnitte zum Zeitpunkt der Zuordnung. Gleichermaßen lässt sich an einem Segment der Zustand eines zugeordneten Segments festmachen.

Als letzter Punkt ist die Unterscheidbarkeit von noch nicht betrachteter zu nicht migrierter Funktionalität aufgeführt. Für eine Verbindung lassen sich diese beiden Formen nicht unterscheiden. Sobald eine Verbindung zwei Quelltextabschnitte einander zuordnet, gilt der jeweilige Quelltextabschnitt aus dem Ursprungsprojekt als migriert. Jeglicher Quelltext, der noch nicht durch eine Verbindung zugeordnet ist, gilt daher als nicht betrachteter Quelltext. Die Nutzung von Segmenten als Einheit erlaubt an dieser Stelle eine detaillierte Unterscheidung. So lässt sich ein Quelltextabschnitt schon als Segment einfassen, ohne dass eine Zuordnung

erfolgen muss. Somit wird dieser Quelltextabschnitt schon als Funktionalitätseinheit identifiziert, während die Migration dieser Einheit (oder die Kennzeichnung als obsolet) noch aussteht.

Die Bewertung der beiden Möglichkeiten einer Einteilung und Klassifizierung legt die Wahl von Segmenten als Mittel zur Einteilung nahe. Diese Wahl stellt einen Kompromiss dar, da ein Segment Schwächen hinsichtlich komplexer Verknüpfungsgeflechte offenbart, dafür das Ausklammern von Quelltext technologiespezifischer Systemanteile erlaubt. Des Weiteren ermöglichen einzelne Segmente eine Einteilung und Klassifizierung eines Systems, ohne die sofortige Zuordnung von Ursprungs- zu Zielabschnitten – ohne die Kopplungen als solche nicht existieren können - zu erzwingen. Dennoch enthält das Modell der Kopplung einige hinsichtlich der Migration interessante Aspekte. Die Vereinfachung der genannten n:m-Zuordnungen von Quelltextabschnitten zueinander ermöglicht die Zusammenfassung größerer Funktionalitätseinheiten beider Projekte in den Endpunkten einer Kopplung. Damit ist bei der Migrationsanalyse eine Betrachtung größerer Systemkomponenten möglich. Diese übergeordnete Einteilung des Systems wird in der vorliegenden Fassung des Plug-Ins nicht unterstützt, allerdings entstand während der Entwicklung das Konzept der Meta-Segmente, denen im Ausblick in Kapitel 6 ein eigenes Kapitel gewidmet ist. Die Idee der Meta-Segmente greift das Konzept der Kopplung auf, mit dem Unterschied, dass Meta-Segmente auf schon vorhandenen Segmenten basieren und nicht, wie die hier beschriebene Kopplung, als alleinstehendes Einteilungs- und Zuordnungskonzept existieren.

Ein weiterer Grund für die Bevorzugung von Segmenten anstelle von Kopplungen als Einheiten für ein Zerlegungs- und Zuordnungsmodell ergibt sich zwar aus Anforderungen an die technische Realisierung, sei der Vollständigkeit halber aber hier schon einmal erwähnt: um für die Migrationsunterstützung zu den vorhandenen Quelltextdateien nicht zusätzliche Datenquellen verwalten und pflegen zu müssen, wurde für die Speicherung der Segmentierungsinformationen der Quelltext selbst um diese Informationen erweitert (vgl. Kapitel 4.3.2.). Da Segmente jeweils genau einen Quelltextabschnitt repräsentieren, lassen sich die in ihnen abgelegten Daten direkt am Abschnitt lokalisieren und speichern. Eine entsprechende Speicherung der Daten einer Kopplung gestaltet sich hingegen schwieriger, da diese nicht speziell einem Abschnitt zugeordnet werden kann, sondern als Verbindung zwischen verschiedenen Abschnitten existiert. Die Lokalisierung einer Kopplung im Quelltext ist somit nicht eindeutig und erschwert das Finden einer geeigneten Quelltextrepräsentation.

Im Folgenden wird der Begriff *Segmentmodell* für das für die Umsetzung im Plug-In gewählte Modell zur Zerlegung und Zuordnung verwendet.

Bestimmung von Segmenten

Ein grundlegendes Problem ergibt sich daraus, wie eine Migration vom Ursprungssystem ins Ziel-system vorgenommen wird und wie Ort und Umfang von Quelltextabschnitten – und somit Segmenten – bestimmt werden, die in einem Migrationsschritt ins Zielsystem überführt werden. Da bei den beiden Systemen unterschiedliche Sprachen und Architekturen zum Einsatz kommen, kann die Umsetzung von Funktionen in beiden Systemen deutlich voneinander abweichen. Bei den Projekten *CommSy* und *JCommSy* kommt einmal eine interpretierte, skriptbasierte, eingeschränkt objektorientierte Programmiersprache (*PHP*) sowie eine kompilierte, typisierte, objektorientierte Programmiersprache (*Java*) zum Einsatz.

Eine mögliche Einheit für das Einfassen in Segmente stellt dafür in beiden Projekten eine Funktion dar. Diese erfüllt dabei eine genau spezifizierte Aufgabe, deren Spezifikation in das Zielsystem übernommen und dort umgesetzt werden kann. Eine Möglichkeit der Zuordnung besteht dann darin, dass Funktionalität mit derselben Spezifikation aus beiden Projekten als Grundlage für die Zuordnung genommen wird. Die Funktionalität kann zu diesem Zweck dann in beiden Projekten in Segmente eingefasst werden.

Aneinanderreihung vs. Verschachtelung
Beim Entwurf des Werkzeugs ergibt sich für die Segmentierung eine wichtige Fragestellung. Es ist zu klären, ob das Werkzeug nur aufeinanderfolgende Segmente erlaubt, die keine Überschneidung zulassen, oder ob ineinander verschachtelte Segmente möglich sind, also ein Segment ein anderes enthalten kann. Die dritte Möglichkeit – das Überlappen von Segmenten – wird nicht weiter betrachtet. Dies lässt eine ungünstige Segmentierung beziehungsweise eine unnötige, da doppelte Implementierung vermuten. Erstens ergibt es wenig Sinn, dass zwei Segmente in unterschiedlichen Zielteilen implementiert werden, aber ebenso einen überlappenden Teil besitzen, der bei beiden – und somit doppelt - migriert wird. Das lässt vermuten, dass im Ursprungsprojekt redundanter Quelltext vorhanden ist. Zweitens erscheint es damit gleichbedeutend mit der Reihenvariante zu sein, da der überlappende Teil einem eigenen Segment im Zielsystem zugeordnet werden kann, um diese Redundanz zu beseitigen.

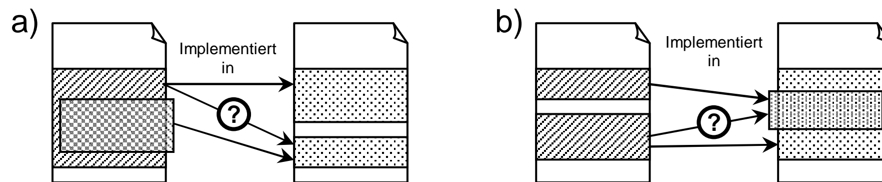


Abb. 3.3.: Zuordnung verschachtelter Segmente im a) Ursprungs- und b) Zielsystem

Die Bedeutung verschachtelter Segmente für die Zuordnung von Ursprungs- zu Zielfunktionalität lässt sich beispielhaft an Abb. 3.3. erläutern. Im Fall a) ist ein Segment in einem größeren Segment eingeschlossen. Die Implementierung der beiden Segmente erfolgt in unterschiedlichen Teilen des Zielsystems (hier der Einfachheit halber in derselben Zieldatei). Die Schwierigkeit an der Deutung dieser Zuordnung liegt darin, dass die im inneren Segment enthaltene Funktionalität eindeutig einem Zielsegment zugeordnet werden kann. Für das äußere Segment stellt sich die Frage, ob die im inneren Segment enthaltene Funktionalität auch dem äußeren Segment zugeordnet wird. Dann wäre das innere Segment redundant und das äußere Segment in zwei Zielsegmenten implementiert. Ist das innere Segment dagegen eigenständig und implementiert einen klar abgegrenzten Funktionsbereich, erscheint eine Auslagerung dieses Segments sinnvoll. Die Migration der spezifischen Funktionalität ist durch die Zuordnung des inneren Segments zum Zielsegment gegeben. Die übrige im äußeren Segment enthaltene Funktionalität lässt sich dementsprechend einem anderen Zielsegment eindeutig zuordnen.

Analog lässt sich der Fall in Abb. 3.3. b) betrachten. In diesem Beispiel ist die Funktionalität aus zwei Ursprungssegmenten in einem Zielsegment implementiert, das ein weiteres Zielsegment vollständig enthält. Die Fragestellung in diesem Fall ist, ob das innere Segment ein integraler Bestandteil des äußeren Segments ist. Dann könnte das innere Segment entfernt und das äußere Segment zwei Ursprungssegmenten zugeordnet werden, deren Funktionalität es vereint. Ist das innere Segment eigenständig, d.h. könnte es zum Beispiel einfach aus dem äußeren Segment hinausgeschoben werden, ohne dessen Funktionalität (aus Sicht des zugeordneten Ursprungssegments) zu verändern, wäre eine Aufteilung in zwei aufeinanderfolgende Segmente möglich.

Ein Beispiel für eine solche Abfolge in einem Segment wäre eine Komponente, die für eine Web-Anwendung eine Sitzung einrichtet, das letzte Anmeldedatum für einen Benutzer in der Datenbank aktualisiert und abschließend die Sitzung zurückgibt. Weiterhin wird angenommen, dass das Einrichten und Zurückgeben der Sitzung und das Aktualisieren des Anmeldedatums im Ursprungssystem an zwei unterschiedlichen Stellen realisiert wurde. An dieser Stelle bietet sich das Einfassen der Aktualisierungsfunktionalität in ein inneres Segment an. Alternativ könnte das Segment vor oder hinter das äußere Segment geschoben werden. Die jeweils implementierte Funktionalität aus dem Ursprungssystem – Sitzung einrichten und zurückgeben und Datum aktualisieren – blieben dabei unverändert.

Zusammenfassend lässt sich sagen, dass das Verschachteln von Segmenten eine Ebene zusätzlich zur direkten Verbindung von Segmenten miteinander einführt. Kann ein Segment

weitere Segmente enthalten, so entsteht eine implizite Abhängigkeit von Ursprungs- und Zielsegmenten zueinander. Das erhöht die Komplexität des Zuordnungsmodells und erschwert die Deutung der Zuordnung von Segmenten zueinander. Demgegenüber steht der oben beschriebene geringe Nutzen, der sich für die Migrationsanalyse ergibt. In den genannten Beispielen lässt sich die Verschachtelung auflösen und durch aufeinanderfolgende Segmente ersetzen.

Aus diesem Grund wurde im Plug-In die Reihenvariante bevorzugt und auf eine Verschachtelung verzichtet. Wird ein Segment innerhalb eines anderen erstellt, so wird das Ursprungssegment aufgeteilt, so dass dem neuen Segment nun ein Segment vorausgeht und ein weiteres Segment folgt.

3.3. Zuordnungsarten

Eine Analyse des Migrationsfortschritts wird durch eine geeignete Verknüpfung von Ursprungsfunktionalität und Zielfunktionalität ermöglicht. Die Segmentierung bietet dem Entwickler dabei die Möglichkeit, die Quelltexte beider Projekte in handliche Einheiten zu unterteilen und gezielt Funktionalität in Segmente einzuschließen. Diese Einteilung erlaubt dann die Zuordnung von Quelltextabschnitten mit einer gewissen Funktionalität im Ursprungsprojekt zu den implementierenden Quelltextabschnitten im Zielprojekt.

Diese Zuordnung tritt in verschiedenen Formen auf, die jeweils hinsichtlich der Migration eine eigene Bedeutung besitzen. Da diese Formen sich ebenso unterschiedlich in der Komplexität bei der Umsetzung im Plug-In niederschlagen, werden sie im Folgenden betrachtet und hinsichtlich ihrer Umsetzbarkeit bewertet.

Die Segmente können auf verschiedene Arten einander zugeordnet werden. Einzig die Richtung ist dadurch vorgegeben, dass Segmente im *PHP*-Quelltext Segmenten im migrierten *Java*-Quelltext zugeordnet sind. Allerdings kann man dabei die in den folgenden Absätzen beschriebenen Arten von Zuordnungen unterscheiden. Die verschiedenen Arten (Kardinalitäten) von Zuordnungen wirken sich auf die Art und Weise aus, wie Informationen in den Segmenten gespeichert werden müssen und wie komplex die Kontrollmechanismen werden, um zum Beispiel eine Änderungsverfolgung zu realisieren.

1:1-Zuordnungen

Dabei ist jeweils genau einem Segment im *PHP*-Quelltext genau ein implementierendes Segment im *Java*-Quelltext zugeordnet. Dies ist die einfachste Form der Zuordnung, zum Beispiel von einer Funktion oder Methode in einer *PHP*-Quelltextdatei zu einer entsprechenden Funktion oder Methode in einer *Java*-Klasse. Ursprungs- sowie Zielsegment können dabei nicht über Klassen- beziehungsweise Dateigrenzen reichen, da sonst zusätzliche Segmente vonnöten wären.

In Abb. 3.4. ist diese einfache Verknüpfung von Ursprungs- und Zielsegment dargestellt. Ein Segment im *PHP*-Quelltext wird dabei einem Segment des *Java*-Quelltexts zugeordnet. Die einzelne Verbindung sagt aus, dass die Funktionalität, die durch das Segment im *PHP*-Quelltext definiert wird, im verknüpften *Java*-Segment implementiert wird. Ursprungs- und Zielsegment benötigen jeweils nur die Information darüber, mit welchem Segment sie verknüpft sind.

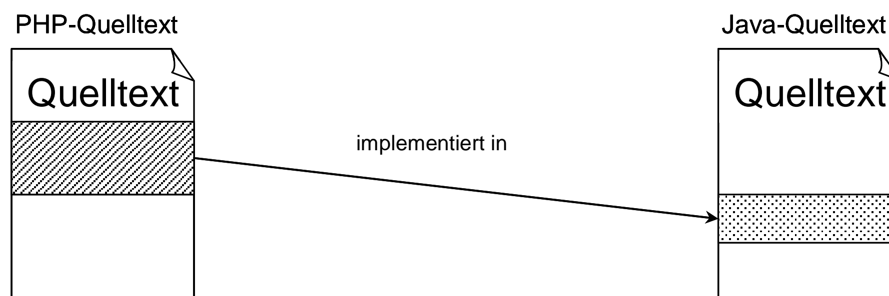


Abb. 3.4.: Die einfache Zuordnung von Ursprungs- zu Zielsegment

1:n-Zuordnungen
In diesem Fall verteilt sich die Implementierung der ursprünglichen, in *PHP* realisierten Funktionalität auf mehrere Zielsegmente, zum Beispiel auf mehrere Methoden einer *Java*-Klasse oder auf verschiedene *Java*-Klassen. Dabei lassen sich die Zielsegmente nicht zu einem zusammenhängenden Block zusammenfassen, so dass eine Mehrfachzuordnung notwendig wird. Beispiel dafür wären Funktionen in *PHP*, die durch die untypisierte Sprache Parameter unterschiedlichen Typs akzeptieren können. Bei der Migration in eine stark typisierte Sprache wie *Java* kann dieses Problem entweder durch eine *Wrapper*-Klasse (auch als Adapter oder Hüllklasse bezeichnet, vgl. [Gamma 2004], S. 171ff) gelöst werden, die die verschiedenen, möglichen Eingangsparametertypen unterhalb einer gemeinsamen Schnittstelle für die Funktion kapselt. Das bedeutet, dass es eine Implementierung der Ursprungsfunktionalität gibt, die genau einen Parametertypen kennt. Die Kapselung der unterschiedlichen Ursprungsparameter geschieht dann in jeweils einer eigenen Wrapper-Klasse, die die Schnittstelle des neuen Parametertypen erfüllt.

Eine weitere Möglichkeit ergibt sich aus der Polymorphie, mit der die unterschiedlichen Parametertypen jeweils auf ihre eigene Funktionsimplementierung abgebildet werden können. Letzteres bedeutet, dass das ursprüngliche *PHP*-Segment dann auf die einzelnen typisierten Funktionen abgebildet wird, so dass eine Zuordnung zu mehreren Zielsegmenten benötigt wird.

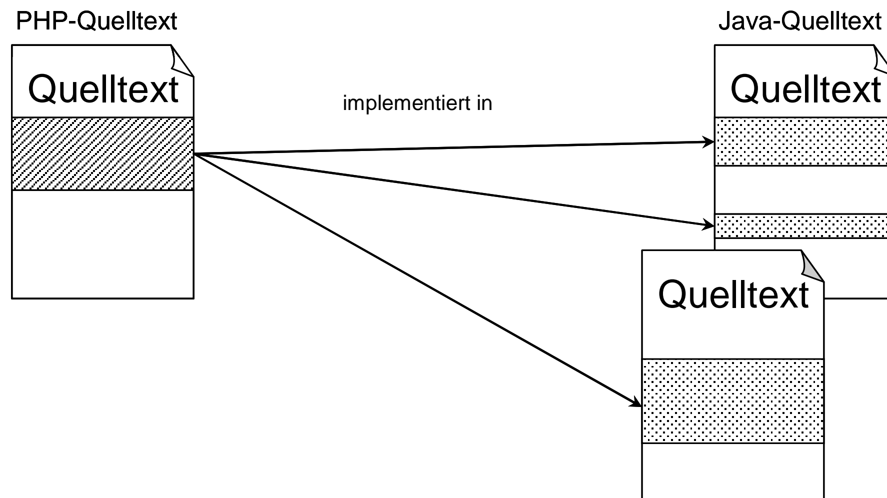


Abb. 3.5.: Zuordnung von einem Ursprungssegment zu *n* Zielsegmenten

Die Zuordnung von einem Ursprungssegment zu mehreren Zielsegmenten ist beispielhaft in Abb. 3.5. dargestellt. Dabei werden einem Segment im *PHP*-Quelltext mehrere Segmente im *Java*-Quelltext zugeordnet. Die Segmente im *Java*-Quelltext können sich dabei, wie oben angedeutet, auf mehrere Quelltextdateien verteilen.

n:1-Zuordnungen
Bei der Migration können Teile des ursprünglichen Quelltexts in den *Java*-Klassen zusammengefasst und konzentriert werden. Somit vereint dann eine einzige *Java*-Klasse die Funktionalität aus verschiedenen *PHP*-Quelltextdateien. Eine Zuordnung mehrerer Ursprungssegmente zu einem Zielsegment wäre hier erforderlich. In Abb. 3.6. ist die Zuordnung von mehreren Segmenten im *PHP*-Quelltext zu einem *Java*-Segment dargestellt. Die Ursprungssegmente können über mehrere *PHP*-Quelltextdateien verteilt liegen. Die verteilt vorliegende Ursprungsfunktionalität wird in einem einzigen Zielsegment zusammengefasst und implementiert.

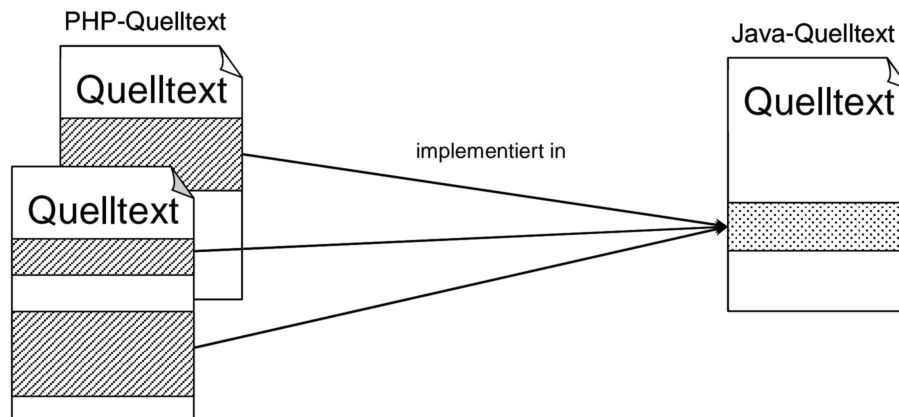


Abb. 3.6.: Zuordnung von verteilter Ursprungsfunktionalität zu einem Zielsegment

Die Ursprungssegmente benötigen nur die Informationen über ein zugeordnetes Zielsegment, was die Zuordnung vereinfacht. Die Verfolgung von Änderungen in den Ursprungssegmenten erfordert allerdings, dass das Zielsegment alle Informationen über die Zustände aller Ursprungssegmente kennen muss.

n:m-Zuordnungen

Im Ursprungssystem verstreut vorliegende Funktionalität wird wiederum in mehrere Zielsegmente migriert. Eine Detaillierung der Zuordnung ist nur dann nicht mehr möglich, wenn mehrere Ursprungssegmente dasselbe Zielsegment besitzen, einzelne dieser Ursprungssegmente aber wiederum mehreren Zielsegmenten zugeordnet sind. Somit lassen sich die einzelnen Zuordnungen nicht in die drei bisher beschriebenen Arten überführen. Diese Art der Zuordnung erscheint wegen ihrer Komplexität nicht wünschenswert und wird daher vom Plug-In nicht unterstützt.

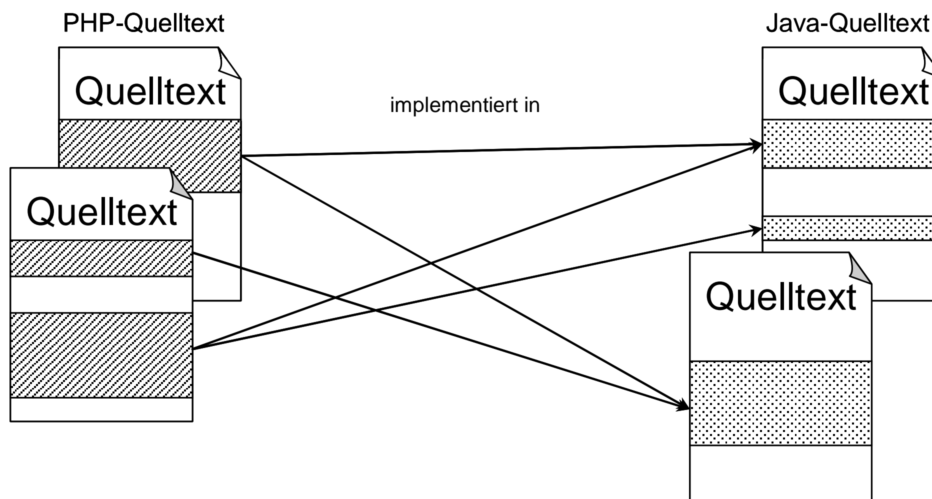


Abb. 3.7.: Zuordnung verteilter Funktionalität im Ursprungssystem zu verteilter Implementierung im Zielsystem

Abb. 3.7. zeigt ein komplexes Zuordnungsgeflecht zwischen einzelnen Ursprungssegmenten im *PHP*-Quelltext und den dazugehörigen Zielsegmenten im *Java*-Quelltext. Dabei überlappen sich die Segmente, die in beiden Systemen eine bestimmte Funktionalität einfassen. Diese Überlappung lässt vermuten, dass entweder in beiden Systemen die Segmente zuviel Funktionalität einschließen und eine feinere Unterteilung dieses Geflecht auflöst, oder dass eine a priori verstreut liegende Funktionalität im Zielsystem ebenso verstreut implementiert wurde.

Letzteres kann zudem ein Hinweis auf eine unsaubere Umsetzung bzw. Architektur im Ursprungs- und im Zielsystem sein.

Eine andere Möglichkeit, eine derart komplexe Form der Zuordnung zu nutzen, wurde schon weiter oben bei der Einführung des Kopplungsmodells beschrieben. Dieses Modell erlaubt die Zuordnung großer Mengen von Ursprungs- und Zielsegmenten zueinander, während die Verbindungen selbst auf eine einzige Kopplung reduziert werden.

3.4. Eindeutigkeit und Änderungsverfolgung

Die Quelltextabschnitte bzw. Segmente, die in beiden Projekten für die Migration herangezogen werden, sind zu einem gegebenen Zeitpunkt durch ihre Position und ihren Inhalt eindeutig bestimmt und identifizierbar. Bei der Zuordnung der Segmente zueinander muss sichergestellt werden, dass die Verbindung zwischen ihnen eindeutig auf die entsprechenden Quelltext-abschnitte abgebildet werden kann.

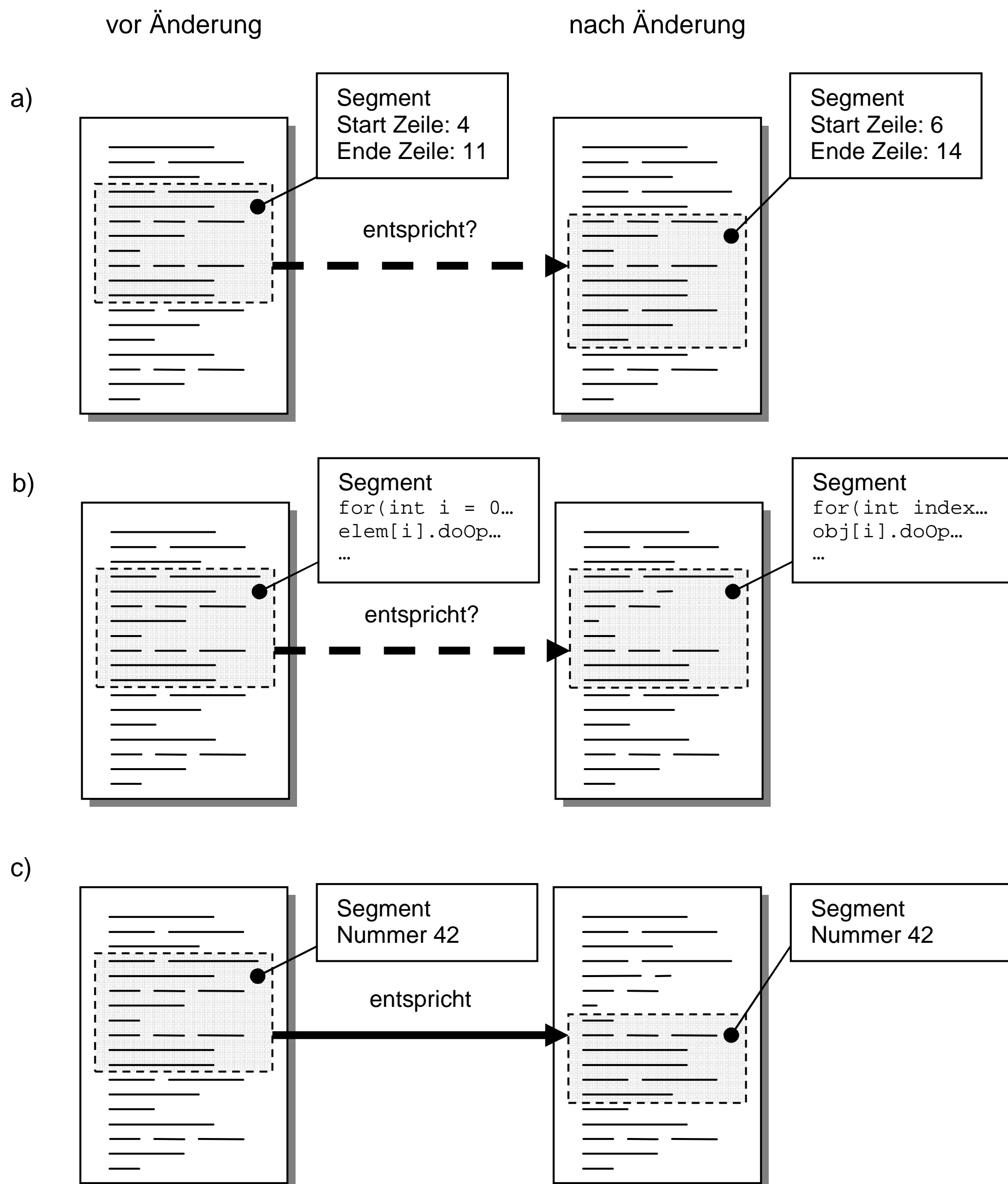


Abb. 3.8.: Auswirkungen von Quelltextänderungen auf die Eindeutigkeit von Segmenten

Änderungen im Ursprungs- und im Zielprojekt können die eindeutige Verknüpfung eines Segments zum zugehörigen Quelltextabschnitt über Position und Inhalt allerdings zerstören. Das Segment mit geändertem Inhalt oder veränderter Position im Quelltext ließe sich nicht mehr dem ursprünglichen Segment zuordnen.

Um die Abhängigkeit eines Segments von einer bestimmten Position oder einem bestimmten Inhalt aufzulösen, wird das Segment nur durch eine Art Start- und Endmarkierung im Quelltext identifiziert. Diese beiden Punkte können durch Änderungen im Quelltext verschoben werden. Deshalb wird für die Segmente ein übergeordnetes Konzept für die Eindeutigkeit benötigt, mit der sich Segmente auch über Änderungen hinaus eindeutig identifizieren lassen. Abb. 3.8. zeigt die genannten Möglichkeiten, Segmente im Quelltext zu identifizieren und stellt beispielhaft dar, wie sich Änderungen im Quelltext auf die Eindeutigkeit von Segmenten auswirken. Links ist das Segment jeweils vor einer Änderung, rechts nach einer Änderung abgebildet (als gestrichelt umrahmter Bereich). Variante a) identifiziert ein Segment anhand der Start- und Endzeile im Quelltext. Diese sind bei Änderungen im Quelltext nicht stabil, zum Beispiel beim Hinzufügen von Quelltext vor einem oder innerhalb eines Segments. Durch die Abweichungen von Start und Ende lässt sich das Segment möglicherweise nicht mehr eindeutig identifizieren, vor allem, wenn noch weitere Segmente im Quelltext vorhanden und von den Änderungen mit betroffen sind.

Variante b) zeigt die Alternative, das Segment durch den enthaltenen Quelltext eindeutig zu identifizieren. Allerdings genügen schon kleinste Abwandlungen, zum Beispiel in der Benennung von Bezeichnern, um aus einem Segment ein neues zu machen. Zudem lassen sich Segmente, die denselben Quelltext enthalten, mit dieser Variante nicht auseinanderhalten. Daher ist diese Variante gänzlich ungeeignet, um Segmente auch durch Änderungen hindurch identifizierbar zu halten.

In der letzten dargestellten Variante c) wird einem Segment daher eine eindeutige Beschriftung, zum Beispiel eine eindeutige Nummer, mitgegeben. Diese bleibt auch nach Anpassungen im Quelltext unverändert und ermöglicht die eindeutige Identifizierung von Segmenten. Um in der Migrationsunterstützung die Eindeutigkeit von Segmenten zu gewährleisten, wurde daher die Umsetzung einer eindeutigen Segmentbeschriftung gewählt.

Die Stabilität der Segment-Identität gegenüber Änderungen ist notwendig, da es sich bei den vorgestellten Projekten einerseits um ein im Betrieb, aber dennoch in der Weiterentwicklung befindliches Ursprungssystem sowie ein unvollständig migriertes Zielsystem handelt. Die Weiterentwicklung stellt die Entwickler des Zielsystems und die Migrationsunterstützung vor eine weitere Herausforderung: einmal implementierte Systemfunktionalität aus dem Ursprungssystem kann nachträglich noch korrigiert und erweitert werden. Die Migrationsunterstützung benötigt daher eine Möglichkeit, nachträgliche Änderungen an Segmenten im Ursprungssystem feststellen zu können, deren Funktionalität schon ins Zielsystem übernommen wurde. Abb. 3.9. stellt die bei Änderungen am Ursprungssegment auftretende Fragestellung dar, ob Änderungen am Ursprung tatsächlich noch durch die Implementierung im Zielsegment abgedeckt sind. Der geänderte bzw. neu eingefügte Quelltext im Ursprungssegment (gepunktet dargestellt) hat möglicherweise keine Entsprechung im Zielsegment. Der im Ursprungssegment eingefasste Quelltextabschnitt stellt damit die Einheit dar, an der auf Änderungen geprüft werden kann. Speziell die Segmente sind dabei von Interesse, die einem Zielsegment zugeordnet sind und als migriert gelten. Nicht zugeordnete Segmente sowie nichtsegmentierter Quelltext brauchen dabei nicht betrachtet werden, da die Migration dieser Systemteile noch aussteht.

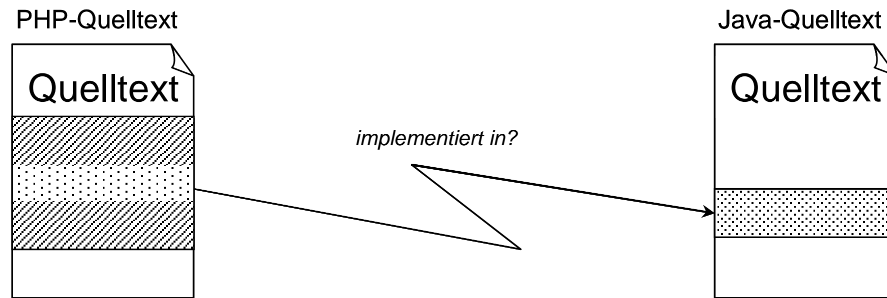


Abb. 3.9.: Änderungen am Ursprungssystem bei migrierter Funktionalität

Dieses Vorgehen erlaubt einige grundlegende Aussagen über das Vorhandensein und die Art von Änderungen. Bei einer Anpassung des Ursprungssegments wird dem Entwickler mitgeteilt, dass am System Korrekturen oder Veränderungen an bereits migrierten Segmenten vorgenommen wurden. Wird Funktionalität (und somit das zugehörige Segment) aus dem Ursprungssystem entfernt, lässt sich dies über die nun unvollständige Zuordnung feststellen, die nur noch das Zielsegment enthält. Das Entfernen des Zielsegments hat zur Folge, dass das Ursprungssegment wieder in den Zustand übergeht, in dem seine Migration noch aussteht.

Die Verfolgung von Änderungen reicht für eine Migrationsunterstützung aus. Nachträgliche Änderungen schon migrierter Funktionalität werden dem Entwickler angezeigt. Eine Differenzierung der Änderungen am Quelltext ist dafür nicht notwendig. Diese Funktionalität bietet das in den Projekten verwendete Versionierungssystem, welches unter anderem für diesen Zweck genutzt wird.

<table>
<tr><th>vor der Änderung</th><th>nach der Änderung</th></tr>
<tr><td>

```
function createSession()
{
  // @segment-begin 68221
  var db = new DBConnection();
  db.createNewSession();

  echo "Session created.";
  // @segment-end 68221
}

function calculateSessionTime()
{
  var session = Session.actual();
  return session.duration();
}
```

</td><td>

```
function login()
{
  // @segment-begin 68221
  var db = fetchDB();
  db.storeSession();

  echo "Session created.";
  // @segment-end 68221
}

function calculateSessionTime()
{
  var session = fetchSession();
  return session.start -
    session.end;
}
```

</td></tr>
</table>

Bsp. 3.10.: Änderungsverfolgung in Segmenten

Das Bsp. 3.10. zeigt einen Ausschnitt aus einem Quelltext des Ursprungssystems: auf der linken Seite befindet sich der Originalquelltext, auf der rechten Seite sind einige Änderungen vorgenommen. Der grau unterlegte Bereich stellt ein Segment dar. Die Änderungen im Segment sorgen dafür, dass das Segment als nachträglich geändert markiert wird (angezeigt durch den gestrichelten Rahmen). Daraufhin kann ein Entwickler prüfen, ob die dahinterliegende Funktionalität eine Änderung am Zielsystem erfordert. Die im Beispiel dargestellte Änderung am nichtsegmentierten Quelltext bleibt der Änderungsverfolgung dabei verborgen. Im Sinne der Migration wurde der entsprechende Quelltextabschnitt noch nicht behandelt, so dass die Änderung sich nicht auf die schon ins Zielsystem übertragene Funktionalität auswirken kann. Es sind allerdings Änderungen am Quelltext möglich, die nicht über die Änderungsverfolgung

entdeckt werden, aber ein schon migriertes Segment und somit die Implementierung im Zielsystem betreffen können, wie am folgenden Beispiel gezeigt wird.

Da die Verfolgung von Änderungen rein auf Quelltextänderungen beruht, sind bei dieser Form der Änderungsverfolgung keine Änderungen entlang des Kontrollflusses feststellbar. Wird innerhalb eines Segments eine Methode aufgerufen, deren Quelltext selbst noch nicht in ein Segment eingefasst und zugeordnet wurde, bleiben Änderungen an dieser Methode unbemerkt. Im Bsp. 3.11. ist links ein Segment im Quelltext dargestellt, in der sich der Aufruf einer unsegmentierten Methode befindet. Obwohl sich die Funktionalität in besagter Methode ändert, führt dies nicht zu einer Änderung des Quelltexts im Segment, so dass die Änderung unbemerkt bleibt. Das Segment ist auf beiden Seiten grau hinterlegt und die Änderungen sind im Quelltext auf der rechten Seite hervorgehoben.

vor der Änderung

```
function login()
{
  // @segment-begin 12786
  var success = 0;
  getUser(success);
  echo "Logging in" + status;
  // @segment-end 12786
}

function getUser(int status)
{
  var db = connectToDatabase();
  status = db.getEntry("default");
}
```

nach der Änderung

```
funtion login()
{
  // @segment-begin 12786
  var success = 0;
  getUser(success);
  echo "Logging in" + status;
  // @segment-end 12786
}

function getUser(int status)
{
  var db =
      new DatabaseAccessor();
  status = db.getEntry("user");
}
```

Bsp. 3.11.: Änderungen an Funktionalität ohne Änderungsverfolgung

Ändert sich hingegen die Signatur der aufgerufenen Methode, ändert dies den Quelltext im Segment. Solange Teile zu migrierender Funktionalität noch im Ursprungs- oder Zielprojekt unsegmentiert oder nicht zugeordnet sind, können Änderungen an der Ursprungsfunktionalität durch die Änderungsverfolgung unbemerkt bleiben. Da das Ziel der Analyse allerdings die möglichst vollständige Segmentierung und Zuordnung beider Projekte hinsichtlich der Funktionalität ist, wird diese Diskrepanz mit Vervollständigung der Migration minimiert.

vor der Änderung

```
function login()
{
  // @segment-begin 12786
  var success = 0;
  getUser(success);
  echo "Logging in " + status;
  // @segment-end 12786
}

function getUser(status)
{
  var db = connectToDatabase();
  status = db.getEntry("default");
}
```

nach der Änderung

```
function login()
{
  // @segment-begin 12786
  var success = 0;
  getUser("default", success);
  echo "Logging in " + status;
  // @segment-end 12786
}

function getUser(name, status)
{
  var db = connectToDatabase();
  status = db.getEntry(name);
}
```

Bsp. 3.12.: Änderungen an Funktionalität mit Änderungsverfolgung

Bsp. 3.12. zeigt eine Änderung an Funktionalität außerhalb eines Segments, von der das Segment betroffen ist. Dabei ändert sich die Signatur eines Methodenaufrufs, weshalb die

Änderungsverfolgung anschlägt und die Änderung an schon migrierter Funktionalität anzeigt. Es sei noch erwähnt, dass das Vorgehen beim Segmentieren einen großen Einfluss auf dieses Problem hat. Wird der Quelltext einer bestimmten Funktionalität des Ursprungsprojekts systematisch entlang des Kontrollflusses im Quelltext segmentiert und ins neue System überführt, so kann die Änderungsverfolgung diese Funktionalität vollständig im Blick behalten.

4. Die Unterstützung der Quelltextmigration als Plug-In

Dieses Kapitel fasst die einzelnen Arbeitsschritte zur Entwicklung der Migrationsunterstützung in Form eines *Eclipse*-Plug-Ins zusammen. Die Grundlage dafür bilden die im vorangegangenen Kapitel erarbeiteten Konzepte. Entwickelt wurde das Plug-In auf den *Eclipse*-Plattformen der Version 3.1.2. sowie 3.2.1. Des Weiteren sind zusätzliche Werkzeuge zur Web-Entwicklung (*WTP – Web Tools Project*) eingebunden. Die Unterstützung der Segmentierung in *PHP*-Editoren basiert auf den Klassen dieser Werkzeuge. Das Plug-In ist in *Java 2, Standard Edition Version 5.0* (vgl. [Sun 2007]) implementiert.

4.1. Anforderungsermittlung

Zu Beginn wurde in Gesprächen mit Projektbeteiligten des *WebMig*-Projekts ein grober Überblick über die geforderte Funktionalität und die grundlegenden Ideen für die Umsetzung der Konzepte ermittelt. Anhand der Beschreibungen wurde ein erster Prototyp erstellt, der exemplarisch einige Grundfunktionen enthielt, unter anderem das Erstellen von Migrationsprojekten, die Segmentierung von Quelltextdateien und die Zuordnung von Segmenten zueinander.

Der Prototyp ermöglichte es, erste Lösungen auszuprobieren und die Anforderungen zu verfeinern bzw. anzupassen. Offene Fragen, wie zum Beispiel die Richtung, in der Segmente einander zugeordnet werden (vom Ursprungsprojekt zum Zielprojekt oder umgekehrt), konnten am Prototyp diskutiert werden. Um die Aufgaben in einen Arbeitskontext zu bringen, wurden *Szenarios* erstellt (vgl. Kapitel 4.2.1.), die den typischen Arbeitsablauf für einen Plug-In-Benutzer darstellen. Nach Züllighoven et al. beschreiben Szenarios *„die aktuelle Arbeitssituation. Dabei liegt das Augenmerk auf den Aufgaben im Anwendungsbereich und der Art und Weise, wie die jeweiligen Aufgaben unter Verwendung von Arbeitsmitteln und Arbeitsgegenständen erledigt werden“ (vgl. [Züllighoven 1998], S. 139).*

Die daraus entstehenden Anforderungen wurden dann in ein Forum aufgenommen, das dazu diente, einzelne Punkte diskutieren zu können sowie erledigte Aufgaben abzuhaken. Das Forum stellte somit die Arbeitsgrundlage für die Entwicklung des Plug-Ins dar, da in ihm noch zu erledigende Aufgaben gesammelt werden konnten. Des Weiteren konnten die Benutzer, die das Plug-In getestet haben, im Forum den Stand sowie Erläuterungen zu gemeldeten Fehlern oder gewünschter Funktionalität nachschlagen.

Um neue Funktionalität – vor allem in der Prototypenphase – frühzeitig testen zu können, wurde während der Entwicklungsphasen regelmäßig ein neues *Release*[1] erstellt. Dabei wurde versucht, die Abstände zwischen den Releases möglichst kurz (innerhalb 1-2 Wochen) zu halten. Ein regelmäßiges, wöchentliches Release wurde dem Umstand geopfert, dass die Entwicklungsphasen sich mit anderen Aufgaben an dieser Arbeit vermischten. Nach Erstellung des Releases wurde dies auf der projekteigenen Internetseite zum Herunterladen bereitgestellt sowie die am Projekt beteiligten Personen per E-Mail über die neue Version informiert.

Auf der Projektinternetseite findet sich ein Änderungsprotokoll, ein so genanntes *Changelog*, in dem die Änderungen von Version zu Version verfolgt werden können. Zusätzlich wurde in den E-Mails kurz zusammengefasst, was sich seit der letzten Version geändert hat, was neu dazugekommen ist und welche Fehler behoben wurden.

Das Plug-In in der Anwendung

Während der Entwicklung ergaben sich weitere Problemstellungen bei der Umsetzung der geforderten Funktionalität. Diese kristallisierten sich in Zusammenarbeit und beim Testen mit den Benutzern des Plug-Ins heraus. Insbesondere geschah dies im Zuge einer Studienarbeit von Alla Remfert, deren Aufgabe darin bestand, den Migrationsfortschritt für einen ausgewählten Teil des *CommSy*-Projekts zu analysieren. Viele ihrer Anmerkungen und Verbesserungsvor-

[1] Ein *Release* bezeichnet die Veröffentlichung einer Softwareversion.

schläge sind in die Ausarbeitung des Plug-Ins sowie in die Entscheidungen eingeflossen, wie und an welchen Stellen in der *Eclipse*-Umgebung die Funktionalität des Plug-Ins angeboten wird.

4.2. Benutzungsmodell

Das Benutzungsmodell dient der Beschreibung der Handhabung des Plug-Ins durch den Benutzer. Es wird zum einen dafür verwendet, um eine Diskussionsgrundlage zu haben, mit der die Gestaltung des Plug-Ins zusammen mit den zukünftigen Benutzern durchgeführt werden kann. Zum anderen ermöglicht es eine angemessene Umsetzung der im Kontext der Migrationsanalyse durchgeführten Aufgabenstellungen in Oberflächenfunktionalität. Im objektorientierten Konstruktionshandbuch findet sich folgende Definition:

> *„Ein Benutzungsmodell ist ein fachlich orientiertes Modell darüber, wie Anwendungssoftware bei der Erledigung der anstehenden Aufgaben im jeweiligen Einsatzkontext benutzt werden kann.*
>
> *Das Benutzungsmodell umfasst eine Vorstellung von der Handhabung und Präsentation der Software, aber auch von den fachlichen Gegenständen, Konzepten und Abläufen, die von der Software unterstützt werden. […]" (vgl. [Züllighoven 1998], S. 71)*

Aus den Gesprächen mit den Benutzern wurde ein Arbeitsablauf konstruiert, der die Konzepte der Quelltextsegmentierung und Zuordnung von Segmenten zueinander in den Kontext eines Entwicklerarbeitsplatzes einbettet. Diese Einbettung und die dafür benötigte Funktionalität werden im Folgenden anhand von Szenarios entwickelt. Dabei werden die Arbeitsschritte betrachtet, die ein Plug-In-Benutzer durchläuft, um bei der Migration ermitteln zu können, inwieweit ein System schon in ein anderes überführt wurde. Die im Folgenden dargestellten Szenarios dienen in erster Linie der Ausarbeitung der in Kapitel 4.3.4. beschriebenen technischen Realisierung der Benutzungsschnittstelle des Plug-Ins. Die einzelnen Arbeitsschritte werden dafür an geeigneten Stellen in die IDE integriert, um den Arbeitsablauf des Benutzers zu unterstützen.

Abb. 4.1. zeigt in einem so genannten *Übersichtsszenario* (vgl. [Züllighoven 1998], S. 611) die Arbeitsschritte eines Plug-In-Benutzers, der den Fortschritt der Migration zwischen zwei Projekten A und B analysieren möchte. Wie in der Abbildung dargestellt, beginnt die Analyse mit dem Erstellen eines Migrationsprojekts (Schritt 1). Die zugrunde liegenden Projekte (Projekt A und B) werden dabei miteinander verbunden. Die Beziehung, die zwischen den beiden Projekten hergestellt wird, weist einem Projekt die Rolle des Ursprungsprojekts, dem anderen Projekt die Rolle des Zielprojekts zu.

Die Quelltexte beider Projekte werden segmentiert, das heißt, der Quelltext wird in Abschnitte unterteilt, die jeweils einer spezifischen Funktion im jeweiligen Softwaresystem entsprechen (Schritt 2). Diese Segmente werden dann einander zugeordnet. Dabei werden Verbindungen hergestellt zwischen den Segmenten des Zielprojekts (Projekt B) und den Segmenten des Ursprungsprojekts (Projekt A), die die von A nach B migrierte Funktionalität enthalten (Schritt 3). Segmente, deren Funktionalität nicht ins Zielsystem migriert wird, werden in diesem Schritt als obsolet markiert. Über die so erhaltenen Segmente und Beziehungen zwischen den beiden Projekten ergeben sich für die segmentierten und nicht segmentierten Bereiche der Projekte verschiedene Zustände hinsichtlich der Migration. Aus diesen unterschiedlichen Zuständen lässt sich dann eine statistische Auswertung erstellen (Schritt 4). Diese Auswertung enthält die Kennzahlen, die eine Aussage über den Fortschritt der Migration erlauben.

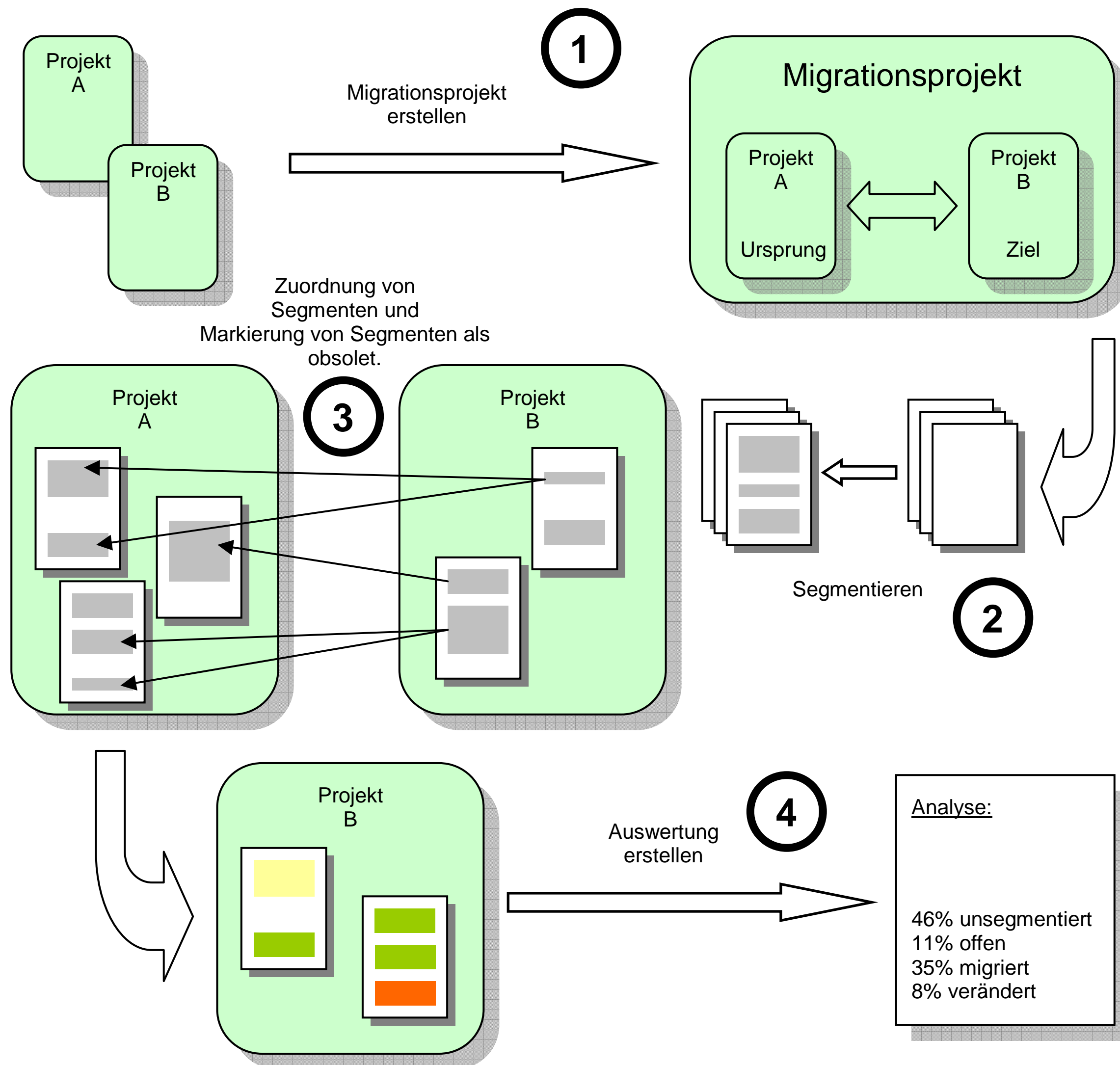


Abb. 4.1.: Übersichtsszenario für die Migrationsanalyse

In den folgenden Absätzen werden die einzelnen Schritte des Arbeitsablaufs detaillierter betrachtet und jeweils in eigene Szenarios eingefasst. Der Arbeitsablauf ist dabei in die vier Teile Projekterstellung, Segmentierung und Zuordnung sowie Analyse des Migrationsprojekts unterteilt. Aus den Szenarios ergeben sich die konkreten Anforderungen, welche Funktionalität das Plug-In an der Oberfläche von *Eclipse* anzubieten hat. Daraus lässt sich die technische Realisierung der einzelnen Anwendungsteile des Plug-Ins ableiten, die in Kapitel 4.3. beschrieben wird.

4.2.1. Szenarios

Um von den im oben beschriebenen Arbeitsablauf enthaltenen Arbeitsschritten zum Benutzungsmodell zu kommen, werden die einzelnen Aufgaben des Benutzers in Form von so genannten *Aufgabenszenarios* (vgl. [Züllighoven 1998], S. 612) beschrieben. Diese Szenarios dienen dazu, einen Arbeitsschritt so detailliert zu beschreiben, dass sich daraus der Teil des Benutzungsmodells für den jeweiligen Schritt ableiten lässt. Während im Arbeitsablauf allgemein das Vorgehen beim Segmentieren und Migrieren beschrieben ist, enthalten die Szenarios den Ort in der Arbeitsumgebung und die Form der Einbindung der jeweiligen Funktionalität. Auf diese detaillierte Beschreibung basiert die technische Realisierung.

Schritt 1: Erstellen eines Migrationsprojekts
Die Grundlage für eine Migration bilden zwei Projekte, von denen eines im Laufe der Migration in das andere überführt wird. Die Arbeit an der Analyse der Migrationsabdeckung beginnt dementsprechend damit, diese zwei Projekte einander zuzuordnen. Der Benutzer stellt eine gerichtete Beziehung zwischen den beiden Projekten her. Ein Projekt fungiert als Ursprungsprojekt und das andere Projekt als Zielprojekt der Migration.

Szenario: Erstellen eines Migrationsprojekts
Der Benutzer erstellt ein neues Migrationsprojekt und wählt aus einer Projektliste Ursprungs- und Zielprojekt der Migration aus. Dazu benötigt er eine Auflistung der an seinem Arbeitsplatz verfügbaren Projekte. Die Reihenfolge, in der der Benutzer die Projekte auswählt, gibt die Zuordnung der Projekte als Ursprung und Ziel der Migration vor. Dabei ist das erste Projekt das Ursprungs- und das zweite das Zielprojekt der Migration. Zum Abschluss der Erstellung eines Migrationsprojekts werden die beteiligten Projekte markiert und sind damit für den Benutzer als Ursprung und Ziel erkennbar.

Schritt 2: Segmentieren des Quelltexts
Um einzelne Teile von Funktionalität im Ursprungs- und Zielprojekt identifizieren und markieren zu können, fasst der Benutzer die dazugehörigen Quelltextabschnitte in beiden Projekten in die in Kapitel 3 beschriebenen Segmente ein. Diese Segmente bilden die Grundlage für die nachfolgenden Arbeitsschritte.

Szenario: Segmentieren des Quelltexts
Der Benutzer durchsucht in beiden Projekten die Quelltexte nach Abschnitten, die eine bestimmte Funktionalität enthalten. Diese Quelltextabschnitte markiert er im Editor und fasst dann den jeweiligen Abschnitt in ein Segment ein. Das Segment wird daraufhin mit Segmentierungsinformationen versehen, die für den Benutzer gut sichtbar am Segment stehen. Auf vorhandene Segmente kann der Benutzer direkt im Quelltext zugreifen.

Durch die Segmentierung erhalten die in Segmente eingefassten Quelltextabschnitte einen Zustand, der ihren jeweiligen Stand hinsichtlich der Migration und Änderungsverfolgung widerspiegelt. Dieser Zustand wird dem Benutzer am Segment im Quelltext angezeigt.

Schritt 3: Segmente zuordnen oder als obsolet markieren
Zwischen einem Segment im Ursprungsprojekt und den entsprechenden Segmenten im Zielprojekt stellt der Benutzer eine Verbindung her, wenn die Zielsegmente die Funktionalität des Ursprungssegments implementieren. Diese Verbindung hat im Kontext der Migration die Bedeutung, dass das entsprechende Ursprungssegment als migriert gilt.

Szenario: Segmente zuordnen
Um ein Segment im Zielprojekt einem Segment im Ursprungsprojekt zuzuordnen, ruft der Benutzer im Quelltext auf dem Segment einen Zuordnungsdialog auf. Der Dialog bietet dem Benutzer eine Auflistung der im Ursprungsprojekt erstellen Segmente an, aus denen dieser das zuzuordnende Segment auswählt. Mit der Zuordnung werden die Segmentierungsinformationen um die Informationen der Zuordnung erweitert, die dort für den Benutzer sichtbar erscheinen.

Der Quelltext des Ursprungsprojekts enthält Abschnitte, in denen sich ausschließlich technologiespezifische Funktionalität befindet, die nur im Ursprungsprojekt benötigt wird und nicht migriert werden muss. Der Benutzer kann derartige Segmente als *obsolet* markieren und somit aus der Migrationsanalyse entfernen.

Szenario: Segmente als obsolet markieren
Um ein Segment als *obsolet* zu markieren, ruft der Benutzer die Markierfunktion im Quelltext auf einem Segment auf. Die Markierung des Segments wird dem Benutzer sofort angezeigt.

Schritt 4: Analyse des Migrationsfortschritts
Neben Segmentierung, Zuordnung und Änderungsverfolgung im Migrationsprojekt werden noch analytische Werkzeuge für eine Unterstützung der Migration benötigt. Dabei sind vor allem die Abdeckung der Quelltexte durch Segmente, die Anzeige noch offener (bzw. nicht migrierter) Segmente sowie die Anzeige veränderter Segmente von Interesse.

Der Benutzer prüft im Laufe eines Migrationsprojekt regelmäßig den Fortschritt der Migration. Der Überblick über die Migration enthält Informationen, welche Teile des Ursprungsprojekts noch nicht betrachtet (bzw. segmentiert) wurden, welche noch nicht im Zielsystem implementiert wurden, welche migriert wurden und welche seit ihrer Migration nachträglichen Änderungen im Ursprungsprojekt unterlagen. Um diesen Überblick zu erzeugen, lässt er eine Auswertung über das Migrationsprojekt laufen. Der Überblick setzt sich dann aus den Analyseergebnissen zusammen. Dazu gehören unter anderem die prozentuale Abdeckung und Zuordnung von Segmenten zueinander.

Szenario: Analyse des Migrationsfortschritts
Der Benutzer startet eine Analyse des Migrationsfortschritts auf einem an der Migration beteiligten Projekt. Die Ergebnisse der Analyse werden in einem separaten Informationsfenster angezeigt und dort in einer Baumstruktur aufbereitet, in der der Benutzer leicht die Ergebnisse den Bereichen des Projekts zuordnen kann. In dieser Baumstruktur ist die mittlere prozentuale Abdeckung des Quelltexts mit Segmenten an den jeweiligen Dateien und Ordnern dargestellt. Quelltextdateien sind mit den in ihnen enthaltenen Segmenten und deren Migrationsstatus aufgeführt. Des Weiteren hat der Benutzer die Möglichkeit, von den Einträgen in der Auswertung direkt zu den zugehörigen Segmenten in den Projekten zu springen.

Zusätzliche Funktionalität im Plug-In
Während der Arbeit an der Migration und deren Auswertung wird der Benutzer durch Funktionalität des Plug-Ins unterstützt, die für den oben beschriebenen Arbeitsablauf nicht erforderlich ist, aber dennoch die Arbeit erleichtert.

Die Zuordnung von Segmenten zueinander verbindet Quelltextabschnitte im Zielprojekt eindeutig mit Quelltextabschnitten im Ursprungsprojekt. Damit kann der Benutzer von einem Zielsegment in das zugeordnete Ursprungssegment springen, um zum Beispiel die dortige Implementierung mit der Umsetzung im Ziel zu vergleichen. Diese Funktion wird direkt auf den Segmenten des Zielprojekts aufgerufen.

Szenario: Navigieren zwischen Segmenten
Der Benutzer ruft im Quelltext auf einem Zielsegment einen Sprung zum Ursprungssegment auf. Dieser Sprung öffnet die zum Ursprungssegment gehörende Quelltextdatei und bringt das Segment in die Ansicht im Editor.

Sind Segmente einander zugeordnet, so gilt das jeweilige Ursprungssegment als in den dazugehörigen Zielsegmenten implementiert bzw. migriert. Wird das Ursprungssegment nach einer Zuordnung verändert, so setzt die Änderungsverfolgung den Zustand der Zielsegmente auf *nachträglich verändert*. Für den Benutzer bedeutet dies eine Arbeitsanweisung, die Änderungen

am Ursprung zu überprüfen und bei Bedarf die Implementierung in den Zielsegmenten entsprechend anzupassen. Ist dies geschehen, validiert der Benutzer die nachträgliche Änderung und setzt den Zustand der Zielsegmente auf *migriert* zurück.

Szenario: Validieren von nachträglich veränderten Segmenten
Der Benutzer hat ein als nachträglich verändert markiertes Segment und die dazugehörige Implementierung im Zielprojekt überprüft. Nun möchte er die Verbindung mit dem Zielsegment wieder herstellen und damit die Änderungen am Ursprungssegment validieren. Um ein Zielsegment zu validieren, ruft der Benutzer auf dem Zielsegment im Quelltext die Validierungsfunktion auf. Diese Funktion passt die Änderungsverfolgung auf das geänderte Ursprungssegment an und setzt damit den Zustand auf *migriert* zurück. Die Zustandsänderung ist für den Benutzer sofort sichtbar.

Eine weitere Funktion, die dem Benutzer die Arbeit erleichtert, stellt das automatisierte Segmentieren von *Java*-Dateien dar. Mit dieser Funktion kann der Benutzer eine Quelltextdatei vom Plug-In vorsegmentieren lassen. Dabei werden im Quelltext automatisch Methoden in Segmente eingefasst.

Szenario: Automatisiertes Segmentieren einer *Java*-Datei
Der Benutzer ruft aus der Projektübersicht heraus auf der zu segmentierenden Datei die Segmentierfunktion auf. Diese Funktion fasst automatisch die in der Quelltextdatei enthaltenen Methoden in Segmente ein. Ist die Quelltextdatei in einem Editor geöffnet, werden die Segmente dem Benutzer angezeigt.

4.2.2. Darstellung der Segmentierungsinformationen

Neben der Unterstützung der Segmentierung und Zuordnung von Segmenten zueinander müssen die verschiedenen Informationen, die sich daraus ergeben, für den Benutzer visualisiert werden. Zu diesen Informationen gehören die Verfolgung von Änderungen, die Zuordnung von Segmenten zueinander sowie die Dichte der Segmentierung in einer Quelltextdatei. Im Plug-In müssen dafür geeignete Mittel zur Verfügung gestellt werden, um diese speziell für die Migration wichtigen Informationen hervorzuheben.

Die Darstellung der Segmentierungsinformationen stellt eine gesonderte Anforderung an das Plug-In dar, die sich nicht auf ein einzelnes Benutzungsszenario beschränken lässt, sondern vielmehr alle Arbeitsschritte begleitet. Daher wird die Darstellung der Segmente in diesem Kapitel gesondert von den Szenarios behandelt, da sie ebenso zum Benutzungsmodell gehört wie die in den Szenarios beschriebenen Arbeitsschritte.

Der Benutzer benötigt die Visualisierung der Informationen zur Orientierung bei der Migration und Steuerung seiner Arbeit. Um Migrationsinformationen in den Quelltextdateien hervorheben zu können, wird durch das Plug-In ein neuer Editor eingebunden, der speziell auf die Darstellung dieser Informationen zugeschnitten ist. Diese Entscheidung wurde getroffen, um eine Überschneidung mit anderen Darstellungsarten, zum Beispiel dem JDT-eigenen *syntax highlighting* von Kommentarblöcken, mit jetzigen oder zukünftigen Versionen der in der IDE verwendeten Editoren zu vermeiden. Für einzelne Aspekte der Darstellung wurden Alternativen diskutiert, zum Beispiel das Ausblenden von Segmentierungsinformationen in den von *Eclipse* bereitgestellten Editoren.

Darstellung der Daten

Die Informationen über einzelne Segmente werden in Kommentarblöcken abgelegt. Einzelne Einträge sind durch ein einleitendes Schlüsselwort (*Kommentarmarke* genannt, vgl. Kapitel 4.3.2.) gekennzeichnet. Um diese Informationen von den übrigen, nicht zur Migration gehörenden Kommentaren abzugrenzen, werden diese Segmentierungsinformationen durch Fettschrift und Färbung hervorgehoben (das so genannte *syntax highlighting*). Alle Segmen-

tierungsinformationen treten damit aus dem übrigen Quelltext hervor und sind leicht zu identifizieren und auszulesen.

Hervorheben von Segmenten
Im speziellen Migrationseditor (vgl. Kapitel 4.3.2.) werden Segmente geeignet vom übrigen Quelltext hervorgehoben. Dazu werden die jeweiligen Bereiche des Quelltexts farblich hinterlegt. Zwei wesentliche Ideen werden damit umgesetzt:

- Segmente sollen leicht im Quelltext zu identifizieren sein und Anfang und Ende eines Segments - und somit dessen Ausdehnung - deutlich erkennbar sein.
- der Status eines Segments soll dargestellt werden und somit auf einen Blick ersichtlich sein. Der Status eines Segments beinhaltet dabei, ob das Segment schon zugeordnet wurde, ob sich seit der Zuordnung Änderungen im Ursprungssegment ergeben haben oder ob ein Segment für die Migration obsolet ist.

Die farbliche Hervorhebung ist dabei eine der Möglichkeiten, um diese Informationen gut sichtbar und interpretierbar darzustellen. Sie ist allerdings dahingehend zu bewerten, dass eine Farbgebung alleine nicht die optimale Lösung ist, wenn man Benutzer mit einbezieht, die durch eine Farbschwäche bei der Benutzung des Plug-Ins benachteiligt werden. Eine zusätzliche textuelle Hervorhebung oder Erweiterung der Darstellung bietet dafür eine Lösung. Im Abschnitt über die Realisierung der Migrationsanalyse (vgl. Kapitel 4.3.8.) wird daher eine entsprechende Erweiterung der dargestellten Informationen um lesbaren Text beschrieben.

Durch die Farbgebung müssen im Prinzip die Zustände *unsegmentiert*, *obsolet*, *nicht zugeordnet*, *zugeordnet* und *nachträglich verändert* visualisiert werden. Die Bedeutung der einzelnen Zustände und die gewählte Farbgebung werden im Folgenden beschrieben:

- der Zustand *unsegmentiert* umfasst die Bereiche im Quelltext, die nicht in Segmente eingefasst sind. Da diese Bereiche durch die Abwesenheit von Segmentierung charakterisiert werden, wurde der Hintergrund weiß belassen, als Metapher für einen hinsichtlich der Segmentierung noch nicht bearbeiteten Bereich.
- Bereiche des Quelltexts, die nicht migriert werden, zum Beispiel technologie-spezifische Quelltextanteile, werden als *obsolet* markiert. Dies geschieht in Abgrenzung zu den *unsegmentierten* Bereichen, bei denen eine Klassifizierung noch aussteht. Angelehnt an die Metapher des „Ausgrauens“ für deaktivierte oder inaktive Oberflächenelemente – wie sie zum Beispiel in *Eclipse* verwendet wird - werden *obsolete* Bereiche mit einem hellen Grauton hinterlegt.
- für Zustände segmentierter Bereiche wurde die Farbwahl ähnlich den Signalen einer Ampel umgesetzt (grün = alles in Ordnung, gelb = Warnung, rot = Gefahr). Ist ein Segment im Zielsystem einem anderen Segment aus dem Ursprungssystem zugeordnet, so gilt die Funktionalität dieses Ursprungssegments im Zielsegment implementiert und migriert. Stimmen die Prüfsummen (vgl. Kapitel 4.3.3.) überein, so wurde seit der Migration in dem Ursprungssegment nichts mehr verändert. Im Sinne der Migration ist mit den beiden Segmenten somit alles in Ordnung, was durch eine grüne Farbgebung des Untergrunds dargestellt wird.
- Segmente im Quelltext, die noch nicht zugeordnet wurden, stehen noch zur weiteren Bearbeitung aus. Diese kann entweder eine Zuordnung zu einem Segment im jeweils anderen Projekt im Zuge der Migration sein oder eine Markierung des Segments als obsolet. Um diese offenen Segmente hervorzuheben, wird der Farbton Gelb gewählt. Diese Farbe charakterisiert eine „milde“ Warnung, dass an dieser Stelle noch etwas zu tun ist.
- wurde ein Segment einmal zugeordnet, so gilt dieses Segment und das ihm zugeordnete als migriert. Werden dann im Ursprungssegment Änderungen vorgenommen, so nimmt das Plug-In an, dass die ursprüngliche Implementierung im Zielsegment nicht mehr zutreffend ist und eine Korrektur erforderlich ist. Das Ignorieren der Änderungen im Ursprungssystem kann zu unerwünschtem und

undefiniertem Verhalten im Zielsystem führen. Einzelne, schon ins Zielsystem migrierte Teile der Ursprungsfunktionalität können mit dem aktuellen Ursprungssystem auseinander laufen. Um die Dringlichkeit einer Überprüfung dieser Segmente zu kommunizieren, wird diesen Segmenten die „starke“ Warnfarbe Rot unterlegt.

4.3. Technische Realisierung des Plug-Ins

In diesem Kapitel wird die Umsetzung der Anforderungen beschrieben. Einer Einführung über das Einbinden von Funktionalität in die Oberfläche von *Eclipse* folgt die Umsetzung des in Kapitel 3.2. entworfenen Segmentmodells sowie des im vorangegangenen Kapitel entworfenen Benutzungsmodells anhand der Szenarios. Für die einzelnen Szenarios beschreibt jeweils ein eigenes Kapitel die technische Umsetzung der dazugehörigen Funktionalität sowie die Einbindung in die Entwicklungsumgebung.

4.3.1. Einbinden von Funktionalität in *Eclipse*

Technisch geschieht die Anbindung des Plug-Ins an die Benutzungsoberfläche und den Kern von *Eclipse* über *extension points*, auch *Einhängepunkte* oder *Einstiegspunkte* genannt. Plug-Ins können sich durch die im *extension point* eines vorhandenen Plug-Ins definierten Schnittstellen in dieses Plug-In einhängen und somit dessen Funktionalität nutzen und erweitern. Im *Eclipse*-Plattformhandbuch von Brüssau et al. beschreiben die Autoren *extension points* folgendermaßen:

> *„Was ist ein Extension Point? Ein Extension Point ist ein Einstiegspunkt, an dem sich ein Plug-In in ein anderes Plug-In einhängen kann. Wenn ein Plug-In – nennen wir es Plug-In A – sich in ein anderes Plug-In – nennen wir es Plug-In B – einhängen will, muss B einen Extension Point bereitstellen und A für diesen Extension Point eine Extension implementieren. Extension Point und Extension werden jeweils im Manifest*[1] *des Plug-Ins deklariert.“ (vgl. [Brüssau 2006], S. 435)*

Im Fall des *Software Migration Plug-Ins* entspricht B den Plug-Ins der Entwicklungsumgebung, in denen sich die Migrationsunterstützung A über die Implementierung der bereitgestellten *extension points* von B einklinkt. Die *extension points* sind also Schnittstellen der Plug-Ins zur Außenwelt, über die weitere Plug-Ins durch die Implementierung dieser Schnittstelle (eine *extension*) die angebotene Funktionalität nutzen können.

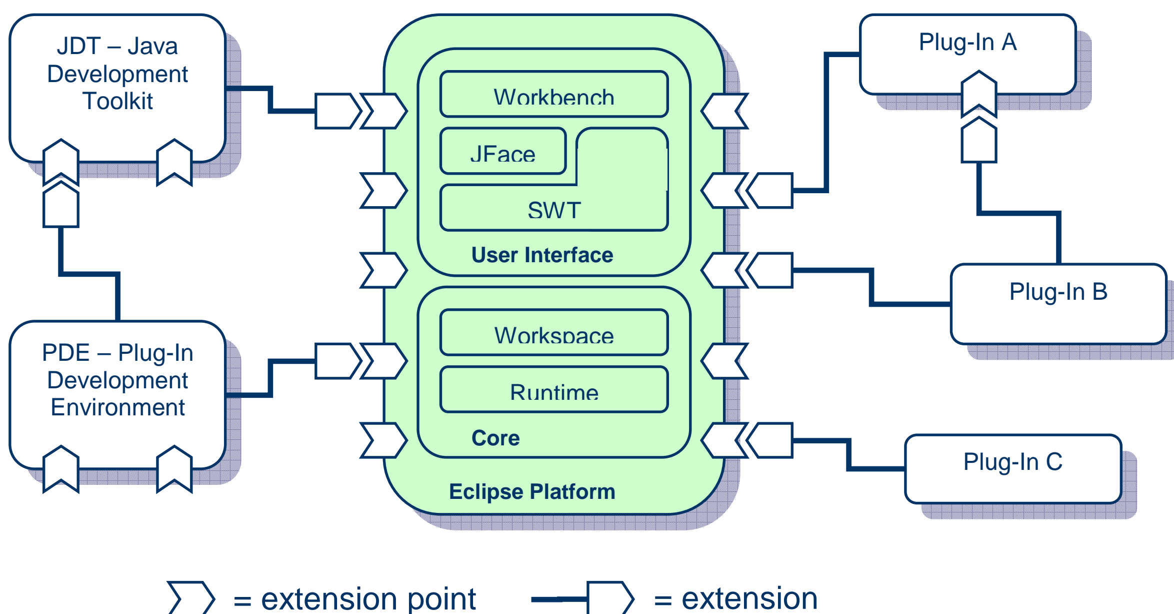


Abb. 4.2.: *Eclipse*-Plattform mit *extension points*

[1] Alle Einstiegspunkte, die ein Plug-In implementiert oder anderen Plug-Ins anbietet, werden im so genannten Manifest (`plugin.xml`) des Plug-Ins definiert (*Eclipse* Version 3.1.2 und 3.2.1).

Abb. 4.2. zeigt den grundsätzlichen Aufbau der *Eclipse*-Plattform hinsichtlich der Plug-Ins und *extension points*: im Kern befinden sich Kern- und Benutzungsoberflächenkomponenten, aus denen die IDE aufgebaut wird und die deren Verhalten steuern. Editoren und weitere Werkzeuge werden über die nach außen angebotenen *extension points* eingeklinkt. Die darüber eingebundenen Plug-Ins müssen dabei selbst nicht unbedingt Endstücke darstellen, sondern können ebenso weitere *extension points* anbieten. Das ermöglicht anderen Plug-Ins, Funktionalität des *Eclipse*-Kerns, der *Eclipse*-Oberflächenbibliotheken sowie schon vorhan-dener Plug-Ins zu verwenden.

In den folgenden Kapiteln werden die Einstiegspunkte erläutert, über die die jeweils beschriebene Funktionalität in die IDE eingebunden wird.

Einbindung von Kontextfunktionalität über IAction

Das Plug-In benutzt verschiedene Einstiegspunkte von *Eclipse*, um seine Funktionalität dem Benutzer zur Verfügung zu stellen. Da ein großer Teil dieser Funktionalität in Form von Kontextfunktionen angeboten wird, soll diese Form der Einbindung hier kurz erläutert werden.

Bei der Umsetzung des Plug-Ins wurde aus Gründen der besseren Testbarkeit und klaren Trennung von Oberflächenanbindung und Funktionalität für jede Oberflächenaktion eine eigene Klasse geschrieben. Die Aufgabe der jeweiligen Aktionsklasse besteht darin, den Aufruf von der *Eclipse*-Oberfläche entgegenzunehmen, eine Instanz der an die jeweilige Aktion gebundene Plug-In-Funktion zu besorgen und mit dieser dann den Aufruf auszuführen.

Die Aktionsklassen müssen die für allgemeine Oberflächenaktionen vorgesehene Schnittstelle IAction implementieren. Diese Schnittstelle bekommt beim Aufruf der konkreten Oberflächenaktion entsprechende Informationen aus dem Kontext hineingereicht. Je nach Kontext existieren spezialisierte Aktionsschnittstellen (abgeleitet von IActionDelegate), die beim Aufruf zusätzliche Informationen bereitstellen. So werden zum Beispiel in Editoren die jeweilige Editorinstanz sowie eine ggf. vorhandene Quelltextselektion über die IEditorActionDelegate-Schnittstelle mitgegeben. Die Editorinstanz kann dann benutzt werden, um den im Editor angezeigten Quelltext abzufragen und mithilfe der Selektion den ausgewählten Abschnitt zu bestimmen.

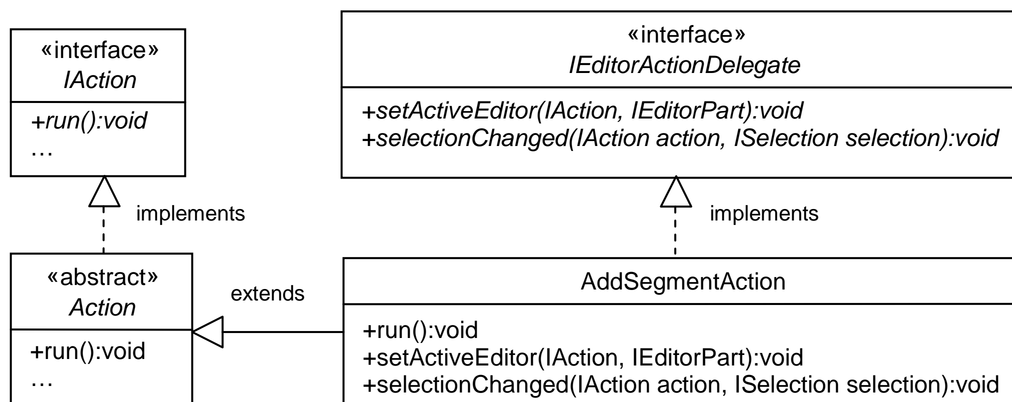


Abb. 4.3.: Implementierung einer IAction am Beispiel der AddSegmentAction

In Abb. 4.3. ist beispielhaft das Klassendiagramm für die AddSegmentAction dargestellt. Die AddSegmentAction ist von der abstrakten Action-Klasse abgeleitet, die die für Oberflächenaktionen geforderte Schnittstelle implementiert. Die AddSegmentAction kann somit als Kontextfunktion registriert werden. Um Funktionalität bereitzustellen, überschreibt die AddSegmentAction die Methode run. Dort wird eine Instanz der Klasse erzeugt, die für das Segmentieren im Quelltext zuständig ist (vgl. Kapitel 4.3.6.). Diese Instanz benötigt zur Ausführung der Segmentierung noch das Quelltextdokument sowie den Abschnitt, der segmentiert werden soll. Daher implementiert die AddSegmentAction zusätzlich die speziell für

Kontextaktionen in Editoren vorgesehene Schnittstelle IEditorActionDelegate, über die beim Aufruf aus einem Editor heraus der Editor und eine Selektion – sofern vorhanden - in die AddSegmentAction hineingereicht werden.

Alle im Plug-In zur Verfügung gestellten Kontextfunktionen sind wie die hier dargestellte AddSegmentAction über eigene Aktionsklassen eingebunden. Da die Aktionsklassen selber kaum Funktionalität enthalten, mit Ausnahme der Erzeugung und dem Aufruf der eigentlichen Funktionsimplementierung, werden sie in den Kapiteln, die die Umsetzung der einzelnen Funktionen beschreiben, nicht mehr gesondert erwähnt.

4.3.2. Umsetzung des Segmentmodells

Das in Kapitel 3.2. erarbeitete Segmentmodell bildet die Grundlage für die Umsetzung von Zerlegung und Zuordnung von Quelltextabschnitten zueinander. Da die Implementierungen aller weiteren, im Benutzungsmodell beschriebenen Funktionen im Plug-In auf der Umsetzung des Segmentmodells beruhen, soll die Implementierung des Segmentmodells hier als Erstes betrachtet werden.

Abb. 4.4. zeigt das Klassendiagramm der im Plug-In verwendeten Segmentklasse. Das Segment fungiert dabei als reiner Datencontainer zum Ablegen der für die Segmentierung und Zuordnung nötigen Informationen.

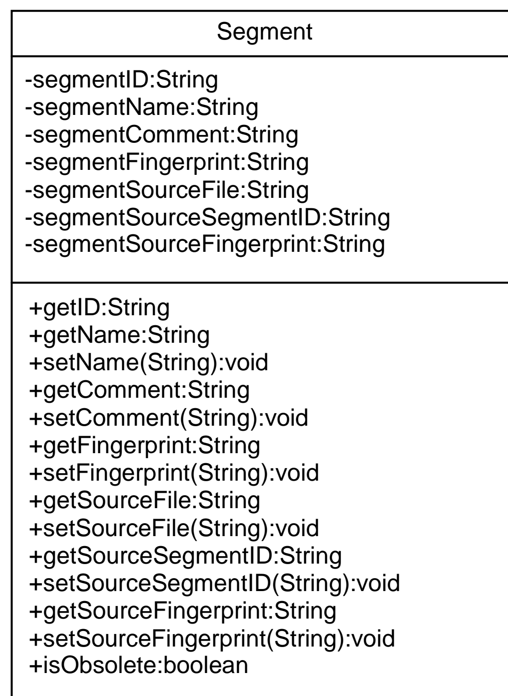


Abb. 4.4.: Die Segmentklasse

Für alle Attribute des Segments gibt es jeweils eine dazugehörige get- und set-Methode. Einzige Ausnahme bildet die Identifikationsnummer (ID) eines Segments, die bei der Erzeugung einer Instanz der Segmentklasse gesetzt wird. Des Weiteren liefert ein Segment die Information, ob es als obsolet markiert ist. Die Markierung lässt sich als solche nicht am Segment setzen, sondern ergibt sich, wenn als Name des Segments das reservierte Schlüsselwort <obsolete> gesetzt wird. Die Attribute, die speziell die Informationen über ein einem Zielsegment zugeordnetes Ursprungssegment enthalten (erkennbar an …Source… im Bezeichner), werden nur in zugeordneten Zielsegmenten benutzt und sind ansonsten leer.

Speichern der Segmentinformationen
Die zu einem Segment gehörenden Informationen wie Identifikationsnummer (ID), Name, zugeordnetes Ursprungssegment, Prüfsumme usw. müssen an einer passenden Stelle im Projekt gespeichert werden. Das Speichern dieser Informationen erlaubt im Verlauf der Migration die Zuordnung der Segmente zueinander, die Navigation zwischen den Segmenten sowie die Änderungsverfolgung an bereits migrierten Segmenten.

Die einzelnen Segmente müssen im Quelltext bezeichnet und mit Informationen versehen werden. Dazu bedient sich das Werkzeug einiger durch Schlüsselwörter gekennzeichneter Kommentare (*Kommentarmarken* genannt), die die Informationen einzelner Segmente in Kommentarform aufnehmen. Die Schlüsselwörter der einzelnen Kommentarmarken und eine Beschreibung der jeweils gespeicherten Segmentierungsinformation findet sich in der Kommentarmarkenreferenz in Anhang C dieser Arbeit.

Als Alternative für die Speicherung von Segmentierungsinformationen kommen *Java*-Annotationen in Frage. Allerdings wurden diese erst mit der Version 1.5 von *Java* eingeführt und hätten somit eine Konfiguration des Plug-Ins hinsichtlich der verwendeten *Java*-Version und der zugehörigen Quelltextrepräsentation erfordert. Da *Java*- und *PHP*-Kommentare denselben Aufbau haben, wurde die Nutzung von Kommentaren bevorzugt. Zu Beginn der Entwicklung wurde der Umstand, dass die *JSP* (*Java Server Pages*) im Zielprojekt wesentliche Funktionalität enthalten, noch nicht berücksichtigt. Die gesonderte Unterstützung von *JSP*-Kommentaren wurde erst später hinzugefügt und floss daher nicht in die Entscheidung mit ein, Kommentare als Speicherort zu nutzen.

Die Single-Source-Lösung
In der ersten Fassung des Plug-Ins wurden alle diese Informationen im Blockkommentar zu Beginn eines Segments gespeichert. Die dadurch im Quelltext verbleibende Segmentierungsinformation genügte damit dem *single source*-Prinzip[1]. Alle für die Segmentierung notwendigen Informationen werden mitsamt den geänderten Quelltextdateien in die gemeinsame Quelltextbasis (zum Beispiel *CVS*[2]) eingebracht.

Durch die Menge an Informationen, die für jedes Segment gespeichert werden müssen, wurde durch die Segmentierung der Quelltext aufgebläht. Vor allem Quelltextdateien mit einer größeren Zahl an Segmenten hatten einen deutlichen Zuwachs an Zeilen. Die größeren Kommentarblöcke minderten zudem die Übersichtlichkeit und Lesbarkeit des Quelltexts. Dieser Umstand macht die Speicherung der vollständigen Segmentinformationen in Kommentaren vor allem für die Entwickler des Ursprungssystems zu einem Ärgernis. Dies ist ein Nachteil der *Single-Source*-Lösung, der dazu führte, dass erst einmal nach Alternativen für die Speicherung von Segmentierungsinformationen gesucht wurde.

Die Zwei-Dateien-Lösung
Da die Informationen über Segmentierung und Verbindung der Quelltexte nur für die Zeit der Migration notwendig sind, ist ein dauerhafter Verbleib der Informationen im Quelltext nicht unbedingt erforderlich. Um während der Migration einen lesbaren Überblick über die Segmentierung des Quelltexts zu erhalten, genügt das Einfassen der Segmente durch einzeilige Kommentare, die eine eindeutige Identifikation eines Segments, zum Beispiel durch die Identifikationsnummer, enthalten und Anfang sowie Ende eines Segments markieren. Um aber den Quelltext ansonsten von Segmentierungsinformationen frei zu halten, werden alle weiteren Informationen in eine gesonderte Datei ausgelagert, die durch geeignete Benennung der Quelltextdatei zugeordnet ist. Vorstellbar wäre hierfür zum Beispiel der Dateiname der Quelltextdatei gefolgt von einer zusätzlichen Dateiendung „.segmentdata“. Über die oben

[1] Als *single source*-Prinzip bezeichnet man in der Softwareentwicklung die Bündelung aller Dokumentationsinformationen in einer Quelltextdatei (vgl. [Pomberger 2002], S. 787 unten).

[2] *Concurrent Versions System* – eine Versionierungssoftware, mit deren Hilfe eine verteilte Softwareentwicklung auf Basis einer gemeinsamen Quelltextsammlung ermöglicht wird. Neben CVS gibt es eine ganze Reihe alternativer Versionierungssysteme, zum Beispiel *Subversion* oder *Perforce*,

genannte Identifikationsnummer werden Segmente und die dazugehörenden Informationen in dieser zweiten Datei miteinander gekoppelt. Das Plug-In kann mithilfe dieser Zusatzinformationen die Visualisierung der Segmentierung und das Navigieren zwischen den Segmenten bewerkstelligen, ohne dass der ursprüngliche Quelltext unnötig aufgebläht wird. Steht das Plug-In einmal nicht zur Verfügung, so können die Segmentierungs- und Verbindungsinformationen dennoch aus den Zusatzdateien rekonstruiert werden.

Die Alternative wurde diskutiert und wieder verworfen. Dem Vorteil, den Quelltext möglichst wenig mit zusätzlichen Segmentierungsinformationen zu kontaminieren, steht der Nachteil gegenüber, dass mit jeder Quelltextdatei eine zusätzliche Segmentierungsdatei anfällt. Diese muss beim Einspielen von Änderungen in die gemeinsame Quelltextbasis berücksichtigt werden. Werden an einem Arbeitsplatz die aktualisierten Segmentierungsinformationen nicht in die Quelltextbasis übernommen, so stehen sie an einem anderen Arbeitsplatz nicht zur Verfügung. Nur die Segmentkommentare, die im Quelltext Anfang und Ende von Segmenten markieren, bleiben erhalten, aber besitzen nun keine Zuordnung mehr zu den detaillierten Informationen.

In einer erweiterten Fassung der Zwei-Dateien-Lösung könnten sogar sämtliche Informationen über die Segmente aus der Quelltextdatei in eine zweite ausgelagert werden, so dass vor allem die Quelltextdateien des Ursprungssystem frei blieben von jeglichen zusätzlichen Segmentierungsinformationen. Die zweite Datei enthielte dann zusätzlich zu den bisherigen Daten noch Anfang und Ende des jeweiligen Segments. Bei dieser Lösung wären, genauso wie beim ersten Vorschlag, jeweils zwei Dateien pro Quelltextdatei zu behandeln; die Entwickler des Ursprungssystems wären gezwungen, diese Dateien bei der Nutzung der gemeinsamen Quelltextbasis mitzuschleppen.

Die völlige Entkopplung des Quelltexts von den Segmentierungsinformationen erschwert zudem die Änderungsverfolgung. Wird in dieser Variante der Zwei-Dateien-Lösung eine Quelltextdatei verändert, zum Beispiel ein Segment gelöscht oder verschoben, so können Quelltext und Segmentierungsinformationen auseinander laufen. Werden in der Segmentdatei Start- und Endzeile der Segmente gespeichert, so kann durch solche Verschiebungen und Löschungen der Quelltext unter den Segmenten durchgeschoben werden, so dass ein Quelltextabschnitt seine ursprüngliche Segmentzuordnung verliert. Des Weiteren kann nicht mehr unterschieden werden, ob ein Segment im Quelltext gelöscht oder nur verschoben wurde. Eine andere Möglichkeit wäre, Anfang und Ende von Segmenten durch kurze Quelltextausschnitte zu identifizieren, die mit in der Segmentdatei abgelegt werden. Diese unterlägen allerdings Änderungen, wie der gesamte übrige Quelltext. Außerdem müssten diese Ausschnitte jederzeit die Bedingung der Eindeutigkeit erfüllen, um Segmente eindeutig identifizieren zu können. Dadurch wird die Auswahl an in Frage kommenden Start- und Endstücken weiter eingeschränkt, was zu Einbußen hinsichtlich der Flexibilität dieser Alternative führt. Die umständliche Handhabung der Speicherung von Segmentierungsinformationen in einer zweiten Datei führte dazu, dass diese Lösung bei der Entwicklung des Plug-Ins nicht weiter in Betracht gezogen wurde.

Die Datenbank-Lösung

Eine Alternative zur Speicherung der Segmentierungsinformationen ist eine Datenbank, die es dem Plug-In ermöglicht, Informationen zu Segmenten auszulagern. Wird diese Datenbank zentral verwaltet und ist von den Entwicklerarbeitsplätzen zugreifbar, so kann über diese eine verteilte Benutzbarkeit der Migrationsunterstützung gewährleistet werden. Zudem bleibt der Quelltext des Ursprungssystems frei von zusätzlichen Kommentaren. Allerdings muss die Aktualisierung der Segmentierungsinformationen bei Änderungen im *CommSy* gewährleistet sein.

Die Datenbank-Lösung bringt ähnliche Probleme mit sich wie die Zwei-Dateien-Lösung. Werden alle Segmentierungsinformationen vollständig in der Datenbank abgelegt, so können bei dieser Lösung mit der Zeit Quelltext und darüber liegende Segmentierungsinformationen auseinander laufen. Der Vorteil der Datenbank liegt aber darin, dass keine zusätzliche Segmentdatei mehr benötigt wird, die von den Entwicklern mit der Quelltextbasis synchronisiert werden müsste. Die Aktualisierung der Segmentierungsinformationen wird

weiterhin durch das Plug-In gesichert. Das bringt eine weitere Einschränkung mit sich: das Plug-In benötigt stets Zugriff auf die Datenbank, um während der Weiterentwicklung eines in der Migration befindlichen Projekts die Segmentierungsinformationen aktuell halten zu können. Während bei der Zwei-Dateien-Lösung die Synchronisation mit der Quelltextbasis zu diskreten Zeitpunkten ausreicht, blockiert ein temporärer Verbindungsverlust des Plug-Ins zu der Datenbank die Weiterentwicklung. Solche Ausfallzeiten können abgefangen werden, indem Änderungen protokolliert werden und zu einem späteren Zeitpunkt in die Datenbank übernommen werden, wenn wieder eine Verbindung zur Datenbank besteht. Es wird dann eine Synchronisation der unterschiedlichen Stände am Arbeitsplatz und der zentralen Datenbank benötigt. In der *Single-Source-* sowie der Zwei-Dateien-Lösung kann diese Aufgabe durch das Versionierungssystem übernommen werden, wodurch die Datenbanklösung an dieser Stelle aufwändiger als die beiden bereits vorgestellten Alternativen erscheint.

Beurteilung der unterschiedlichen Alternativen

Die vorgestellten Lösungen (*Single-Source-*, Zwei-Dateien- und Datenbank-Lösung) haben jeweils eine Teilmenge der Probleme adressiert, die mit dem Einbringen zusätzlicher Informationen in vorhandene Projekte entstehen. Während die *Single-Source*-Lösung Fehlerquellen vermeidet, die durch mehrere Quellen für ein und dasselbe Dokument entstehen können – durch das Auseinanderlaufen der Stände in den unterschiedlichen Dateiquellen – wird durch die Zwei-Dateien- und die Datenbanklösung eine Kontamination des Quelltexts vermieden, indem alle Informationen vom Quelltext entkoppelt gespeichert werden. Diese Entkopplung erschwert aber wiederum die Handhabung der unterschiedlichen Informationsquellen, während das Einbringen aller Informationen in den Quelltext dessen Lesbarkeit vermindert.

Für das im Zuge dieser Diplomarbeit entwickelte Plug-In wurde die *Single-Source-*Lösung favorisiert und nach Alternativen gesucht, die Quelltextanteile, die die Migration betreffen, zumindest für die Entwickler des Ursprungssystems soweit auszublenden (oder zu minimieren), dass diese bei ihrer Arbeit nicht mehr bzw. möglichst wenig behindert werden. Eine solche Verbesserung ergab sich aus dem Vorschlag, anstatt der ursprünglich verwendeten Blockkommentare einzeilige Kommentare zu verwenden. Mit dieser Änderung konnten für jeden Segmentierungskommentar jeweils zwei Zeilen für den Start und das Ende des Blockkommentars eingespart werden.

Eine weitere Optimierung stellte das Zusammenfassen der beiden in ausnahmslos jedem Segment vorhandenen Informationen ID und Name unterhalb derselben Kommentarmarke (segment-begin, vgl. Anhang C) dar. Außerdem wurde das Namensfeld dazu verwendet, *obsolete* Segmente anhand eines reservierten Schlüsselworts zu kennzeichnen. Durch die Komprimierung dieser drei Attribute eines Segments konnten zwei weitere Kommentarzeilen pro Segment eingespart werden.

Um Segmente eindeutig einander zuordnen zu können, wird der Quelltextdateiname sowie die ID des verknüpften Segments benötigt. Durch Umbenennung einer Quelltextdatei, in der sich ein zugeordnetes Segment befindet, kann diese Verbindung verloren gehen, da der Name der Datei fest in den Segmentinformationen des Quelltextes abgelegt ist. Diese Information wird durch die Umbenennungsfunktion der *JDT*-Editoren nur bedingt aktualisiert; es ist bei einer Umbenennung möglich, im Quelltext davon betroffene Kommentare mit anpassen zu lassen. Dies macht den Umbenennungsschritt umständlicher, da die IDE sich diesen Schritt erst für alle Änderungen bestätigen lässt.

Denkbar wäre eine durch das Plug-In realisierte, automatisierte Aktualisierung von Segmenten im Falle einer Umbenennung. Das erfordert allerdings für alle betroffenen Ursprungssegmente eine vollständige Suche nach allen zugeordneten Zielsegmenten. Es besteht in der Migrationsunterstützung keine bidirektionale Verbindung zwischen den Segmenten der beiden Projekte, um die Informationsmenge, die in Ursprungssegmenten abgespeichert wird, möglichst gering zu halten. Das heißt, dass alle Informationen über eine Zuordnung in den Zielsegmenten abgelegt werden. Daher kann bei einer Umbenennung aus den Informationen am Ursprungssegment nicht direkt auf die davon betroffenen Zielsegmente geschlossen werden. Eine zusätzliche Unterstützung seitens des Plug-Ins erscheint daher in Hinblick auf den hohen Aufwand für die Realisierung und dem geringen Mehrwert nicht gerechtfertigt.

Zugriff auf die Segmentierungsinformationen

Der Zugriff auf die im Quelltext enthaltenen Segmente erfolgt im Plug-In über zwei gesonderte Schnittstellen, dem ISegmentUtility und dem ISegmentExtractor. Das ISegmentUtility stellt die Schnittstelle für eine Hilfsklasse bereit, die einige nützliche Methoden für die Arbeit in den Quelltextdokumenten zusammenfasst.

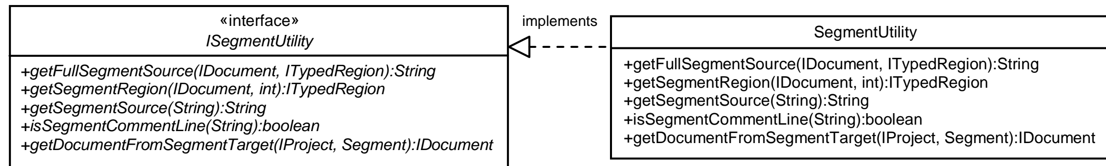


Abb. 4.5.: Die SegmentUtility-Schnittstelle

Die Hilfsklasse bietet die in Abb. 4.5. gezeigten Methoden an. So gibt es Methoden, um aus einer Quelltextdatei die in Segmente gefassten Abschnitte zu gewinnen oder für eine Zeilennummer zu ermitteln, ob sich die zugehörige Zeile innerhalb oder außerhalb eines Segments befindet. Ferner lässt sich für ein gegebenes Segment die Quelltextdatei des zugeordneten Ursprungssegments ermitteln.

Für das Auslesen von Segmentinformationen aus dem Quelltext wird die Schnittstelle ISegmentExtractor für jede der drei unterstützten Sprachen (*Java*, *JSP* und *PHP*) implementiert. In Abb. 4.6. ist das dazugehörige Klassendiagramm dargestellt: die abstrakte Klasse AbstractSegmentExtractor implementiert die allgemeine Funktionalität zum Auslesen von Segmenten aus dem Quelltext, während die konkreten Implementierungen die sprachspezifischen Kommentarbegrenzungen liefern, mit der die Informationen aus dem jeweiligen Quelltext extrahiert werden können.

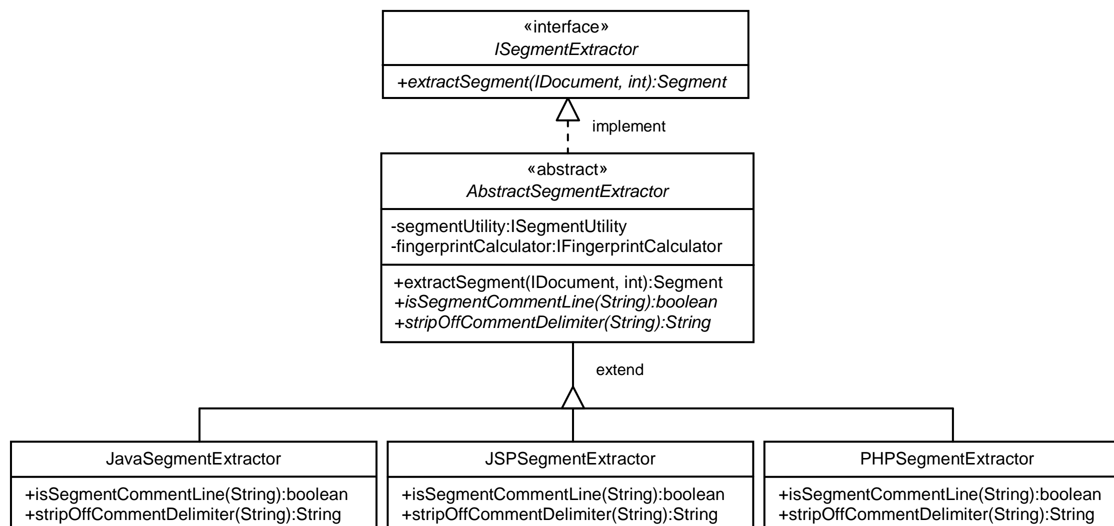


Abb. 4.6.: Die Schnittstelle für das Auslesen von Segmentierungsinformationen

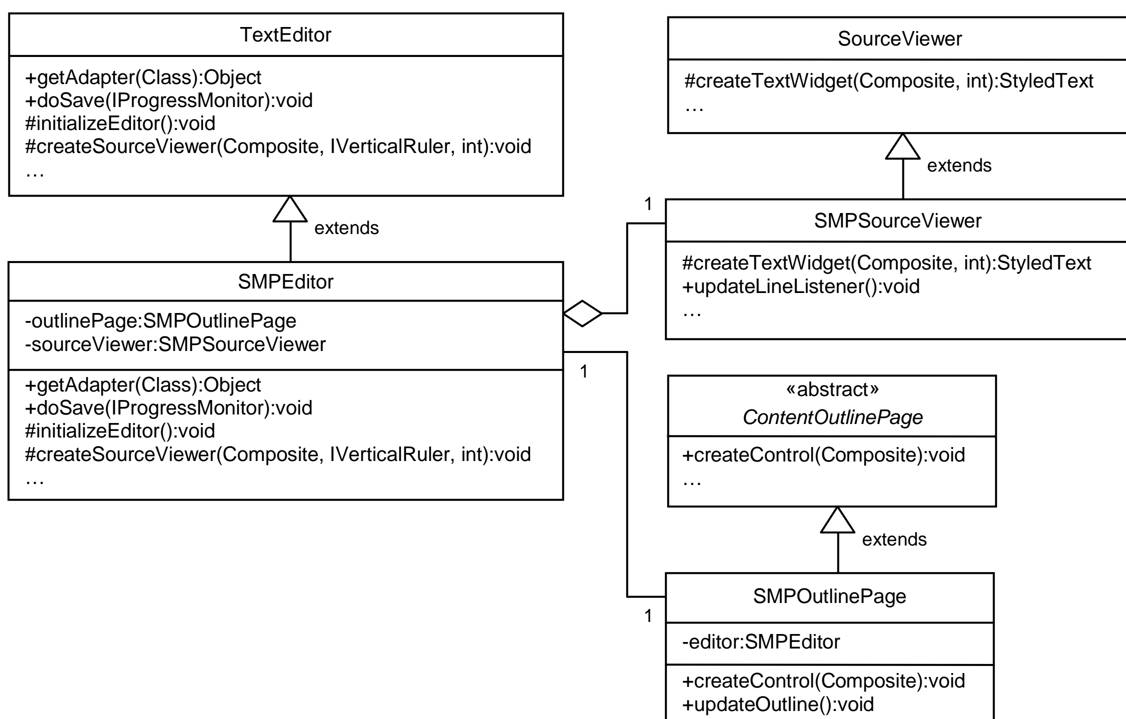
Der AbstractSegmentExtractor benutzt jeweils eine Instanz der oben beschriebenen Segmentierungshilfsklasse sowie eines Prüfsummenrechners, der die Schnittstelle IFingerprintCalculator erfüllt. Der Prüfsummenrechner wird zur Änderungsverfolgung eingesetzt und detailliert im nachfolgenden Kapitel beschrieben. Die benötigten Instanzen der Segmentierungshilfsklasse und des für die Änderungsverfolgung verwendeten Prüfsummenrechners werden dem JavaSegmentExtractor, JSPSegmentExtractor und PHPSegmentExtractor bei ihrer Erzeugung mitgegeben.

Dieses Vorgehen wird als *dependency injection* bezeichnet und löst die Abhängigkeit der Klassen zu konkreten Implementierungen der in ihnen verwendeten Schnittstellen auf (auch als *Inversion of Control*-Prinzip bezeichnet, vgl. [Fowler 2004]). Die in der Klasse

AbstractSegmentExtractor benötigten Instanzen von Hilfsklasse und Prüfsummenrechner werden außerhalb der Klasse erzeugt und bei der Instanzierung in diese hineingereicht, so dass sie selbst nicht mehr Kenntnis darüber benötigt, wie die Instanzen gebaut werden. Ändern sich die Details der Erzeugung der verwendeten Klassen, so bleibt die Implementierung der AbstractSegmentExtractor-Klasse (und von ihr abgeleiteter Klassen) davon unberührt. Dieses Prinzip findet sich noch an weiteren Stellen der Implementierung des Plug-Ins und dient der klaren Trennung der einzelnen Funktionseinheiten zum Erstellen, Auslesen, Ändern und Markieren von Segmenten im Quelltext.

Hervorheben von Segmenten im Editor

Das Hervorheben der Segmente erfolgt über das *syntax highlighting* der SMPEditor-Klasse. Der SMPEditor erweitert den TextEditor der *Eclipse*-Plattform. Abb. 4.7. zeigt im Klassendiagramm den Editor und die verwendeten Schnittstellen der IDE. Der Übersichtlichkeit halber ist die Darstellung der Schnittstellen auf einige wesentliche Methoden beschränkt.



Bsp. 4.7.: Klassendiagramm der SMPEditor-Klasse

Über die Methode getAdapter holt *Eclipse* vom Editor Instanzen für unterschiedliche Darstellungselemente ab, sofern der Editor diese zur Verfügung stellt. Dazu reicht *Eclipse* die geforderte Schnittstelle in die Methode, zum Beispiel IContentOutlinePage für eine *Outline*[1], und erwartet die Rückgabe einer Instanz, die diese Schnittstelle erfüllt. Die Einschubmethode doSave wird vom SMPEditor überschrieben, um beim Speichern auf Änderungen im Quelltext reagieren und die Outline und das *syntax highlighting* im Editor entsprechend aktualisieren zu können. Die dafür benötigten Instanzen von SMPOutlinePage und SMPSourceViewer werden in der Methode initializeEditor erzeugt. Um das im TextEditor standardmäßig verwendete *syntax highlighting* durch das eigene, im SMPSourceViewer implementierte zu ersetzen, überschreibt der SMPEditor die Methode createSourceViewer, die eine Instanz vom Typ SourceViewer zurückliefert.

[1] Als *Outline* wird in *Eclipse* eine zu einem Editor gehörende Ansicht bezeichnet, die zu bestimmten Dateitypen eine baumartige Strukturübersicht liefert, zum Beispiel bei *Java*-Dateien eine Auflistung der enthaltenen Methoden und Funktionen.

Der Editor enthält genau eine SMPSourceViewer-Instanz, um an dieser bei Änderungen über updateLineListener die Einfärbung des Editorhintergrunds gemäß der Segmentierung des Quelltexts und der Zustände der Segmente zu aktualisieren. Intern geschieht diese Einfärbung über ein spezielles Ereignis, ein Event, das durch den Editor beim Darstellen des Dokumentinhalts für jede angezeigte Quelltextzeile aufgerufen wird und die darzustellende Hintergrundfarbe erfragt.

Des Weiteren kennt der Editor eine Instanz der SMPOutlinePage, über die mit updateOutline die in der Outline dargestellten Segmentierungsinformationen synchronisiert werden. Diese Instanz enthält den Editor, zu der sie gehört und über den das in der Outline darzustellende Quelltextdokument zugegriffen werden kann.

Die Einbindung des Editors erfolgt über den Einstiegspunkt org.eclipse.ui.editors. Die Editor-Schnittstelle, die diesen Einstiegspunkt implementiert, sorgt dafür, dass im Editor die benötigten Werkzeuge an die jeweilige Editorinstanz gebunden werden. Dazu gehören unter anderem die oben genannte Outline, das *syntax highlighting* sowie die Dokumentstruktur der im Editor bearbeiteten Dokumente. Bsp. 4.8. zeigt die für die Registrierung von Editoren benötigten Informationen. Erwähnenswert sind dabei besonders die Einschränkung auf bestimmte Dateitypen (in diesem Beispiel *Java*-Quelltextdateien) sowie die Zuordnung des Editors zu einer vorgefertigten Klasse von Editoren. Die *contributor class* ermöglicht es dem Entwickler, den Editor mit einem grundlegenden Verhalten zu versehen, ohne die gängige Funktionalität komplett selber nachprogrammieren zu müssen. Zu diesem Verhalten gehört zum Beispiel das Kopieren, Ausschneiden und Einfügen von Texten im Editor.

```
<extension point="org.eclipse.ui.editors">
  <editor icon="icons/logo.gif"
           class="org.eclipse.contribution.smp.editors.SMPEditor"
           default="false"
          contributorClass=
             "org.eclipse.ui.texteditor.BasicTextEditorActionContributor"
          name="%Label.JavaEditorLabel"
          id="org.eclipse.contribution.smp.editors.smpjavaeditor"
          extensions="java" />
  ...
</extension>
```

Bsp. 4.8.: Registrierung eines Editors (im Plug-In-Manifest)

Explizite vs. implizite Segmentierungsinformationen

Zu Beginn des Projekts wurde im Prototyp die Segmentierungsinformation gänzlich im Kopf eines Segments in den zugehörigen Kommentaren abgelegt. Das führte unter anderem zu Problemen, wie sie schon zu Beginn des Kapitels beschrieben wurden (Aufblähen des Quelltexts, Einschränken der Lesbarkeit). Im Laufe der Entwicklung wurden aus diesem Grund einige Informationen, die explizit im Segment im Quelltext abgelegt wurden, wieder daraus entfernt und an andere Stellen verlagert:

Projekt des Ursprungssegments

Dieser Eintrag erlaubte eine bequeme Navigation aus dem Quelltext heraus zum Ursprungsprojekt (und der darin enthaltenen Quelltextdatei). Das Projekt wurde redundant in allen Zielsegmenten gespeichert. Die lokale Umbenennung von Projekten hätte eine Änderung aller betroffenen Zielsegmente nach sich gezogen. Diese Information wanderte in die zentrale Migrationsprojektdatei, um die mögliche Umbenennung eines Projekts zentral auf einen Ort im Migrationsprojekt zu beschränken. Die Information, in welchem Projekt sich eine zugeordnete Quelltextdatei befindet, wird implizit aus der Zugehörigkeit der Zieldatei zu einem Projekt geschlossen. Dieses ist wiederum durch das Migrationsprojekt eindeutig einem Ursprungsprojekt zugeordnet.

Prüfsumme eines Segments
In einem Segment wurde eine Prüfsumme eingetragen, die zur Erkennung von Änderungen herangezogen werden konnte. Der Wert wurde zusammen mit dem Segment ausgelesen und erlaubte es bei zugeordneten Segmenten, Änderungen im Ursprungssegment zu identifizieren. Ursprünglich wurde diese Information redundant in allen Segmenten gespeichert, um einen effizienteren Zugriff zu erlauben, da für die Visualisierung von Änderungen nur die Segmentköpfe ausgelesen werden mussten. Dies erforderte allerdings eine Aktualisierung der gespeicherten Prüfsumme bei jeder Änderung innerhalb des dazu gehörenden Segments. Die ursprüngliche Implementierung sah vor, die Prüfsumme bei allen ändernden Zugriffen auf die Quelltexte erneut zu berechnen. Das hätte zusätzlich die Entwicklung eines *Prozessors* notwendig gemacht. Dieser Prozessor wäre dafür zuständig, alle Quelltextdateien automatisiert zu durchlaufen und inkonsistente Prüfsummen zu korrigieren. Da der Aufbau der Segmente im Quelltext – mit einleitendem und abschließendem Quelltextkommentar – es erfordert, beim Extrahieren der Segmentierungsinformationen ein Dokument vollständig zu durchlaufen, wurde die Berechnung der Prüfsumme in den schon beschriebenen AbstractSegmentExtractor (=die Klasse, die die Segmentierung aus Quelltextdateien ausliest) verschoben. Die Prüfsumme ergibt sich implizit aus dem im Segment eingeschlossenen Quelltext und wird nun *on-the-fly* beim Durchlaufen des Quelltexts mitberechnet. Durch die Berechnung der Prüfsummen beim Auslesen der Segmente aus einer Datei entfällt die redundante Speicherung der Prüfsummen.

Name eines Segments
In der ersten Fassung des Plug-Ins wurde eine Zuordnung von Segmenten zueinander ausschließlich über die ID ermöglicht. Im Zuordnungsdialog konnte ein Ursprungssegment nur anhand seiner ID identifiziert werden, da nur diese im Dialog angezeigt wurde. Das bedeutete für den Benutzer, dass dieser selbst den Überblick behalten musste, welche ID die von ihm erstellten Segmente in einer Quelltextdatei haben. Deshalb wurde zusätzlich das Setzen und Speichern eines Namens implementiert. Dieser Name wird für die Identifikation und Unterscheidung der Segmente nicht benötigt, erleichtert aber dem Benutzer die Zuordnungsarbeit. Um diese Arbeit weiter zu vereinfachen, wurde der Segmentierung im Quelltext eine Funktion hinzugefügt, die speziell beim Erstellen von Segmenten in *PHP*-Quelltexten versucht, einen Namensvorschlag aus einer im Segment eingeschlossenen *PHP*-Funktionssignatur zu generieren.

4.3.3. Eindeutigkeit und Änderungsverfolgung

Die Eindeutigkeit bzw. eindeutige Identifizierbarkeit von im Quelltext definierten Segmenten ist für eine Zuordnung von Zielsegment zu Ursprungssegment erforderlich, die über Änderungen am Quelltext hinaus erhalten bleibt. Das Plug-In benötigt daher eine Möglichkeit, Segmente mit einer Form von Identifikationsnummer zu versehen, die das Segment im Bereich, in dem es mit anderen Segmenten sichtbar ist, eindeutig identifizierbar machen. Dafür wurden zwei Alternativen betrachtet:

a) ein vom Benutzer gewählter Bezeichner bzw. Name für ein Segment.
b) eine vom Plug-In automatisch beim Erstellen eines Segments vergebene, technische Identifikationsnummer.

Die Umsetzung von Variante a) bedingt die Erweiterung des Plug-Ins um Mechanismen, um die Eindeutigkeit trotz freier Namenswahl seitens des Benutzers zu gewährleisten. Denkbar wäre an dieser Stelle eine Warnmeldung, wenn der Benutzer einen Namen wählt, der schon für ein anderes Segment vergeben wurde. Die Nachteile dieser Variante ergeben sich aus der Annahme, dass es bei sehr großen Quelltexten für den Benutzer sehr schwierig werden kann, einen Überblick über die schon verwendeten Namen zu behalten. Die automatische Erstellung einer ID gemäß Variante b) erscheint für diesen Verwendungszweck sinnvoller, da der Benutzer von der Aufgabe befreit wird, selbst für die eindeutige Identifizierbarkeit von Segmenten zu sorgen.

Die Berechnung einer technischen ID wurde aus diesem Grund für die Identifizierung von Segmenten umgesetzt. Trotzdem wurde die gleichzeitige Auszeichnung von Segmenten mit einem Namen in die Segmentierungsinformationen übernommen, um Segmente für den Benutzer leichter identifizierbar zu machen. Der Name eines Segments wird vom Plug-In nicht zur Identifizierung benutzt. Die Umsetzung im Plug-In geschieht über die Berechnung einer fünfstelligen Zufallszahl, was im Sichtbarkeitsbereich einer einzelnen Quelltextdatei als ausreichend betrachtet wird, um für alle enthaltenen Segmente eine eindeutige ID zu erzeugen. Segmente in unterschiedlichen Quelltextdateien sind durch die Zuordnung zur jeweiligen Datei voneinander abgegrenzt, so dass das Vorhandensein derselben ID in unterschiedlichen Quelltextdateien desselben Projekts unproblematisch ist.

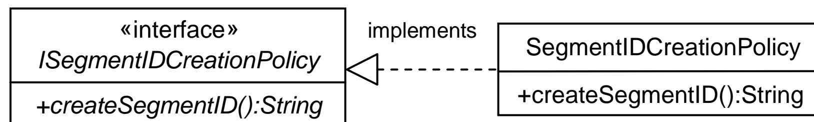


Abb. 4.9.: Die Schnittstelle der Klasse SegmentIDCreationPolicy

Die in Abb. 4.9. gezeigte SegmentIDCreationPolicy dient der Erzeugung von Identifikationsnummern für die eindeutige Identifizierung von Segmenten innerhalb einer Quelltextdatei. Eine solche Berechnungsvorschrift wird bei der Erzeugung neuer Segmente angewendet. Die in der vorliegenden Plug-In-Version verwendete Implementierung erzeugt eine fünfstellige Zufallszahl, was sich im bisherigen Gebrauch als praktikabel herausgestellt hat. Die Zahl der bisher in einzelnen Quelltextdateien erstellten Segmente ist weit geringer, so dass die Wahrscheinlichkeit einer zufälligen Doppelung einer ID innerhalb einer Quelltextdatei als ausreichend gering anzusehen ist.

Um Änderungen am Ursprungsprojekt verfolgen zu können, bietet sich in erster Linie das in den Projekten verwendete Versionierungssystem der zentralen Quelltextbasis an. Allerdings müsste dabei für jede einzelne geänderte Quelltextdatei bei der Synchronisation geprüft werden, ob sich darin Änderungen befinden, die schon migrierte Systemteile betreffen. Danach wäre für jede einzelne Änderung zusätzlich zu prüfen, ob für eine bereits vorgenommene Implementierung im Zielsystem eine Korrektur oder Erweiterung erforderlich ist. Schon bei wenigen Änderungen im Ursprungssystem wird diese Aufgabe aufwändig. Bei fortlaufenden Änderungen mit den dadurch notwendig werdenden Prüfungen würde dieses Vorgehen bei der Migration einen beachtlichen Zeit- und Arbeitsfaktor ausmachen.

Eine Lösung für dieses Problem ist es, miteinander verbundene Segmente um Informationen zu erweitern, die die in Kapitel 3.4. beschriebene Änderungsverfolgung im Plug-In ermöglichen. Mit Hilfe dieser Informationen können Änderungen am Ursprungssystem automatisiert erfasst und dem Benutzer zur Ansicht gegeben werden. Eine Möglichkeit stellt dafür die Berechnung einer Prüfsumme, eines so genannten *hash codes* aus dem im Segment eingefassten Quelltext dar, der zum späteren Vergleich abgespeichert werden kann.

Die Grundlage dafür bilden die Ausführungen über die Berechnung von Hash-Werten aus dem *Handbook of Applied Cryptography* ([Menezes 1997], S. 322ff), dem einige der im Folgenden angeführten (engl.) Begriffe entnommen sind. Die Definition einer solchen, dafür benutzten *hash function* stammt aus [Menezes 1997], S. 322:

> „*A* hash function *(in the unrestricted sense) is a function h which has, as a minimum, the following two properties:*
> 1. compression – *h maps an input x of arbitrary finite bitlength, to an output $h(x)$ of fixed bitlength n.*
> 2. ease of computation – *given h and an input x, $h(x)$ is easy to compute.*“

Ein Hash-Wert (Streuwert) ist also die Ausgabe einer Hash-Funktion (Streufunktion), die eine endliche Eingabe, wie z.B. in diesem Fall eine Quelltextzeichenkette, durch eine einfache Berechnungsvorschrift auf einen Wert, z.B. eine natürliche Zahl fester Länge, abbildet.

Im Kontext der Änderungsverfolgung sind beide Punkte von Bedeutung. Die Errechnung eines Hash-Werts fester Länge ist für die Speicherung in den Segmentierungsinformationen wichtig, vor allem, um den Quelltext nicht durch beliebig lange Einträge zu belasten. Weiterhin ist eine effiziente Art der Berechnung relevant für die Performanz des Plug-Ins, da der Hash-Wert oft neu berechnet wird. Des Weiteren muss die für die Änderungsverfolgung benötigte Prüfsumme zwei grundlegende Bedingungen (zusätzlich zu denen aus der Definition des Hash-Werts) erfüllen:

a) die Möglichkeit, Änderungen am Quelltext erkennen zu können (*modification detection code*, *MDC*) sowie
b) eine ausreichend gleichmäßige Verteilung der erzeugten Hash-Werte zu gewährleisten, um Änderungen am Quelltext nicht durch zufällige Übereinstimmung der zugehörigen Hash-Werte zu maskieren (*collision resistant hash function*, *CRHF*).

Die im Prototypen benutzte Version der Berechnungsfunktion wurde zu Testzwecken stark vereinfacht und bildete über die einzelnen Zeichen des Segmentquelltexts das Produkt der numerischen (Unicode-)Zeichenwerte. Die Bedingung des Aufdeckens von Änderungen konnte durch diese Funktion nicht erfüllt werden. Wurde im Quelltext ein Teil innerhalb eines Segments verschoben, wurde diese Änderung durch die ungenügende Empfindlichkeit der benutzen Funktion maskiert, da das berechnete Produkt gleich blieb (ungenügende *2nd-preimage resistance*). Um eine solche Änderung aufzudecken, kann der Funktion noch eine Positionsempfindlichkeit hinzugefügt werden, zum Beispiel eine Wichtung der einzelnen Zeichen des Quelltexts anhand ihrer Position relativ zum Anfang des Segments.

Um den Bedingungen für die Erkennung von Änderungen im Quelltext gerecht zu werden, wurde daher ein Algorithmus gesucht, der eine ausreichende Sicherheit mitbringt, dass unterschiedliche Eingaben zu unterschiedlichen Prüfsummen führen. Der zugehörige Algorithmus sollte zudem frei verfügbar und theoretisch gut verstanden sein. Als Beispiele wären *MD5* und *SHA* zu nennen (vgl. [Menezes 1997], S. 343ff). Obwohl diese Verfahren eher zur Verschlüsselung von zum Beispiel E-Mails eingesetzt werden, geht es bei der Migrationsunterstützung ausschließlich um das durch diese Algorithmen realisierte *hashing*. Da es im Fall der Änderungsverfolgung nicht um die derzeit maximal mögliche Sicherheit der berechneten Hash-Werte geht (im Gegensatz zur Berechnung von digitalen Signaturen zum Beispiel, wo diese Algorithmen zum Einsatz kommen), sondern nur um eine möglichst gleichmäßige Verteilung, wurde der MD5-Algorithmus (*message digest* 5 der Firma *RSA Data Security*) ausgewählt. Die Wahrscheinlichkeit, dass zwei beliebige, unterschiedliche Eingaben bei dieser Methode denselben Hash-Wert erzeugen, wird mit $1/2^{128}$ als ausreichend gering angesehen, so dass MD5 als ein für diese Zwecke praktikabler Weg der Prüfsummen-Berechnung angesehen wird.

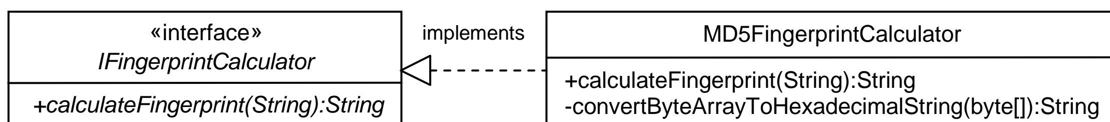


Abb. 4.10.: Die Schnittstelle der Klasse MD5FingerprintCalculator

Die Schnittstelle zur Berechnung von Prüfsummen aus einem gegebenen Quelltextabschnitt findet sich im IFingerprintCalculator (Abb. 4.10.), die im Plug-In durch den in der Abbildung dargestellten MD5FingerprintCalculator implementiert wird. Intern wandelt diese Implementierung die aus einem Quelltextabschnitt errechnete MD5-Zeichenkette in eine Hexadezimaldarstellung um. Damit soll verhindert werden, dass eine Prüfsumme, die sich aus dem Quelltext eines Segments ergibt, zufällig Zeichenfolgen enthält, die für den jeweiligen Quelltext von Bedeutung sein könnten. Das könnten zum Beispiel Zeichenfolgen sein, die einen Kommentarblock beenden. Eine solche Zeichenfolge kann einen Fehler im Quelltext verursachen, wenn das Abspeichern der Prüfsumme im Segmentkommentar die zugehörige

Kommentarzeile vorzeitig terminiert und die nachfolgenden Zeichen (der Rest der Prüfsumme) fälschlicherweise als Programmtext interpretiert werden.

4.3.4. Umsetzung des Benutzungsmodells

Die Benutzungsschnittstelle umfasst alle Oberflächenelemente, die durch das *Software Migration Plug-In* zur Verfügung gestellt werden. Der Umfang der Funktionalität, die dem Benutzer für die einzelnen Arbeitsaufgaben zur Verfügung gestellt wird, ergibt sich aus dem in Kapitel 4.2. beschriebenen Benutzungsmodell bzw. den einzelnen Szenarios. Um dem Benutzer eine intuitiv zugängliche Benutzungsschnittstelle zu bieten, wurde in den Szenarios für jede Arbeitsaufgabe ermittelt, wo in der IDE die zugehörige Funktionalität vom Benutzer erwartet wird. Je nach Aufgabe ergaben sich dabei eine oder mehrere Alternativen, von denen jeweils eine umgesetzt wurde.

Für alle durch die Migrationsunterstützung angebotenen Oberflächenelemente wurde deren Einbindung in der IDE diskutiert. Als Orientierung dafür wurde nach einer Grundlage gesucht, mit deren Hilfe Entscheidungen hinsichtlich der Orte getroffen werden können, an denen Funktionalität in die IDE integriert wird. Oppermann beschreibt in seiner Arbeit über Software-Ergonomie die grundlegenden Begriffe, darunter den Begriff der *Erwartungskonformität* ([Oppermann 1992], S. 340ff.). Oppermann definiert diesen Begriff wie folgt:

> *„Ein Dialog ist erwartungskonform, wenn er den Erwartungen der Benutzer entspricht, die sie aus Erfahrungen mit bisherigen Arbeitsabläufen oder aus der Benutzerschulung mitbringen sowie den Erfahrungen, die sich während der Benutzung des Dialogsystems und im Umgang mit dem Benutzerhandbuch bilden“ (vgl. [Oppermann 1992], S. 341)*

Dieses Prinzip wurde nun herangezogen, um einen Anhaltspunkt für die Entscheidungen zu finden, nach der die Benutzungsschnittstelle des Plug-Ins umgesetzt werden kann. Die an der Migration beteiligten Entwickler sind die Arbeit in *Eclipse* gewohnt. Bei der Einbindung neuer Funktionalität ist also zu beachten, dass bestimmte Arten von Interaktionen (zum Beispiel Quelltextoperationen oder das Erstellen von Ressourcen) vom *Eclipse*-Entwickler an bestimmten Orten in der Entwicklungsumgebung erwartet werden. Dementsprechend wurde die Funktionalität des Plug-Ins jeweils an den Stellen der Oberfläche eingebunden, an denen sich gleichartige Funktionen der Standardinstallation finden.

Als Beispiel sei die Einrichtung eines Migrationsprojekts genannt. Das zugehörige Szenario beschreibt für diesen Arbeitsschritt das Erstellen des Projekts als neue Ressource. Für das Erstellen von Ressourcen gibt es in *Eclipse* einen Eintrag im Dateimenü, über den eine Auswahl an erzeugbaren Ressourcentypen aufgerufen werden kann. Dementsprechend wurde das Migrationsprojekt als Ressource für diese Auswahl registriert und kann über das Dateimenü erzeugt werden.

Die aus dem Oberflächenverhalten von *Eclipse* abgeleitete Erwartungshaltung des Benutzers ist nicht immer eindeutig bzw. lässt sich nicht immer mit der tatsächlichen Erwartung zur Deckung bringen. Sie stellt eine Annahme dar, die sich aus der Betrachtung der Funktionsweise schon vorhandener IDE-Anwendungsteile herleitet. Diese Annahme dient dabei ausschließlich als Orientierungshilfe. Aus diesem Grund ergeben sich bei der Benutzung stellenweise Diskrepanzen zwischen der Erwartungshaltung der Benutzer und der Umsetzung durch das *Software Migration Plug-In*. In Kapitel 4.3.6. wird dieses Problem beispielhaft an einer Plug-In-Funktion beschrieben, die kontextsensitives Verhalten bezüglich Mauszeigerposition und Kontextfunktionalität *Eclipse*-konform umgesetzt hat. Dass die Benutzer an dieser Stelle ein leicht anderes Verhalten erwarteten – wie sich während der Entwicklung herausstellte - lässt vermuten, dass die Annahme falsch oder zumindest nicht ausreichend ist, die derzeitige *Eclipse*-Oberfläche allein genüge als Grundlage hinsichtlich der Erwartungskonformität.

Die Schwierigkeit in diesem speziellen Punkt der Entwicklung lag darin, zwischen der Konformität mit dem *Eclipse*-Oberflächenverhalten und der Erwartungshaltung der Benutzer abzuwägen. Wie sich herausstellte, war selbst erfahrenen *Eclipse*-Nutzern das spezielle

Verhalten der Entwicklungsumgebung für das genannte Beispiel nicht bewusst, weshalb das Verhalten des *Software Migration Plug-Ins* an besagter Stelle zuerst als fehlerhaft empfunden wurde.

Überblick über die Plug-In-Funktionen

Aus den Szenarios wurde ein grundlegender Satz an Funktionen ermittelt, die die jeweiligen Arbeitsschritte unterstützen sollen. Die Arbeitsschritte wurden in Kapitel 4.2. nacheinander beschrieben. Die jeweils zugeordneten Szenarios enthalten Hinweise, wo die entsprechende Funktionalität in die Oberfläche eingebunden werden soll. Die im Plug-In jeweils umgesetzte Lösung für ein gegebenes Szenario wurde dahingehend untersucht, ob sie im Kontext der verwendeten *Eclipse*-Entwicklungsumgebung den Benutzer ausreichend unterstützt.

Tab. 4.11. bietet einen Überblick über die Funktionen, die das Plug-In dem Benutzer an der *Eclipse*-Oberfläche anbietet. Die in der ersten Spalte genannten *Aktionen* umfassen aktive Benutzeraktionen, während reine Informationselemente als *Ansicht* bezeichnet werden. Die passiven Ansichten liefern dem Benutzer zusätzliche Informationen, müssen allerdings nicht aktiv aufgerufen werden (zum Beispiel die Editor-Outline). Die Ansichten sind in der ersten Spalte kursiv dargestellt.

Aktion/Ansicht	Ort	Beschreibung
Projekt erstellen	Neu-Menü	Öffnet einen *Wizard*[1]-Dialog zur Erstellung eines Migrationsprojekts, in dem ein Ursprungsprojekt einem Zielprojekt zugeordnet wird.
Java-Datei segmentieren	Kontextmenü im *Navigator*[2]	Segmentiert *Java*-Quelltext, indem Funktionen und Methoden automatisch in Segmente eingeschlossen werden. Arbeitet auf einer vollständigen Quelltextdatei.
Segment erstellen	Kontextmenü im Editor	Erstellt aus einer Selektion im Editor ein Segment. Befindet sich in der Selektion eine Funktion, so wird dem Segment der Name dieser Funktion gegeben (nur in *PHP*-Dateien). Wird ein Segment innerhalb eines vorhandenen Segments erstellt, wird das vorhandene Segment in drei neue Segmente aufgeteilt.
Segment als obsolet markieren	Kontextmenü im Editor	Markiert ein Segment im Ursprungsprojekt als obsolet und somit als nicht zu migrieren.
Segment zuordnen	Kontextmenü im Editor	Ordnet einem Segment aus dem Zielprojekt ein Segment aus dem Ursprungsprojekt zu. Funktioniert nur in den entsprechenden Projekten eines Migrations-projekts.
Segment validieren	Kontextmenü im Editor	Setzt den Zustand eines Zielsegments zurück, dessen Ursprungssegment nachträglich verändert wurde. Durch das Validieren wird die Prüfsumme am

[1] Ein *Wizard* (auch *Assistent* genannt) ist eine besondere Form von Eingabedialog in *Eclipse* und unterstützt den Benutzer bei der Eingabe von Informationen, die z.B. für das Erstellen neuer Dokumente oder Projekte benötigt werden.

[2] Der *Navigator* stellt die in *Eclipse* zugreifbaren Projekte und die in ihnen enthaltenen Ressourcen in einer baumartigen Übersicht dar.

Aktion/Ansicht	Ort	Beschreibung
		Zielsegment auf das veränderte Ursprungssegment angepasst.
Zum Segment springen	Kontextmenü im Editor	Ist einem Segment im Zielprojekt ein Segment im Ursprungsprojekt zugeordnet, kann der Benutzer direkt vom Ziel- ins Ursprungssegment springen.
Migration analysieren	Kontextmenü im Navigator	Mithilfe der Segmentierung und Zuordnung kann der Benutzer den Migrationsfortschritt automatisiert analysieren. Dabei werden im Ursprungs- und Zielprojekt die Segmente sowie deren Zuordnung und Zustand ermittelt. Daraus wird eine statistische Auswertung generiert. Diese Funktion kann auf Projekten aufgerufen werden, die Teil eines Migrationsprojekts sind.
Segmentinformation	Outline im Editor	Im Plug-In-eigenen Editor werden in der Outline alle im Dokument enthaltenen Segmente und die dazugehörigen Informationen angezeigt.
Segmentstatus	Im Editor	Segmentierter Quelltext wird im Editor farblich hinterlegt. Dabei werden verschiedene Zustände der Segmente farblich unterschiedlich dargestellt.

Tab. 4.11.: Aktionen/Ansichten und ihre Lokalisierung in *Eclipse*

An dieser Stelle sei angemerkt, dass diese Übersicht nur die für die Benutzungsschnittstelle relevante Funktionalität enthält. Die für die Zuordnung von Segmenten zueinander erforderliche Eindeutigkeit einzelner Segmente sowie die für die Änderungsverfolgung benötigte Prüfsummenberechnung sind im vorangegangenen Kapitel behandelt worden.

4.3.5. Erstellen von Migrationsprojekten

Zu Beginn der Arbeit an einer Migration wird ein Migrationsprojekt erstellt. In einem solchen Migrationsprojekt werden zwei Projekte miteinander verbunden. Auf der einen Seite das Ursprungsprojekt, dessen Segmente die Ursprungsfunktionalität enthalten, die in das Zielprojekt auf der anderen Seite migriert werden. Zudem dient diese Zuordnung als Abgrenzung zu anderen Migrationsprojekten.

Projektzuordnung

Für das Erstellen von Ressourcen wie Projekte oder Quelltextdateien steht in *Eclipse* eine Vielzahl von Wizards zur Verfügung. Um ein neues Migrationsprojekt zu erstellen, durchläuft der Benutzer dementsprechend einen dafür vom Plug-In registrierten Wizard. Dieser Dialog fragt nacheinander Ursprungs- und Zielprojekt der Migration ab und versieht dann beide Projekte mit einer entsprechenden Projektdatei. Das Ursprungsprojekt wird mit einer `migration-source.xml`-Datei markiert, das Zielprojekt mit einer `migration-target.xml`-

Datei. Beide Dateien enthalten jeweils die Information über die Zuordnung der Projekte zueinander. Die Auswahl an Projekten bezieht der Wizard aus dem *Workspace*[1] von *Eclipse*.

Damit sich der Benutzer bei der Projektauswahl zurechtfindet, werden alle Projekte aus dem Workspace von *Eclipse* in einer flachen Baumstruktur angezeigt, die den Baumstrukturen im Navigator oder im *Package Explorer*[2] von *Eclipse* entspricht. Damit wird auf zwei aufeinander folgenden Dialogseiten jeweils ein Projekt ausgewählt und beim Beenden des Wizards die Zuordnung vorgenommen. Die Auswahl der Projekte wird auf den Workspace von *Eclipse* beschränkt, da das Plug-In aus seinem Kontext heraus nur auf die im Workspace enthaltenen, lokalen Projekte zugreifen kann.

Die Verknüpfung der an einem Migrationsprojekt beteiligten Projekte erfolgt ausschließlich über den Projektnamen. Es ist in *Eclipse* allerdings möglich, einem Projekt einen lokalen Namen zu geben, der sich von dem im *Repository* eingetragenen Projektnamen unterscheidet. In der ersten Version des *Software Migration Plug-Ins* wurde die Information über das zugeordnete Ursprungsprojekt noch im Quelltext in den Segmenten gespeichert. Diese Informationen wurden zusammen mit dem Quelltext in die zentrale Quelltextbasis eingespielt und waren somit für alle Entwickler zentral festgeschrieben. Durch eine lokale Änderung des Projektnamens konnte damit die Zuordnung eines Quelltextsegments zum Ursprungssegment verloren gehen, wenn sich an einem Entwicklerarbeitsplatz die Projektnamen voneinander unterschieden. Daher wurde eine Lösung gesucht, die es erlaubte, die Projekte innerhalb eines Migrationsprojekts umbenennen zu können, um die Zuordnung wiederherzustellen. Für die erste Fassung des Plug-Ins hätte das bedeutet, dass im Falle einer Umbenennung alle Segmente der beteiligten Projekte überprüft und die in ihnen gespeicherte Projektinformation hätte angepasst werden müssen.

Als Lösung für dieses Problem wurde die Projektinformation auf eine zentrale Instanz beschränkt. Die Informationen über Ursprungs- und Zielsystem sind in den Migrationsprojektdateien der jeweiligen Projekte gespeichert. Die eindeutige Zuordnung von Quelltextdateien zu Projekten erfolgt nun implizit über deren Platz in den Projektverzeichnissen im Workspace von *Eclipse*. Wird für die Ermittlung des Zustands eines Segments oder für die Navigation zwischen den Segmenten das jeweilige Ursprungs- oder Zielprojekt benötigt, so kann diese Information aus der zugehörigen Migrationsprojektdatei des jeweiligen Projekts ausgelesen werden. Die Projektinformation ist in den Segmentkommentaren somit redundant und wurde daher wieder aus diesen Kommentaren entfernt.

Wird der Name eines Projekts lokal angepasst, geht dadurch die Verbindung zwischen den Segmenten verloren. Diese Verbindung kann nun durch eine einfache Aktualisierung des Migrationsprojekts wiederhergestellt werden. Eine Aktualisierung aller Segmente ist bei dieser Umsetzung nicht mehr notwendig.

Wird im Verlauf der Migration eines der Projekte lokal umbenannt, so kann der Benutzer über den *Wizard* die neue Zuordnung wiederherstellen und die Verbindung zwischen Ursprungs- und Zielprojekt erneuern.

[1] Als *Workspace* wird der Arbeitsbereich von *Eclipse* bezeichnet, in der sich die Projekte befinden und alle dazugehörigen Ressourcen abgelegt sind.

[2] Der *Package Explorer* stellt die in *Eclipse* zugreifbaren Projekte und die in ihnen enthaltenen Quelltextdateien, Klassen und Bibliotheken in einer baumartigen Übersicht dar.

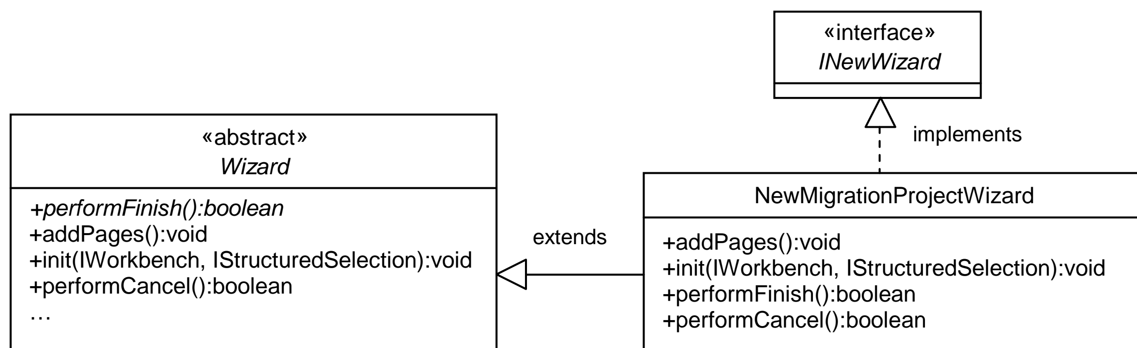


Abb.. 4.12.: Das Klassendiagramm für den NewMigrationProjectWizard

Um einen Wizard-Dialog zum Erstellen eines Migrationsprojekts in die IDE einzubinden, implementiert das *Plug-In* den Einstiegspunkt org.eclipse.ui.newWizards und bindet dort eine Implementierung der Schnittstelle INewWizard ein. Die Wizard-Implementierung hat dabei die Aufgabe, die verschiedenen Konfigurationsschritte in Form von einzelnen Eingabeformularen zusammenzuführen und zu steuern.

Abb. 4.12. zeigt das Klassendiagramm der NewMigrationProjectWizard-Klasse des Plug-Ins. Die Schnittstelle INewWizard selbst ist leer, erweitert allerdings IWizard. Die abstrakte Wizard-Klasse implementiert diese Schnittstelle bis auf die Methode performFinish, die mindestens vom NewMigrationProjectWizard implementiert werden muss. Die Methode kann das Beenden des Dialogs gestatten oder im Falle fehlender oder falscher Eingaben verweigern. Analog ermöglicht die Methode performCancel, einen vorzeitigen Abbruch des Dialogs zu steuern. In performFinish wird mit den ausgewählten Projekten ein Migrationsprojekt erstellt.

Um dem Dialog die einzelnen Eingabeformulare hinzuzufügen, überschreibt der NewMigrationProjectWizard die Methode addPages und erzeugt dort die Formular-instanzen. Die Methode init kann dazu verwendet werden, aus der Arbeitsumgebung von *Eclipse* Ressourcen (über die IWorkbench-Schnittstelle) zuzugreifen oder den Dialog anhand einer Selektion von Ressourcen (über die IStructuredSelection-Schnittstelle) zum Beispiel mit Standardvorgaben zu befüllen.

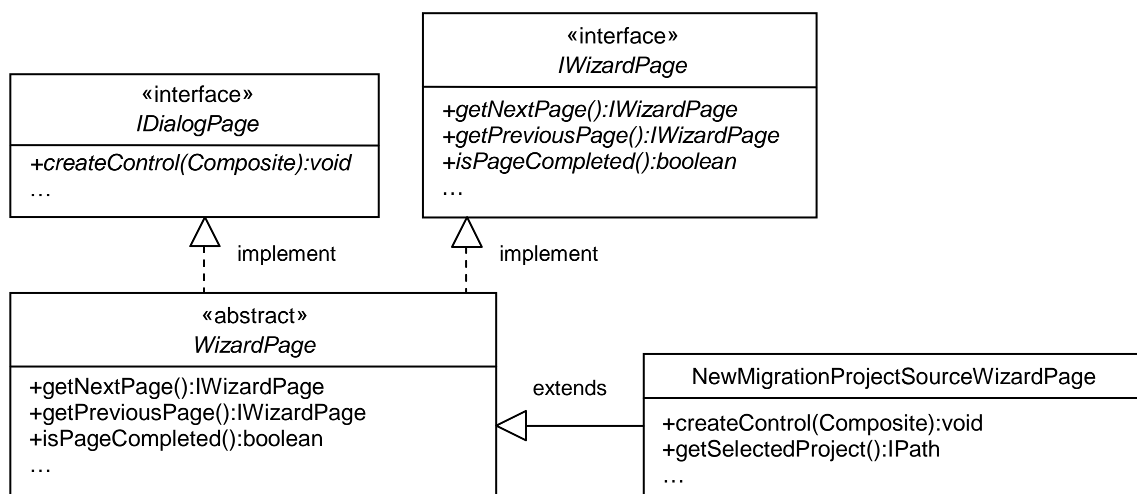


Abb.. 4.13.: Das Klassendiagramm für den NewMigrationProjectSourceWizardPage

Die einzelnen Formulare erweitern die abstrakte Klasse WizardPage. Abb. 4.13. zeigt das Klassendiagramm für die Klasse NewMigrationProjectSourceWizardPage, die den

Benutzer nach dem Ursprungsprojekt fragt. Die Verknüpfung von aufeinanderfolgenden Formularseiten geschieht in der Implementierung der abstrakten WizardPage-Klasse. Die Schnittstelle IWizardPage wird dabei vollständig implementiert. Zwischen den einzelnen Seiten bzw. Schritten kann der Benutzer blättern. Das Blättern kann eingeschränkt werden, wenn ein Konfigurationsschritt Informationen benötigt, die bei einem vorhergehenden angegeben werden müssen. Dazu kann die konkrete Formularseitenimplementierung die Methode isPageCompleted überschreiben. Hat der Assistent alle benötigten Informationen aller Formularseiten beisammen, lässt sich der Erstellvorgang abschließen.

Die über org.eclipse.ui.newWizards eingebundenen Assistenten erscheinen automatisch nach Start der IDE in den Erstellungsmenüs, müssen allerdings bei der Erstinstallation des Plug-Ins einmalig für die IDE-Menüs freigeschaltet werden (vgl. Anhang A). Bsp. 4.14. zeigt einen Ausschnitt aus dem Plug-In-Manifest, in dem einige für das Einbinden des Wizards benötigte Informationen aufgeführt sind. Dazu gehören ein Name, der zur Anzeige in den Menüs benutzt wird, eine ID sowie ein optionales Icon.

```
<extension point="org.eclipse.ui.newWizards">
<wizard
    name="%New.wizard"
    icon="icons/logo.gif"
    class="org.eclipse.contribution.smp.wizards.NewMigrationProjectWizard"
    ...
    id="org.eclipse.contribution.wizards.NewMigrationProjectWizard">
    <description>%New.wizardDescription</description>
  </wizard>
</extension>
```

Bsp. 4.14.: Einbinden eines Wizards (im Plug-In-Manifest)

4.3.6. Segmentieren und Zuordnen von Segmenten

Beim Segmentieren müssen Teile des Quelltexts geeignet markiert und dadurch in ein Segment eingefasst werden, das wiederum alle nötigen Segmentierungsinformationen enthält. Das Erstellen von einzelnen Segmenten erfolgt im Quelltext. Daher wird diese Funktionalität als Kontextmenüeintrag in den zur Verfügung stehenden Editoren angeboten, in denen der Quelltext bearbeitet wird. Damit nicht beliebige Texte in den Editoren segmentiert werden können, ist die Funktionalität auf *Java*-, *JSP*- und *PHP*-Dateien beschränkt.

Die Umsetzung der Segmentierung kann dabei auf mehrere Arten erfolgen. Das hängt von den Möglichkeiten des Zugriffs auf die internen Dokumentstrukturen bzw. vom gewünschten Detaillierungsgrad ab, mit dem ein Benutzer den Quelltext segmentieren möchte. Zum Beispiel benötigt eine Funktion, die automatisiert Quelltext nach gewissen Kriterien segmentiert, Zugriff auf eine Form von Dokumentstruktur. Diese dient als Entscheidungsgrundlage, um im Quelltext Abschnitte identifizieren zu können, die in Segmente eingefasst werden können. Die folgende Auflistung enthält einige Alternativen, wie die Segmentierung umgesetzt werden kann:

1) Der Ort im Quelltext, an dem über die rechte Maustaste das Kontextmenü geöffnet wird, gibt den Anhaltspunkt für eine Segmentierung. Zum Beispiel kann eine Methode, in der der Kontextmenüaufruf stattfindet, als Ganzes in ein Segment eingefasst werden. Das setzt voraus, dass der Quelltext Strukturen enthält, die ein automatisiertes Einfassen in Segmente erlauben, zum Beispiel klar abgegrenzte Methodenkörper.
2) Eine Selektion im Quelltext bestimmt das Segment. Der markierte Quelltext wird dabei in ein Segment eingefasst und mit Segmentierungsinformationen versehen. Diese Variante entspricht dem in Kapitel 4.2. beschriebenen Szenario für das Erstellen von Segmenten im Quelltext.

3) Der gesamte im Editor angezeigte Quelltext wird per Kontextmenübefehl automatisch segmentiert. Bei dieser Variante muss der Quelltext über Strukturen verfügen, die automatisch in Segmente gefasst werden können.
4) Ein Kontextmenübefehl im Navigator oder Package Explorer von *Eclipse* lässt den kompletten Quelltext einer Datei automatisiert in Segmente zerlegen. Ähnlich der Segmentierung im Editor muss bei dieser Art der Segmentierung der jeweilige Quelltext eine Struktur aufweisen, die sich automatisch in Segmente fassen lässt.
5) Ein Kontextmenübefehl in der Klassenübersicht im Navigator bzw. in der zum Editor gehörigen Outline kann dort ausgewählte Methoden in Segmente fassen.

In der Migrationsunterstützung wurden die Varianten 2) und 4) umgesetzt. Bei der Arbeit im Quelltext steht die Segmentierung von beliebigen Quelltextabschnitten im Vordergrund, bei der hauptsächlich Bruchteile von Methoden und Funktionen sowie Teile von wenig strukturiertem *PHP*-Quelltext in Segmente gefasst werden. Daher wurde die Segmentierung anhand von Selektionen im Quelltext bevorzugt und auf eine zusätzliche Segmentierungsfunktion gemäß Variante 1) verzichtet.

Das automatische Segmentieren ganzer Quelltextdateien nach Variante 4) wurde nur prototypisch für die Segmentierung von *Java*-Dateien umgesetzt. Die interne Struktur von *Java*-Klassen kann über das JDT (*Java Development Toolkit*, siehe Kapitel 2) zugegriffen werden. Diese Struktur kann das Plug-In dafür nutzen, die Methoden und Funktionen einer *Java*-Datei in Segmente zu fassen. Der Zugriff auf *PHP*-Dokumentstrukturen über das verwendete *PHP*-Plug-In hingegen ist für andere Plug-Ins nicht erlaubt. Dieser Punkt wird weiter unten noch einmal genauer erläutert.

Da Variante 3) die Segmentierung ganzer Quelltextdateien aus dem Editor erlaubt, es sich allerdings um dieselbe Funktionalität wie in Variante 4) handelt, wurde auf die Implementierung von Variante 3) verzichtet. Da diese Funktion auf kompletten Quelltextdateien arbeitet, erscheint die Verknüpfung dieser Funktionalität mit den im Package Explorer oder Navigator dargestellten Dateiobjekten naheliegender.

Variante 5) ähnelt Variante 1) in der Hinsicht, dass eine Methode (oder eine andere in der Übersicht aufgeführte Struktur) den Inhalt des zu erstellenden Segments bestimmt. Diese Variante wurde aus mehreren Gründen für die erste Fassung des *Software Migration Plug-Ins* nicht berücksichtigt. Diese Funktionalität wäre wiederum nur auf *Java*-Dateien beschränkt, da der Zugriff auf die Outline im verwendeten *PHP*-Plug-In ebenso gesperrt ist wie auf die Dokumentstruktur von *PHP*-Dateien. Des Weiteren verwendet das Plug-In für die Visualisierung der Segmente im Quelltext einen eigenen Editor, dessen Outline die vorhandene Segmentierung einer Quelltextdatei widerspiegelt und die Dokumentstrukturübersicht ersetzt. Somit wäre Variante 5) einzig in die Implementierung von Segmentierungsfunktionen im *Java*-Editor eingeflossen, weshalb darauf verzichtet wurde. Diese Variante stellt dennoch eine gute Alternative bzw. Erweiterung zu den im Plug-In implementierten Möglichkeiten dar.

Um im Quelltext beliebige Abschnitte in Segmente einfassen zu können, wird dem Plug-In-Benutzer eine kontextsensitive Lösung (gemäß Variante 2) angeboten. Dabei markiert der Benutzer im Quelltext einen beliebigen Bereich, ruft auf dieser Selektion das Kontextmenü auf und wählt darin die Funktion für das Erstellen eines Segments. Dabei werden die Informationen zu Segmenten zusammen mit der Start- und Endmarkierung im Quelltext abgelegt. Eine detaillierte Beschreibung der im Quelltext gespeicherten Segmentierungsinformationen findet sich im Anhang C. Die aus den Informationen ableitbaren Segmentzustände sind genauer in Kapitel 4.2.2. beschrieben.

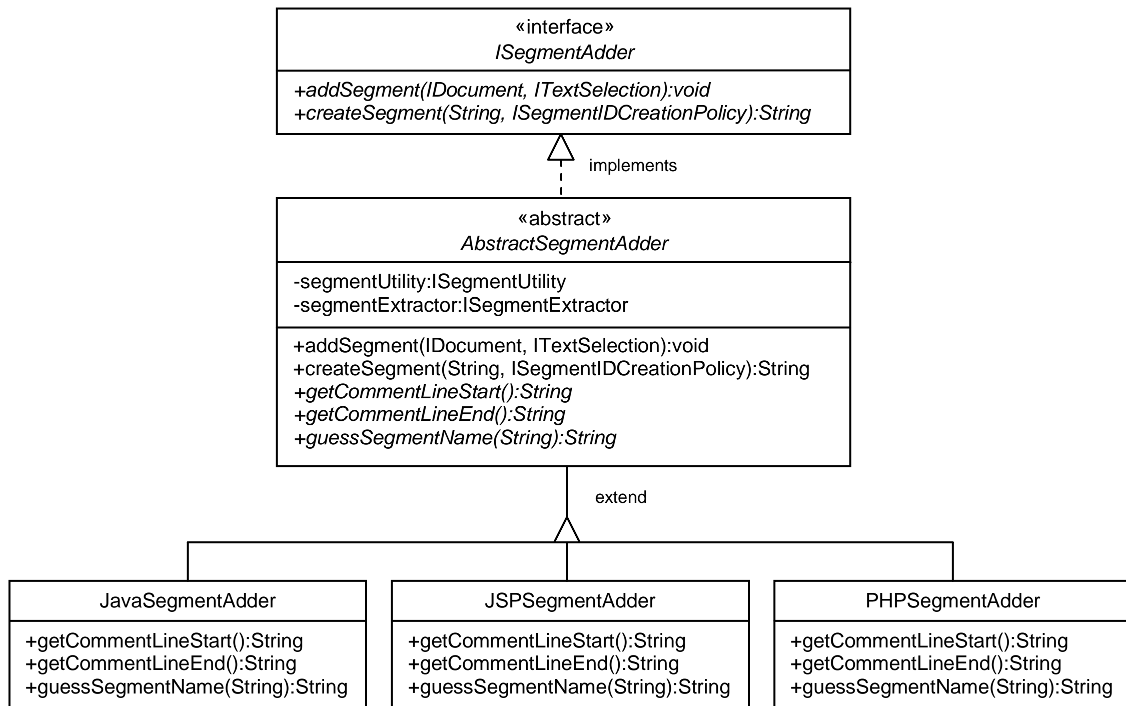


Abb.. 4.15.: Das Klassendiagramm der unterschiedlichen *Segmentierer*-Klassen

Die Segmentierung ist über die allgemeine Schnittstelle ISegmentAdder angebunden (Abb. 4.15.). Diese Schnittstelle beschreibt einen allgemeinen *Segmentierer*, dessen Aufgabe es ist, in Quelltextdokumenten Abschnitte in Segmente einzufassen. An dieser Schnittstelle werden zwei Methoden angeboten. Die Methode addSegment erhält das Quelltextdokument sowie den Abschnitt, der in ein Segment eingefasst werden soll. Die Methode createSegment fasst den Quelltextabschnitt dann in Segmentkommentare ein und versieht das Segment mithilfe einer ISegmentIDCreationPolicy mit einer eindeutigen Identifikationsnummer.

Die Segmentierung wird über die abstrakte Klasse AbstractSegmentAdder implementiert. Diese Klasse gibt die abstrakten Methoden getCommentLineStart sowie getCommentLineEnd vor, die durch die konkreten Segmentierungsklassen jeweils für *Java*-, *JSP*- und *PHP*-Quelltextkommentare implementiert werden.

Die Methode guessSegmentName liefert für einen gegebenen Quelltextabschnitt einen Namensvorschlag. Diese Methode wird speziell von der *PHP*-Implementierung der Segmentierungsfunktionalität genutzt, um einem neu erstellten Segment automatisch einen Namen setzen zu können, der sich aus einer im Segment vorhandenen Funktionssignatur herleitet.

Der Segmentierer arbeitet mit Instanzen von Klassen, die die Schnittstellen ISegmentUtility und ISegmentExtractor erfüllen, um aus einem Dokument Informationen über schon vorhandene Segmente zu erhalten. Gemäß der weiter oben schon beschriebenen *dependency injection* werden diese Instanzen dem Segmentierer bei der Erzeugung mitgegeben. Die beiden Schnittstellen werden in Kapitel 4.3.2. über die Quelltextrepräsentation von Segmenten beschrieben.

Der Segmentierer nutzt den ISegmentExtractor, um im Quelltext erkennen zu können, welche Teile davon schon segmentiert und welche noch offen sind. Aufgrund der strikten 1:1-Zuordnung bzw. 1:n-Zuordnung von Segmenten zueinander erlaubt die Segmentierfunktion allerdings nicht die Verschachtelung von Segmenten (vgl. Kapitel 3.2. und Kapitel 3.3.). Wird innerhalb eines vorhandenen Segments eine Selektion in ein neues Segment eingefasst, so wird das ursprüngliche Segment in drei neue Segmente aufgeteilt. Das ursprüngliche Segment besteht dann aus dessen Kopf bis zur Selektion, sowie zwei weiteren aus der Selektion und dem Rest des ursprünglichen Segments ab Selektionsende. Diese

Funktionalität erlaubt dem Benutzer ein schnelles Segmentieren des Quelltexts, ohne vorhandene Segmente von Hand löschen oder umbauen zu müssen.

Der Segmentierung von Quelltext folgt die Zuordnung von Segmenten des Zielprojekts zu Segmenten des Ursprungsprojekts. Das erlaubt eine detaillierte Zuordnung von ursprünglicher zu migrierter Funktionalität beider Projekte. Diese Zuordnung erfolgt ebenso wie die Segmentierung im Quelltext. Dazu kann der Benutzer im Quelltext über eine Kontextfunktion einen Zuordnungsdialog in Form eines Wizards aufrufen. Dieser Dialog bietet dem Benutzer das Ursprungsprojekt in einer Baumansicht an, die der Ressourcenansicht im Navigator bzw. Package Explorer nachempfunden ist. Das erleichtert dem Benutzer die Suche nach der Ursprungsquelltextdatei, in der sich das zuzuordnende Segment befindet. Unterhalb der Quelltextdateieinträge in dieser Ansicht werden die in der Datei enthaltenen Segmente aufgelistet. Der Benutzer wählt eins dieser Segmente aus und beendet den Dialog. Das ausgewählte Segment wird dann dem Segment zugeordnet, auf dem die Kontextfunktion aufgerufen wurde. Die für eine Zuordnung benötigte Information über Ursprungs- und Zielprojekt wird aus der in einem Projekt abgelegten Migrationsprojektdatei ausgelesen. Kann die Kontextfunktion diese Datei oder ein in ihr enthaltenes Projekt nicht finden, wird dem Benutzer eine entsprechende Meldung angezeigt.

Die Informationen über die Zuordnung werden am Zielsegment abgelegt. Die Änderung der Segmentierungsinformationen erfolgt über die in Abb. 4.16. dargestellte ISegmentApplier-Schnittstelle.

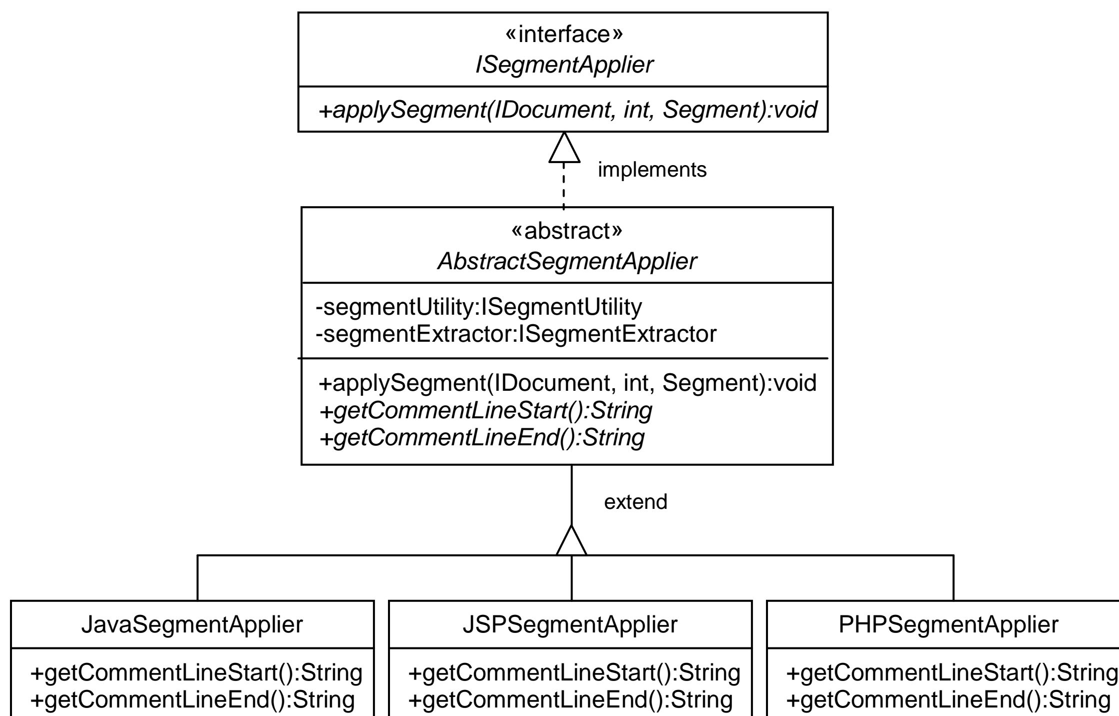


Abb. 4.16.: Das Klassendiagramm der ISegmentApplier-Schnittstelle und ihrer Implementierungen

Um die am Segment gespeicherten Informationen zu aktualisieren, werden der Änderungsfunktion applySegment das Quelltextdokument, die Zeilennummer des Kontextaufrufs sowie das Segment mit den veränderten Informationen mitgegeben. Die Funktion ersetzt und erweitert dann die in den Segmentkommentaren abgelegten Daten. Die Unterscheidung der Kommentarformen in *Java*, *JSP* und *PHP* geschieht hier wieder in den vom AbstractSegmentAdder abgeleiteten Klassen.

In der ersten Fassung des Plug-Ins befanden sich sämtliche Informationen am Segment selbst. Zu diesen Informationen gehörten die Verknüpfungsinformationen mit dem Ursprungsprojekt sowie der Ursprungsdatei und dem Segment. Einige dieser Informationen

wurden dabei als projektweit eindeutig und unveränderlich identifiziert und wieder entfernt, um die Menge der Informationen einzuschränken, die am Segment gespeichert werden müssen. Kapitel 4.3.2. befasst sich eingehender mit der Speicherung der Segmentierungsinformationen und den sich daraus ergebenden Konsequenzen für den Quelltext. Dieser Punkt wird hier deshalb erwähnt, da ein Großteil dieser Informationen bei der Zuordnung von Segmenten zueinander entsteht und daher für das jeweilige Segment gespeichert und angezeigt wird.

Einbindung von Kontextfunktionen im Editor

Die Einbindung der Kontextfunktionalität für das Segmentieren und das Zuordnen von Segmenten geschieht im Editor über *viewer contributions*. Diese werden im Plug-In-Manifest für die verschiedenen Editoren registriert. Bsp. 4.17. zeigt diese Einbindung für das Segmentieren in den Kontext des *Java*-Editors in *Eclipse*. In der ersten Fassung des Plug-Ins wurden diese Kontextaktionen für alle Editoren registriert, allerdings erlaubt die Zuordnung von einer Kontextaktion zur targetID eines Kontexts eine differenzierte Registrierung. Die Zuordnung von Segmenten zueinander erfolgt zum Beispiel nur von Zielsegment zu Ursprungssegment. Im *WebMig*-Projekt erfolgt die Zuordnung ausschließlich aus den *Java*- und *JSP*-Dateien des Zielprojekts heraus, so dass diese Funktionen nur für die entsprechenden Editoren registriert werden.

```
<viewerContribution
  targetID="#CompilationUnitEditorContext"
  id="org.eclipse.contribution.smp.actions.addSegment.default">
  <action
    label="%ContextMenu.segmentSource.label"
    icon="icons/logo.gif"
    class="org.eclipse.contribution.smp.actions.AddSegmentAction"
    ...
    id="org.eclipse.contribution.smp.actions.AddSegmentAction">
  </action>
</viewerContribution>
```

Bsp. 4.17.: Einbinden von Kontextaktionen (im Plug-In-Manifest)

Kontextsensitives Verhalten in *Eclipse*

Die Zuordnung von Segmenten zueinander wurde im Plug-In über die Einbindung von Kontextmenüeinträgen in die Editoren gelöst. Dabei fiel auf, dass sich diese kontextsensitive Funktion in *Eclipse* nicht erwartungskonform verhielt. Als Beispiel sei die Situation gegeben, dass der Benutzer im Quelltext einer *Java*-Datei ein dort vorhandenes Segment mit der rechten Maustaste anklickt, um über das Kontextmenü den Zuordnungsdialog für dieses Segment aufzurufen. Die Kontextfunktionen in Editoren arbeiten auf Selektionen. Diese transportieren ausschließlich Start und Ende einer Selektion oder – falls sich keine Selektion im Quelltext befindet – die aktuelle Textcursorposition. In der vorliegenden Version des Plug-Ins wird die Funktion ausgehend von dem Segment – falls vorhanden – aufgerufen, welches sich unter der jeweils aktuellen Textcursorposition befindet.

Dieses Verhalten wurde zuerst als Fehler gemeldet, da es intuitiv unverständlich erscheint, warum die Klickposition in diesem Fall nicht Ausgangspunkt für einen solchen Kontextfunktionsaufruf ist. In den nachfolgenden Tests stellte sich allerdings heraus, dass dieses Verhalten in den verwendeten *Eclipse*-Versionen (3.1.2 und 3.2.1) den Standard darstellt und dass sich Kontextfunktionen im Allgemeinen in *Eclipse* auf diese Art und Weise verhalten – also dass alle Kontextfunktionen sich auf Textcursorpositionen oder Selektionen, nicht aber auf die Position des Rechtsklicks beziehen. Der Aufruf einer Kontextfunktion im Editor führt nicht zu einer Neupositionierung des Textcursors. Dies geschieht explizit zum Beispiel durch einen Linksklick im Text.

4.3.7. Markieren von Segmenten als obsolet

Das Markieren von Segmenten als obsolet erfolgt über eine Kontextfunktion im Quelltext. Wird dort in einem Segment die Funktion aufgerufen, wird der Name des Segments mit dem Schlüsselwort <obsolete> versehen.

Abb. 4.18. zeigt die Schnittstelle ISegmentObsoleteMarker für das Markieren von Segmenten als obsolet. Die Funktionalität zum Setzen dieser Markierung wird im abstrakten AbstractSegmentObsoleteMarker implementiert. Das Quelltextdokument sowie der Ort des Aufrufs im Dokument in Form einer Quelltextselektion wird der Methode markSegmentAsObsolete hineingereicht. Die konkreten Implementierungen für die einzelnen Sprachen liefern die für die Erkennung der Segmentkommentare nötigen Kommentarbegrenzungen.

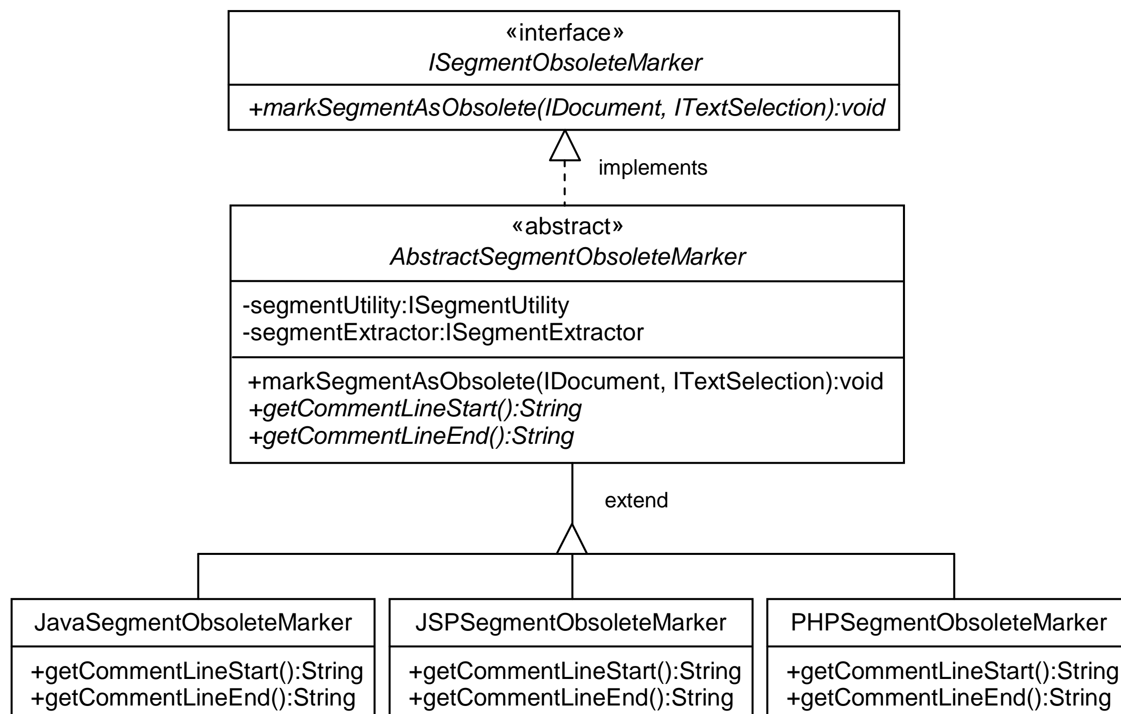


Abb. 4.18.: Schnittstelle und Implementierungen von ISegmentObsoleteMarker

Die Einbindung in die Oberfläche der IDE erfolgt wie beim Segmentierer über *viewer contributions*. Bsp. 4.19. zeigt die Einbindung der Markierungsfunktionalität im Plug-In-Manifest. Um diese Funktionalität in den Editoren aller im Migrationsprojekt verwendeten Programmiersprachen anzubieten, wird die Funktion am allgemeinen Kontext der Texteditoren registriert.

```
<viewerContribution targetID="#TextEditorContext"
  id="org.eclipse.contribution.smp.actions.markSegmentAsObsolete.default">
  <action
    label="%ContextMenu.markSegmentAsObsolete.label"
    icon="icons/logo.gif"
    class="org.eclipse.contribution.smp.actions.MarkSegmentAsObsoleteAction"
    id="org.eclipse.contribution.smp.actions.MarkSegmentAsObsoleteAction">
  </action>
</viewerContribution>
```

Bsp. 4.19.: Einbinden der Kontextaktion zum Markieren obsoleter Segmente (im Plug-In-Manifest)

4.3.8. Analyse des Migrationsfortschritts

Um anhand der Segmentierung des Quelltexts und der Zuordnung der Segmente zueinander Aussagen über den Fortschritt der Migration gewinnen zu können, bietet das *Software Migration Plug-In* eine Analysefunktion an. Die Analyse läuft über das Migrationsprojekt und gewinnt Kennzahlen aus der vorhandenen Segmentierung.

Die Analyse durchläuft alle Quelltextdateien eines Projekts und sammelt jeweils die Größe der Dateien (in *LOC*[1] gerechnet), die enthaltenen Segmente und die Anzahl Zeilen *relevanten Quelltexts* ein. Als relevant, im Sinne der Migration, gilt Quelltext genau dann, wenn es sich nicht um eine Leerzeile oder eine Kommentarzeile handelt. Eine Klassifizierung hinsichtlich der Relevanz von Quelltextzeilen ist deshalb interessant, da sie eine Aussage erlaubt, wie viele *LOC* mit Funktionalität schon segmentiert bzw. migriert wurden. Die Berücksichtigung von Kommentarblöcken und Leerzeilen könnte diese Kennzahl verfälschen.

MigrationProgressCollector
-project:IProject
+determineCoverage(IProgressMonitor):Map +calculateCoverageForFolder(IFolder, Map):double +calculateCoverageForFile(IFile):double #isLineRelevant(String):boolean

Abb. 4.20.: Die Klasse MigrationProgressCollector

Die in Abb. 4.20. gezeigte Klasse MigrationProgressCollector durchläuft in einem Projekt alle Verzeichnisse und Dateien und ermittelt die jeweilige prozentuale Abdeckung des Quelltexts durch Segmente. Bei der Analyse des Quelltexts wird mit der Methode isLineRelevant für jede Zeile geprüft, ob die gegebene Zeile eine Leer- oder Kommentarzeile ist und somit aus der Berechnung der Abdeckung herausgenommen wird. Der MigrationProgressCollector wird mit einem Projekt (IProject) instanziert und mit determineCoverage gestartet. Dieser Methode wird eine Fortschrittsanzeige vom Typ IProgressMonitor mitgegeben, da es sich um einen langwierigen Prozess handelt und somit dem Benutzer der Fortschritt der Analyse angezeigt werden kann.

Das Ergebnis der Analyse wird in einer Baumansicht ähnlich der Ressourcenansicht im Navigator bzw. Package Explorer aufbereitet und angezeigt. Das Projekt selbst stellt die Wurzel dieser Baumansicht dar. Die Quelltextdateien befinden sich in diesem Baum an den äußersten Knoten, an denen die Segmente, sofern vorhanden, als Blätter hängen. An einer Quelltextdatei steht der prozentuale Anteil von *LOC* segmentierten Quelltexts zu gesamtem Quelltext. In beiden Fällen wird nur der relevante Quelltext miteinbezogen. Verzeichnisse sind in diesem Baum mit dem Durchschnitt der Prozentwerte aller darunterliegenden Dateien und Verzeichnisse versehen. Der Wurzelknoten enthält somit die durchschnittliche Segmentierungsabdeckung über das gesamte Projekt.

Im Zielprojektbaum wird der Status der Segmente (*offen, migriert* und *nachträglich verändert*) unterhalb des jeweiligen Quelltextknotens farblich angezeigt. Die Farbgebung entspricht der in Kapitel 4.2.2. detailliert beschriebenen Auswahl an Farben für den jeweiligen Status. Zusätzlich sind die Segmentblätter mit dem Status in Textform versehen. Diese Information dient dazu, die Aussagekraft der Auswertung nicht alleine auf die Farbgebung zu stützen. Per Doppelklick auf ein Segmentblatt gelangt der Benutzer direkt in die dazugehörige Quelltextdatei und das dazugehörige Segment. So lässt sich die Auswertung dazu nutzen, um zum Beispiel als *nachträglich verändert* angezeigte Segmente nacheinander betrachten und korrigieren zu können.

[1] *LOC*, *lines of code*, Quelltextzeilen.

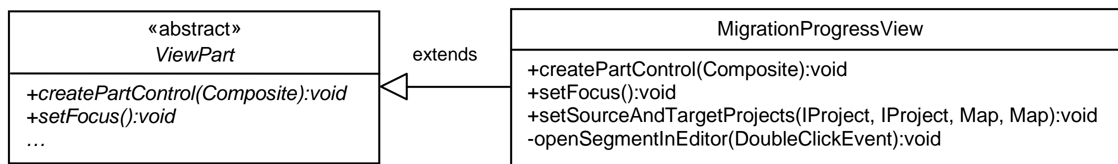


Abb. 4.21.: Die Klassen ViewPart und MigrationProgressView

Die Darstellung der Auswertung erfolgt in einem ViewPart, einer Oberflächenkomponente von *Eclipse*. Der MigrationProgressView implementiert diese abstrakte Klasse (Abb. 4.21.) und stattet eine Composite-Instanz mit den oben beschriebenen Bäumen aus. Dieses Composite ist ein generischer Rahmen für die internen Fenster in der *Eclipse*-Oberfläche und erlaubt das Einbinden frei gestaltbarer Oberflächenkomponenten. Der Rahmen dient dazu, diese Oberflächenkomponenten mit Funktionalität zum Verschieben und Einbinden innerhalb der IDE auszustatten.

Über die Methode setSourceAndTargetProjects werden dem Auswertungsfenster die anzuzeigenden Projekte (Ursprungs- und Zielprojekt der Migration) sowie die für diese Projekte ermittelten Abdeckungszahlen gesetzt. Über die interne Methode openSegmentInEditor wird das Dokument und Segment im Editor geöffnet und angezeigt, auf dem im MigrationProgressView ein Doppelklick ausgeführt wurde.

Die Auswertung bietet verschiedene Möglichkeiten der Erweiterung. Mithilfe eines Filters könnten die in der Auswertung dargestellten Bäume nach bestimmten Kriterien wie Segmentstatus oder Mindestabdeckung gefiltert werden, um gezielt bestimmte Bereiche des Projekts zu betrachten. In der vorliegenden Fassung des *Software Migration Plug-Ins* wurde allerdings noch keine Filterfunktion implementiert, da die Frage nach einer Verfeinerung der Auswertung in den jetzigen Anforderungen noch keine Berücksichtigung fand.

Einbindung von Informationsfenstern (*Views*) in die IDE

Genauso wie die schon in Kapitel 4.2.2. beschriebenen dateiweiten Segmentierungsfunktionen wird die Analyse als Kontextfunktion im Navigator bzw. Package Explorer angeboten. Es handelt sich dabei um eine *object contribution*, die Aktionen auf Ressourcen vom Typ IProject ausführt. Wird auf einem Projekt die Analysefunktion aus dem Kontextmenü aufgerufen, wird im Projekt nach der Migrationsprojektdatei gesucht. Ist eine solche Datei vorhanden, wird anhand der darin enthaltenen Informationen über Ursprungs- und Zielprojekt der Migration die Auswertung gestartet.

Die Auswertung selber wird über einen *view* eingebunden. *Views* sind in *Eclipse* Informationsfenster. Das Plug-In nutzt einen *view*, um damit die Ergebnisse aus der Auswertung anzuzeigen. In Bsp. 4.22. ist der Ausschnitt aus dem Plug-In-Manifest dargestellt, in dem das Informationsfenster registriert wird. Mittels der *category* wird ein *view* einer bestimmten Kategorie, in diesem Fall der *Java*-Entwicklungsumgebung, zugeordnet.

```
<extension point="org.eclipse.ui.views">
  <view name="%View.MigrationProgress"
        icon="icons/logo.gif"
        class="org.eclipse.contribution.smp.views.MigrationProgressView"
        category="org.eclipse.jdt.ui.java"
        id="org.eclipse.contribution.smp.views.MigrationProgressView" />
</extension>
```

Bsp. 4.22.: Einbinden eines *views* (im Plug-In-Manifest)

4.3.9. Umsetzung zusätzlicher Funktionen

Die in den Szenarios beschriebenen Arbeitsschritte für Navigation zwischen Segmenten und Validieren von nachträglich geänderten Segmenten lassen sich als einfache Kontextfunktionen auf Segmenten implementieren. Bei diesen Funktionen handelt es sich um Werkzeuge, die dem Benutzer seine Arbeit wesentlich erleichtern können, aber nicht essenziell für die Migration sind. Das heißt, dass der Benutzer ohne diese Funktionen auskommen kann, obgleich die Arbeit sich dann schwieriger gestaltet. Daher sind diese Funktionen als Arbeitserleichterungen zusätzlich zu den eigentlichen Arbeitsschritten der Segmentierung und Zuordnung anzusehen. Gemäß den Szenarios gehören die folgenden drei Funktionen zu diesen Arbeitserleichterungen:

1) Um Änderungen schon migrierter Funktionalität leichter überprüfen und nachvollziehen zu können, kann der Benutzer zwischen einander zugeordneten Segmenten navigieren.
2) Zudem kann er den Status der Segmente nach einer Überprüfung und eventuellen Nacharbeit am Zielsegment aktualisieren und auf migriert zurücksetzen.
3) Zur Erleichterung der Segmentierung steht dem Benutzer eine automatisierte Segmentierung von kompletten *Java*-Quelltextdateien zur Verfügung.

Navigieren zwischen Segmenten

Das Navigieren ist die einfachste der drei Funktionen. Sie benutzt eine Instanz vom Typ ISegmentExtractor, um aus einem Dokument für die Position des Kontextaufrufs das Segment auszulesen. Findet die Funktion ein Segment und ist dieses einem Ursprungssegment zugeordnet, dann wird die zugehörige Ursprungsquelltextdatei geöffnet und der zum Segment gehörende Quelltextabschnitt in die Ansicht geholt.

Validieren von Segmenten

Für das Validieren wird ein Segment auf Änderungen eines zugeordneten Ursprungssegments angepasst, das heißt, dass die von der Änderungsverfolgung im Segment abgelegte Prüfsumme für das geänderte Ursprungssegment neu berechnet und entsprechend aktualisiert wird.

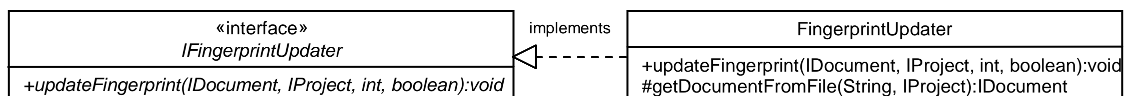


Abb. 4.23.: Die Schnittstelle der FingerprintUpdater-Klasse

In Abb. 4.23. ist die Schnittstelle der FingerprintUpdater-Klasse für das Validieren von Segmenten abgebildet. Die Schnittstelle ist sehr einfach gehalten und es existiert nur eine Implementierung. Da der FingerprintUpdater explizit die Prüfsumme in den Segmentkommentaren ersetzt und diese eine feste Länge besitzt, werden keine speziellen Implementierungen für die unterschiedlichen Kommentarformen benötigt. Die Kommentarmarke für das Prüfsummenattribut (vgl. Anhang C) reicht aus, um die Prüfsumme in den Segmentkommentaren zu lokalisieren und zu ändern.

```
<viewerContribution
  targetID="#CompilationUnitEditorContext"
  id="org.eclipse.contribution.smp.actions.updateSourceFingerprint.java">
  <action
    label="%ContextMenu.updateSourceFingerprint.label"
    class="org.eclipse.contribution.smp.actions.UpdateSourceFingerprintAction"
    ...
    id="org.eclipse.contribution.smp.actions.UpdateSourceFingerprintAction">
  </action>
</viewerContribution>
```

Bsp. 4.24.: Einbinden von Kontextaktionen im Editor (im Plug-In-Manifest)

Wie bei der Segmentierungsfunktion erfolgt die Einbindung der Kontextfunktionalität für das Navigieren und Validieren im Editor über *viewer contributions*. Bsp. 4.24. zeigt die Registrierung der Validierungsfunktion. Da das Navigieren dazu dient, dass der Benutzer direkt von einem Zielsegment ins zugeordnete Ursprungssegment springen kann, findet sich diese Funktionalität in den Editoren des Zielprojekts. Ebenso wird das Validieren für das Zielprojekt registriert, da nur die Zielsegmente nach einer Änderung im Ursprungssegment den Zustand wechseln und sich das Validieren daher auf Segmente im Zielprojekt beschränkt.

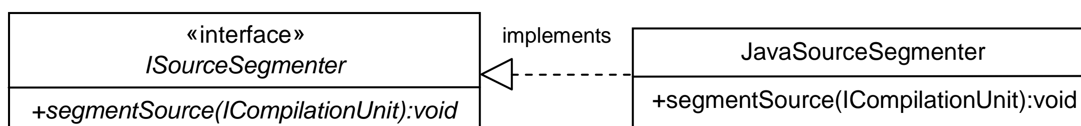
Automatisierte Segmentierung von *Java*-Dateien
Die Migrationsunterstützung bietet eine automatische Segmentierung von *Java*-Dateien an, mit der die Funktionen und Methoden einer in der Datei enthaltenen *Java*-Klasse in Segmente eingefasst werden. Diese Automatisierung nimmt dem Plug-In-Benutzer einen Großteil der standardmäßigen Einteilung von Quelltexten ab, wenn bei der Migration überwiegend nur Funktionen und Methoden umgesetzt werden.

Automatisierte Segmentierung von *PHP*-Dateien
Diese Unterstützung war ursprünglich ebenso für die *PHP*-Dateien des Ursprungssystems vorgesehen. Für die Segmentierung von *Java*-Dateien bietet die IDE mit ihren Werkzeugen ein Modell der im *Java*-Editor bearbeiteten Dateien an, über das die Struktur der Quelltextdatei erfasst wird. So erlaubt dieses Modell den Zugriff auf die einzelnen Strukturelemente der in einer Quelltextdatei spezifizierten Klasse, unter anderem die Methoden und Funktionen. Dadurch kann mithilfe des Modells diese Information für *Java*-Quelltextdateien ausgelesen und zur Segmentierung herangezogen werden.

Bei *PHP*-Dateien existiert für das in der Arbeit verwendete *PHP IDE* Plug-In ein solches Modell. Das *PHP IDE* Plug-In extrahiert diese Informationen aus der Quelltextdatei, um das *syntax highlighting* und die Outline zur Verfügung stellen zu können. Für die Verwendung im *Software Migration Plug-In* steht dieses Modell allerdings nicht zur Verfügung, da der Zugriff darauf durch das *PHP IDE* Plug-In gesperrt ist.

Eine Lösung für dieses Problem wäre durch ein externes Werkzeug zum Parsen und Strukturieren von *PHP*-Dateien gegeben. Da sichergestellt werden müsste, dass die *PHP*-Version für das interne *PHP*-Modell und für das im *Software Migration Plug-In* verwendete *PHP*-Modell dieselbe ist, wurde auf eine Umsetzung - in Hinblick auf den unverhältnismäßig hohen Aufwand für die Bereitstellung einer automatisierten *PHP*-Segmentierung - in dieser Arbeit verzichtet.

Einbindung der automatischen *Java*-Segmentierung
Für das automatisierte Segmentieren von Dateien ist im Plug-In die in Abb. 4.25. gezeigte Schnittstelle ISourceSegmenter vorgesehen. Diese arbeitet auf einer Einheit, der ICompilationUnit, die die Klassenstruktur der in der Quelltextdatei beschriebenen Klasse enthält. Aus den oben dargelegten Gründen existiert in der vorliegenden Fassung des Plug-Ins nur eine Implementierung für *Java*-Dateien.



Bsp. 4.25.: Das Klassendiagramm der Schnittstelle ISourceSegmenter und der *Java*-Implementierung

Eclipse bietet die Einbindung von Kontextfunktionalität auf Ressourcentypen an. So besteht die Möglichkeit, über entsprechende *extension points* Kontextaktionen für *Java*-, *JSP*- und *PHP*-

Dateien zu registrieren. Die Einbindung der Kontextfunktionalität erfolgt wie bei den Kontextfunktionen im Quelltext über den *extension point* org.eclipse.ui.popupMenus, über den alle Kontextaktionen der IDE registriert werden. Im Unterschied zu den Kontextfunktionen im Quelltext arbeitet die automatische Segmentierung auf Ressourcen wie Dateien und Klassen. Dieser Typ von Kontextfunktion wird als *object contribution* (wird im Plug-In-Manifest als objectContribution eingebunden) bezeichnet. Eine *object contribution* verbindet jeweils einen Ressourcentyp im Workspace von *Eclipse* mit dazugehöriger Kontextfunktionalität.

```
<objectContribution id="org.eclipse.contribution.smp.tool.segmentFile.java"
  objectClass="org.eclipse.core.resources.IFile"
  nameFilter="*.java">
  <action
    id="org.eclipse.contribution.smp.tool.segmentFile.java.segment"
    label="%ContextMenu.label"
    ...
    class="org.eclipse.contribution.smp.actions.SegmentSourceAction"
    enablesFor="1">
  </action>
</objectContribution>
```

Bsp. 4.26.: Einbinden von Kontextaktionen auf Ressourcen (im Plug-In-Manifest)

Bsp. 4.26. zeigt die Registrierung für Kontextfunktionen auf Ressourcen im Plug-In-Manifest. Die gezeigte *object contribution* wird für einen bestimmten Ressourcentyp, mit einem (optionalen) Namensfilter sowie für eine Einfachselektion eingetragen.

4.4. Testen der Anwendung

Um eine hohe Qualität bei der Entwicklung des *Software Migration Plug-Ins* zu gewährleisten, wurden wesentliche Teile der Entwicklungsarbeit durch Tests abgestützt. Diese Tests erlauben ein Sicherstellen der Konsistenz der entwickelten Funktionalität trotz vieler Änderungen (so genannte *Refactorings*[1]) am Plug-In, die während der Weiterentwicklung durchgeführt wurden. Die unterschiedlichen Teile des Plug-Ins erlauben unterschiedliche Formen des Testens, die im Folgenden kurz beschrieben werden. Johannes Link bietet in seinem Buch über das Testen in *Java* eine Übersicht über die verschiedenen Testformen (vgl. [Link 2002], S. 5-6). Ziel der Implementierung von Tests war eine hundertprozentige Testabdeckung der Plug-In-Funktionalität. Eine Ausnahme hinsichtlich der Testabdeckung bilden die Aktionsklassen, die die Oberflächenfunktionalität an die Plug-In-Benutzungsschnittstelle binden. Diese Aktionen delegieren den Aufruf an eine zugehörige Funktionalitätsklasse weiter, die die eigentliche Funktionalität kapselt. Da die Aktionen somit nur die weiterleitenden Aufrufe enthalten und selbst nichts zur eigentlichen Funktionalität beitragen, wurde auf Tests dieser trivialen Delegation verzichtet. Des Weiteren ist der korrekte Aufruf der einer Oberflächenkomponente zugeordneten Aktion integraler Bestandteil der *Eclipse*-Funktionalität.

Eine weitere Ausnahme bilden die *Registraturklassen*, die als zentrale Sammelstellen zum Registrieren und Abrufen von überall im Plug-In verwendeten Instanzen genutzt werden. Dazu gehören die SMPPlugin-Klasse, die ToolRegistry-Klasse und die SMPConstants-Klasse. In der SMPPlugin-Klasse werden hauptsächlich die im Plug-In verwendeten Grafiken erzeugt und bereitgestellt. Die ToolRegistry hält für die unterschiedlichen Funktionen (Segmentieren, Zuordnen, Markieren) Instanzen bereit und sorgt für deren Initialisierung beim ersten Zugriff. Die SMPConstants-Klasse bündelt die im Plug-In verwendeten Konstanten.

[1] *„Refactoring ist die Umstrukturierung eines Softwaresystems unter Beibehaltung seiner Funktionalität. […] Durch Refactoring kann die Struktur eines Softwaresystems flexibel gestaltet und ständig mit wenig Aufwand an die aktuellen Bedürfnisse angepasst werden.“* – ([Lippert 2002], S. 76)

Da die Registraturklassen keine wesentliche Funktionalität kapseln, sind sie von den Tests ausgenommen worden.

Testen mit *JUnit*

Alle für das *Software Migration Plug-In* geschriebenen Tests basieren auf dem unter anderem von Kent Beck und Erich Gamma entwickelten *JUnit*-Rahmenwerk [JUnit 2007]. Für die einfacheren Klassen des Plug-Ins, wie zum Beispiel die Segmente oder die Baumstrukturelemente der an mehreren Stellen im Plug-In verwendeten Bäume, reichen die mit diesem Rahmenwerk entwickelten *Unit-Tests* (vgl. [Link 2002], S. 5-6) aus. Vereinzelt verhindern allerdings Abhängigkeiten von *Eclipse*-Komponenten das Testen von Funktionalität.

So sorgt die Benutzung der Klasse `org.eclipse.swt.graphics.Color` in den Klassen des *syntax highlightings* dafür, dass beim Testen Instanzen von `Color` erzeugt und damit native Oberflächenbibliotheken von *Eclipse* gebunden werden. Diese Bibliotheken sind beim Aufruf des Plug-Ins aus der *PDE* in *Eclipse* sichtbar, nicht aber beim Ausführen des Tests. Schließlich ist der Zugriff auf diese Bibliotheken spätestens beim Ausführen der Tests auf einem Integrationsserver problematisch, da dieser in der Regel in einer Kommandozeilenumgebung läuft und keine gesonderte grafische Oberfläche anbietet.

Testen mit *EasyMock*

Ein wesentlicher Teil der im Plug-In enthaltenen Funktionalität enthält komplexe Quelltextoperationen, die durch häufige Aufrufe der durch *Eclipse* bereitgestellten Dokument- und Editorschnittstellen gekennzeichnet sind. Als Beispiel für eine solche Funktionalität sei das Einfügen neuer Segmente in vorhandene Segmente erwähnt. Dazu wird zuerst der Quelltext, in den ein neues Segment eingefügt wird, nach schon vorhandenen Segmenten durchsucht. Befindet sich der selektierte Quelltext innerhalb eines schon vorhandenen Segments, so wird das vorhandene Segment aufgetrennt und in drei neue Segmente unterteilt. Die Positionen der schon vorhandenen Segmentierungskommentare dürfen dabei nicht verändert werden. Speziell für das abschließende dritte Segment wird die Segment-ID am Segmentende durch die ID des neu erstellten, dritten Segments ersetzt.

Eine Möglichkeit, diese komplexen Abläufe zu testen, bietet das für die Benutzung von so genannten *Mock-Objekten* in Tests entwickelte *EasyMock* [EasyMock 2007], das unter *Open Source*-Lizenz erhältlich ist. Die Mock-Objekte erfüllen als Stellvertreterobjekte dabei die in der zu testenden Funktionalität benutzten Schnittstellen der *Eclipse*-Komponenten, von denen diese abhängt. Die Schnittstellen werden dafür zu Testzwecken instrumentalisiert, so dass Aufrufe der Schnittstelle getestet werden können sowie das Rückgabeverhalten im Test kontrolliert werden kann. Mit vorgegebenen Testdokumenten lässt sich das Verhalten der Quelltextoperationen somit sehr detailliert testen. Bei den mit *EasyMock* erweiterten Tests handelt es sich weiterhin um die für die Unit-Tests verwendeten *JUnit-TestCases*.

Ermitteln der Testabdeckung mit *Clover*

Während der Entwicklung wurde die Testabdeckung mithilfe des durch die Firma *Cenqua* vertriebenen *Clover*-Plug-Ins [Clover 2007] ermittelt. *Clover* ermöglicht es anhand der Instrumentalisierung der Plug-In-Klassen, beim Durchlaufen der Tests die Aufrufe in diesen Klassen zu protokollieren. Daraus lässt sich ermitteln, welche Teile der Plug-In-Funktionalität infolge der Tests aufgerufen werden und somit als getestet gelten.

Das Werkzeug wurde vorwiegend zur Identifizierung von Plug-In-Teilen benutzt, in denen die komplexe Oberflächenfunktionalität das vorgezogene Schreiben von Tests erschwert hat bzw. vorhandene Funktionalität aus dem Plug-In-Prototyp übernommen wurde. Dadurch konnten ungetestete Bereiche durch Hinzufügen von Tests mit abgedeckt und qualitativ verbessert werden.

5. Auswertung

Mithilfe der in dieser Arbeit entworfenen Migrationsunterstützung lassen sich Schlüsse hinsichtlich der Umsetzbarkeit und Tauglichkeit des verfolgten Lösungsansatzes gewinnen. Diese Schlüsse ergeben sich vor allem aus dem Vergleich der zu Beginn dieser Arbeit gesteckten und durch die Umsetzung im Plug-In schließlich erreichten Ziele. Im Folgenden sollen die wesentlichen Punkte dieses Vergleichs dargelegt werden. Dabei lassen sich zwei grundsätzliche Fragestellungen unterscheiden:

- Inwieweit wurden die mit dem Plug-In verfolgten Ziele hinsichtlich eines Überblicks über den Stand der Migration erreicht?
- Wie gut ließ sich die entsprechende Unterstützung der Migration in die IDE einbetten?

Diese beiden Punkte werden in den beiden folgenden Kapiteln betrachtet.

5.1. Nutzen der Segmentierung

Ein wesentlicher Grund für die Entwicklung des Plug-Ins ist die Analyse des Migrationsfortschritts im Kontext des *WebMig*-Projekts. Die Arbeit an der Entwicklung des Plug-Ins ermöglicht nun einige Aussagen hinsichtlich Umsetzbarkeit und Informationsgehalt der angestrebten Softwarelösung.

Einbettung der Segmentierungsinformationen

Der direkte Nutzen der Segmentierung ergibt sich für den Benutzer aus der Darstellung segmentierter Bereiche im Quelltext. Diese Darstellung ermöglicht es ihm, während der Arbeit im Quelltext zu erkennen, welche Bereiche des Ursprungssystems hinsichtlich der Migration bisher offen geblieben sind, migriert oder nachträglich verändert wurden. Diese Informationen liegen zum Teil in lesbarer Form als Segmentkommentar mit im Quelltext. Während die Darstellung der Segmentierung im Quelltext über eine Hintergrundfärbung zumindest keine negativen Rückmeldungen seitens der Benutzer ergab, wurde von den Entwicklern des Ursprungsprojekts das übermäßige Aufblähen des *CommSy*-Quelltexts durch die Segmentkommentare bemängelt.

Das führte zu den in Kapitel 4 beschriebenen Anpassungen beim Speichern der Segmentierungsinformationen im Quelltext. Trotzdem lässt sich diesbezüglich festhalten, dass die derzeitige Umsetzung einen Kompromiss darstellt zwischen der angestrebten *Single-Source*-Variante (vgl. Kapitel 4.3.2.) und der Zumutbarkeit zusätzlicher Kommentare in den Quell-texten beider Projekte. Die Optimierung beim Umfang der Segmentierungsinformationen soll die Akzeptanz der Softwarelösung bei den betroffenen Entwicklern erhöhen, wenngleich die zusätzlichen Segmentkommentare in den Quelltexten immer noch als Störung empfunden werden.

Quantitative vs. qualitative Analyse

Die Segmentierung und Zuordnung von Segmenten zueinander erlaubt quantitative Aussagen über den Migrationsfortschritt bzw. einzelne Aspekte davon:

- das Verhältnis von der Gesamtanzahl *LOC*[1] zu den *LOC* in segmentiertem Quelltext.
- der Anteil *LOC* am gesamten Quelltext in schon einander zugeordneten Segmenten zu noch nicht zugeordneten Segmenten. Dies entspricht dem Anteil *LOC* an der Gesamtzahl von migriertem Quelltext zu noch nicht migriertem Quelltext.

[1] LOC, lines of code, Quelltextzeilen.

- die Eingrenzung dieser Zahlen auf relevante *LOC*. Als relevant werden in diesem Zusammenhang die *LOC* gerechnet, bei denen es sich nicht um Leerzeilen und Kommentare handelt, also Teile des Quelltexts, die mit hoher Wahrscheinlichkeit Funktionalität enthalten.
- nachträgliche Änderungen am Ursprungssystem können quantitativ erfasst werden. Die *LOC* in Segmenten, die nachträglich geänderten Ursprungssegmenten zugeordnet sind, kann im Verhältnis zur Gesamtzahl *LOC* betrachtet werden.

Die genannten Kennzahlen sind durchweg quantitativer Natur. Wie sich herausgestellt hat, gestaltet sich die Gewinnung qualitativer Aussagen aus der Segmentierung weit schwieriger als anfangs angenommen. Tatsächlich ist die qualitative Aussage darüber, ob und wie weit ein bestimmter Teil der Ursprungsfunktionalität ins Zielsystem migriert wurde, nur schätzungsweise möglich. Die aus der Segmentierung und Zuordnung gewonnenen Zahlen über migrierte *LOC* geben wenig Auskunft darüber, inwieweit eine Funktionalität tatsächlich migriert wurde. Dazu schätzt der Benutzer ab, in welchem Umfang die Zielsegmente die Funktionalität im Ursprungsprojekt implementieren. Schwieriger wird es zudem, wenn eine bestimmte Komponente hinsichtlich des Migrationsfortschritts bewertet werden soll, die erst zu einem gewissen Teil ins Zielsystem überführt wurde. Das erfordert eine Wichtung einzelner Segmente.

Ein Beispiel dafür stelle eine Komponente dar, die die Benutzerauthentifizierung im Ursprungs- und Zielsystem übernimmt. Die quantitative Auswertung der Migration kann dabei durchaus ergeben, dass die Funktionalität beinahe vollständig segmentiert und ins Zielsystem übertragen wurde. Enthält der verbleibende, noch nicht migrierte Quelltext allerdings den Zugriff auf einen Authentifizierungsserver, fehlt der Implementierung im Zielsystem eine wichtige Authentifizierungskomponente. Das relativiert die Aussage über die fast vollständige Migration wieder. Die Verbindung von quantitativer und qualitativer Aussagekraft der Migrationsanalyse beruht damit auf Einschätzungen des jeweiligen Benutzers, der die Segmentierung und Zuordnung vorgenommen hat. Erst eine vollständige Segmentierung sowie Zuordnung von Projektteilen – also eine hundertprozentige Abdeckung - vereinfacht die Analyse bezüglich qualitativer Fragestellungen.

Hinsichtlich der Aussagekraft der Auswertung spielt das Vorgehen bei der Segmentierung und Zuordnung eine wichtige Rolle. Wird im Quelltext entlang des Kontrollflusses in einer Methode segmentiert, müssen Verzweigungen in andere Methoden derselben Klasse oder sogar in andere Klassen (und somit Quelltextdateien) berücksichtigt werden. Einzelne Methoden können vollständig ins Zielsystem überführt sein, während sich ein großer Anteil noch nicht migrierter Funktionalität in den referenzierten Klassen befindet. Qualitative Aussagen über den Fortschritt der Migration werden dadurch erschwert. Wird bei Verzweigung des Kontrollflusses systematisch in der aufgerufenen Methode weitersegmentiert, kann dieser Schwachpunkt der Analyse ausgeschaltet werden.

Erleichterung der Migrationsanalyse

Ein weiteres wichtiges Ziel der Migrationsunterstützung war es, dem Benutzer die Analyse über Stand und Fortschritt der Migration zu erleichtern. Leider gestaltete sich die Segmentierung der Quelltexte beider Projekte trotz teilweiser Automatisierung immer noch zeit- und arbeitsintensiver als anfangs angenommen. Das ist nicht zuletzt auf die Tatsache zurückzuführen, dass trotz der vereinfachten Segmentierungsarbeit im Quelltext gleichartige Funktionalität in beiden Projekten nur durch einen systemkundigen Entwickler identifiziert werden kann oder zumindest sehr viel Einarbeitung notwendig ist. Zudem haben beide Projekte inzwischen einen Umfang erreicht, der das Segmentieren allein schon wegen der Menge an zu segmentierendem Quelltext zu einem langwierigen Prozess macht.

Die vorliegende Version des Plug-Ins bietet keine Funktionalität zur automatisierten Analyse der Migration an, über die aus Projekten zum Beispiel im Zuge eines automatisierten Release-Prozesses auf einem Integrationsserver Migrationsinformationen gewonnen werden könnten. Die Analyse erfolgt aktiv durch den Benutzer.

5.2. Einbettung des Plug-Ins

Die *Eclipse*-IDE bietet durch die Plug-In-Architektur die Möglichkeit, die Entwicklungsumgebung beliebig zu erweitern und zu gestalten (vgl. Kapitel 4.3.1.). Bei der Entwicklung einer Erweiterung (in Form eines Plug-Ins) bietet *Eclipse* verschiedene Möglichkeiten der Einbindung in die vorhandenen Strukturen und die vorhandene Oberfläche. Je allgemeiner die für eine einzubindende Funktionalität gewählte Schnittstelle (=*extension point*) ist, desto unabhängiger ist in der Regel das Plug-In von der Version von *Eclipse*, in der es eingebunden wird. Die konsequente Erweiterung von Schnittstellen (vgl. [Rivières 2006]) sowie die symbolbasierte Registrierung und Verwaltung von Plug-Ins ermöglicht eine langfristige Abwärtskompatibilität neuerer *Eclipse*-Versionen zu älteren Plug-Ins.

Wie in Kapitel 4.3. beschrieben, wurden für die Migrationsunterstützung viele Funktionen im Kontext von Ressourcen und vom Quelltext in unterschiedlichen Editoren eingebunden. Die Zuordnung einer Kontextfunktion erfolgt dabei zu einem allgemeinen Editorkontext oder zu einer speziellen Editorklasse, je nachdem, in welchen Editoren der IDE die Funktionalität angeboten wird. In der ersten Version des Plug-Ins wurden die Quelltextfunktionen am allgemeinen Kontext für Texteditoren registriert, in der Annahme, die Kontextfunktionen in allen Editoren anbieten zu können, mindestens aber in denjenigen für *Java*-, *JSP*- und *PHP*-Quelltext.

Diese allgemeine Registrierung reichte allerdings nicht aus, damit die Funktionalität aus den gewünschten Editoren heraus aufgerufen werden konnte. Daher wurde jede Kontextaktion jeweils am *TextEditorContext* (für den *JSP*-Editor sowie den Plug-In-eigenen *SMP*-Editor), am *CompilationUnitEditorContext* (für den *Java*-Editor) und schließlich für die *PHP*-Quelltexte am Kontext der Oberklasse des in der *PHP IDE* verwendeten Editors (org.eclipse.wst.sse.ui.StructuredTextEditor.EditorContext) registriert.

Die Unterstützung des *PHP*-Editors durch das Plug-In erzeugt nun eine Abhängigkeit des Plug-Ins von den verwendeten Werkzeugen zur Web-Entwicklung. Diese finden sich allerdings nicht in der Standardinstallation von *Eclipse* und müssen separat installiert werden (siehe *Web Tools Project*, [Eclipse 2007]). Während das Fehlen dieser Plug-Ins noch keine Auswirkungen auf die Lauffähigkeit der Migrationsunterstützung hat (wird der geforderte Kontext nicht vorgefunden, so wird keine Funktionalität für die nicht vorhandenen Editoren registriert), so ist die Bindung im Falle des *PHP*-Editors sehr spezifisch. Daher ist die Wahrscheinlichkeit hoch, dass zukünftige Versionen der Web-Werkzeuge die mitgelieferten Editoren für einen allgemeineren Kontext (zum Beispiel den *TextEditorContext*) registrieren. Die Unterstützung von *PHP*-Editoren durch das *Software Migration Plug-In* bleibt somit auf Versionen beschränkt, die den *PHP*-Editor im besagten Kontext anbieten.

6. Zusammenfassung und Ausblick

6.1. Zusammenfassung

Die Zielsetzung dieser Arbeit bestand im Entwurf einer als Plug-In realisierten Unterstützung der Migration im *WegMig*-Projekt. Der im *WebMig*-Projekt verfolgte Ansatz der schrittweisen Migration von Funktionalität anhand des beiden Projekten zugrundeliegenden Quelltexts sollte dafür entsprechend instrumentalisiert und erweitert werden.

Zuerst wurde dafür der Kontext ermittelt, in den die Migrationsarbeit eingebettet ist. Zu diesem Kontext gehören die Arbeitsumgebung und die Aufgaben der an den beiden Projekten *CommSy* und *JCommSy* beteiligten Entwickler. Auf der Grundlage der beschriebenen Vorgehensweise bei der Migration und des Arbeitskontexts wurden dann erste Konzepte und Ideen entwickelt, wie sich die Migration am Quelltext der Projekte festmachen und nutzen lässt, um die geforderte Unterstützung bei der Migration bewerkstelligen zu können. Diese Konzepte wurden im Laufe der ersten Entwicklungsschritte des Plug-Ins weiter konkretisiert und ausgebaut.

Diese Entwicklungsschritte führten zu einem Prototyp, der einen ersten Vorschlag zur Realisierung der Konzepte darstellte. Mithilfe des Prototyps und der daraus entstandenen weiterführenden Anforderungen konnte dann der Arbeitsablauf für die Analyse des Migrationsfortschritts beschrieben werden. Der Arbeitsablauf wurde in einzelne Arbeitsschritte unterteilt, um detailliert jeden Arbeitsschritt in Form eines Szenarios einzufassen. Die Szenarios wiederum dienten als Grundlage für die Umsetzung der jeweils für einen Schritt benötigten Funktionalität und lieferten Anhaltspunkte für eine geeignete Einbindung in die Oberfläche der Entwicklungsumgebung.

Um den Benutzern des Plug-Ins ein frühes Testen und Rückmeldungen zu den bereits umgesetzten Plug-In-Funktionen zu ermöglichen, wurden regelmäßig neue Versionen des Plug-Ins bereitgestellt. Dadurch konnte das Plug-In nach den Wünschen der Benutzer angepasst und weiterentwickelt werden. Während der Entwicklung ergaben sich hinsichtlich der zu unterstützten Arbeit im *WebMig*-Projekt unterschiedliche Problem- und Fragestellungen, die zu Beginn der Arbeit nicht bedacht worden waren. So stellte sich das Einbringen der Segmentierungsinformationen in den Quelltext als eine Hürde für die Akzeptanz des Plug-Ins heraus, da die Lesbarkeit und Übersichtlichkeit des Quelltexts durch die zusätzlichen Informationen stärker als anfangs angenommen eingeschränkt wurde. Zudem erwies sich das Segmentieren selbst als arbeitsintensiv. Des Weiteren blieb die erhoffte qualitative Aussagekraft der gewonnenen Migrationsinformationen hinter den Erwartungen zurück, da die Umsetzung des Plug-Ins Schwächen beim Analysieren übergeordneter Systemkomponenten offenbarte.

Die Programmierung des Plug-Ins erfolgte mit hohen Ansprüchen an die Qualität. Um diesen Ansprüchen gerecht zu werden, wurde die entwickelte Funktionalität mit geeigneten Tests abgedeckt, um vor allem die schon umgesetzten Funktionen gegen die stetigen Anpassungen während der Weiterentwicklung zu stabilisieren. Die Anbindung an die Oberfläche und die angebundene Funktionalität wurden klar von einander getrennt. Für die funktionalen Anteile des Plug-Ins wurden durchgehend jeweils separate Schnittstellen definiert und darauf aufbauend die konkrete Implementierung realisiert, um die Testbarkeit und Wartbarkeit zu verbessern und eine mögliche spätere Erweiterung des Plug-Ins zu erleichtern.

Die Auswertung der während der Entwicklung und bisherigen Nutzung des Plug-Ins gesammelten Erfahrungen lieferte ein eher ernüchterndes Ergebnis. Aus der Segmentierung lässt sich eine Vielzahl quantitativer Aussagen in Bezug auf die Migration gewinnen. So hat der Benutzer mithilfe der Migrationsanalyse einen Überblick über die prozentuale Abdeckung der Quelltexte mit Segmenten sowie der jeweiligen Anteile von Segmenten unterschiedlichen Zustands an der Gesamtsegmentierung. Die wesentlichen Fragen bei der Migration, nämlich ob und in welchem Umfang Funktionalität aus dem Ursprungssystem schon im Zielsystem abgebildet wurde, lassen sich aus den gewonnenen Zahlen allerdings nur unter Vorbehalt beantworten. Die qualitative Aussagekraft der Segmentierung hängt vor allem vom Vorgehen beim Segmentieren ab. Außerdem muss der Benutzer bei jeder Zuordnung von Neuem entscheiden, ob eine Implementierung im Zielsystem die ursprüngliche Funktionalität korrekt

widerspiegelt. Die Tauglichkeit des Plug-Ins als ein die Migration unterstützendes Werkzeug lässt sich unter diesen Gesichtpunkten nur noch eingeschränkt bestätigen.

Trotz der umfangreichen Entwicklungstätigkeit am Plug-In konnten einige Anforderungen im Zuge dieser Arbeit nicht umgesetzt werden. Die automatisierte Vorsegmentierung ganzer Quelltextdateien war eine frühe Anforderung an das Plug-In. Während der Entwicklung stellte sich allerdings heraus, dass der Zugriff auf einen wesentlichen Teil der dazu nötigen Informationen nicht realisiert werden konnte. Diese Anforderung konnte daher nur prototypisch für den Teilbereich der *Java*-Quelltexte erfüllt werden. Eine weitere Anforderung betraf das Ausblenden der Segmentierungs-informationen im Quelltext. Die Umsetzung gestaltete sich dann aber weit komplexer als anfänglich vermutet. Da zudem die von der Erweiterung des Quelltexts am ärgsten betroffenen Entwickler des Ursprungsprojekts nicht unbedingt zu *Eclipse*-Nutzern zählten und somit eine Umsetzung im Plug-In einen fraglichen Nutzen gehabt hätte, wurde in dieser Arbeit darauf verzichtet. Zum Ausgleich wurde an einer Verringerung des Umfangs an im Quelltext abgelegten Segmentierungsinformationen gearbeitet. Diese Lösung erschien auch praktikabler zu sein, da der Quelltext dadurch nun allgemein durch die Segmente weniger an Umfang zunahm. Schließlich wurden noch einige Anforderungen bei der Bereitstellung der Analyseergebnisse zurückgestellt. So sollte ursprünglich in der Auswertung ersichtlich sein, welche Segmente in Ursprungs- und Zielsystem einander zugeordnet sind. Stattdessen wurde nur eine Möglichkeit eingebunden, von in der Auswertung aufgeführten Segmenten in den zugehörigen Quelltextabschnitt bzw. im Quelltext selbst von einem Zielsegment in das zugeordnete Ursprungssegment zu springen.

6.2. Ausblick

Die Segmentierung des Quelltexts von Ursprungs- und Zielsystem erlaubt die Zuordnung von atomaren Funktionalitätseinheiten zueinander. Die Obergrenze für die Größe eines Segments ist durch die Größe einer zusammenhängenden Quelltextdatei gegeben. In der Regel entspricht der in einer solchen Datei enthaltene Quelltext einer dargestellten Seite (im *PHP*-Ursprungssystem) beziehungsweise einer Klasse (im *Java*-Zielsystem). Übergeordnet zu den einzelnen einander zugeordneten Segmenten im Quelltext ist der Fortschritt der Migration größerer Systemteile von Interesse. Diese Systemteile enthalten Funktionalität, die über mehrere Quelltextdateien verstreut liegt, zum Beispiel als Klassen innerhalb eines Moduls[1].

Die Zuordnung der einzelnen Segmente eines Moduls zu den Segmenten des dazugehörigen Zielmoduls ist möglich, aber die Auswertung, inwieweit ein Modul migriert wurde, kann nur manuell erfolgen. Dazu betrachtet der Entwickler die Segmentierung von Ursprungs- und Zielmodul und wertet die Zuordnungen aus. Im Plug-In gibt es keine spezielle Unterstützung für diese Aufgabe. Daher gab es den Vorschlag, eine Art *Meta-Segmentierung* zu erlauben, mit deren Hilfe größere Einheiten von Segmenten gebildet werden können, die über die Grenze einzelner Quelltextdateien hinausgehen.

Diese Meta-Segmente könnten in die Auswertung der Segmentierung mit aufgenommen werden, um Aussagen über den Migrationsfortschritt von Moduln zu erlauben. Eine Möglichkeit zur Erstellung solcher Meta-Segmente stellt ein Werkzeug dar, mit dem vorhandene atomare Segmente zusammengefasst und mit einem symbolischen Namen versehen werden können. Im Unterschied allerdings zur Auswertung der atomaren Segmentierung von Ursprungs- und Zielsystem enthält ein Meta-Segment kein Wissen über nicht segmentierten Quelltext, da nur vorhandene Segmente (zugeordnete, nicht zugeordnete und obsolete Segmente) in einem Meta-Segment zusammengefasst werden können.

Denkbar wäre eine übergeordnete Segmentierung des Quelltexts, um Aussagen über nicht segmentierten Modul-Quelltext zu erhalten. Dazu müsste eine weitere Form von Segmenten eingeführt werden, die teilweise oder ganze Quelltextdateien in Segmente einfasst. Diese Segmente haben eine grundlegend andere Semantik als die in dieser Arbeit verwendeten Funktionalitätssegmente, da sie a) selber solche Segmente enthalten können, b) dementsprechend unsegmentierten Quelltext enthalten (also Quelltext, der nicht in einem

[1] Ein Modul ist eine Sammlung von Algorithmen und Datenstrukturen, die einer bestimmten, abgeschlossenen Aufgabe zugeordnet sind (vgl. [Pomberger 2002], S. 523).

atomaren Segment eingefasst ist) und c) ausschließlich der Zusammenfassung und Auswertung durch Meta-Segmente dienen. Eine Änderungsverfolgung würde dabei nur auf Änderungen in atomaren Segmenten beruhen, nicht aber auf Änderungen im unsegmentierten Quelltext, da diese in den durch Meta-Segmente eingefassten Quelltext ebenso als *nicht migriert* gelten.

Ein weiterer Vorschlag betrifft die Erweiterung des jetzigen Segmentmodells. Um die qualitative Aussagekraft der Segmentierung zu erhöhen, könnten die Verbindungen zwischen den Segmenten sowie die Segmente selbst mit einer Art Wichtungsfunktion versehen werden. Diese Wichtung der einzelnen Elemente wäre eine Möglichkeit, Segmente und Verbindungen hinsichtlich ihrer Relevanz für die Migration einer spezifischen Funktionalität einzustufen. Diese Erweiterung adressiert das in der Auswertung in Kapitel 5 beschriebene Problem der Falscheinschätzung des Migrationsfortschritts, wenn für die Fortschrittsbetrachtung allein die ermittelten LOC der Analysefunktion herangezogen werden. Essenzielle Teile der Ursprungsfunktionalität könnten selbst bei einer fast vollständigen Migration immer noch im Zielsystem fehlen. Die Wichtung solcher Teile mit einer hohen Relevanz für die zu migrierende Funktionalität könnte die Analyse korrigieren. Ein alternativer Ansatz beschreibt die Idee, Verbindungen zwischen Segmenten geeignet annotieren zu können, um für die Migration wichtige Informationen in menschenlesbarer Form an den Verbindungen zu deponieren, die eine qualitative Betrachtung der Analyse unterstützen.

Anhang A: Installation des Plug-Ins

Alle für die Installation des Plug-Ins (Version 1.3.0) erforderlichen Dateien (zusammen mit Quelltext und *Eclipse*-Projektdateien) befinden sich auf der dieser Arbeit beiliegenden CD-Rom. Ältere Versionen des Plug-Ins werden auf der zum Projekt gehörenden Internetseite unter http://smp.dinamixx.info zum Herunterladen angeboten.

Das Plug-In wird manuell in die *Eclipse*-Umgebung installiert. Dazu wird das Archiv org.eclipse.contribution.smp_1.3.0.zip dekomprimiert und das enthaltene Verzeichnis in das plugin-Verzeichnis von *Eclipse* kopiert. Um Überschneidungen mit ggf. schon vorhandenen, älteren Versionen des Plug-Ins zu vermeiden, sind vorher alle Versionen bzw. alle Verzeichnisse des Plug-Ins aus dem plugin-Verzeichnis zu entfernen.

Beim Start der Entwicklungsumgebung wird das *Software Migration Plug-In* geladen und in die Umgebung integriert. Die korrekte Einbindung des *Plug-Ins* kann in der Plug-In-Registratur von *Eclipse* eingesehen werden (Menü Help → About Eclipse SDK → Plug-In Details). Dort findet sich bei korrekter Installation ein Eintrag für das „Software Migration Plug-In" gefolgt von der Versionsnummer (1.3.0).

Um neue Migrationsprojekte erstellen zu können, muss bei der Erstinstallation die entsprechende Ressource in *Eclipse* für die Erzeugung aus dem New-Menü aktiviert werden. Die Abb. A.1. zeigt den dazu gehörenden Konfigurationsdialog von *Eclipse* (Version 3.2.1).

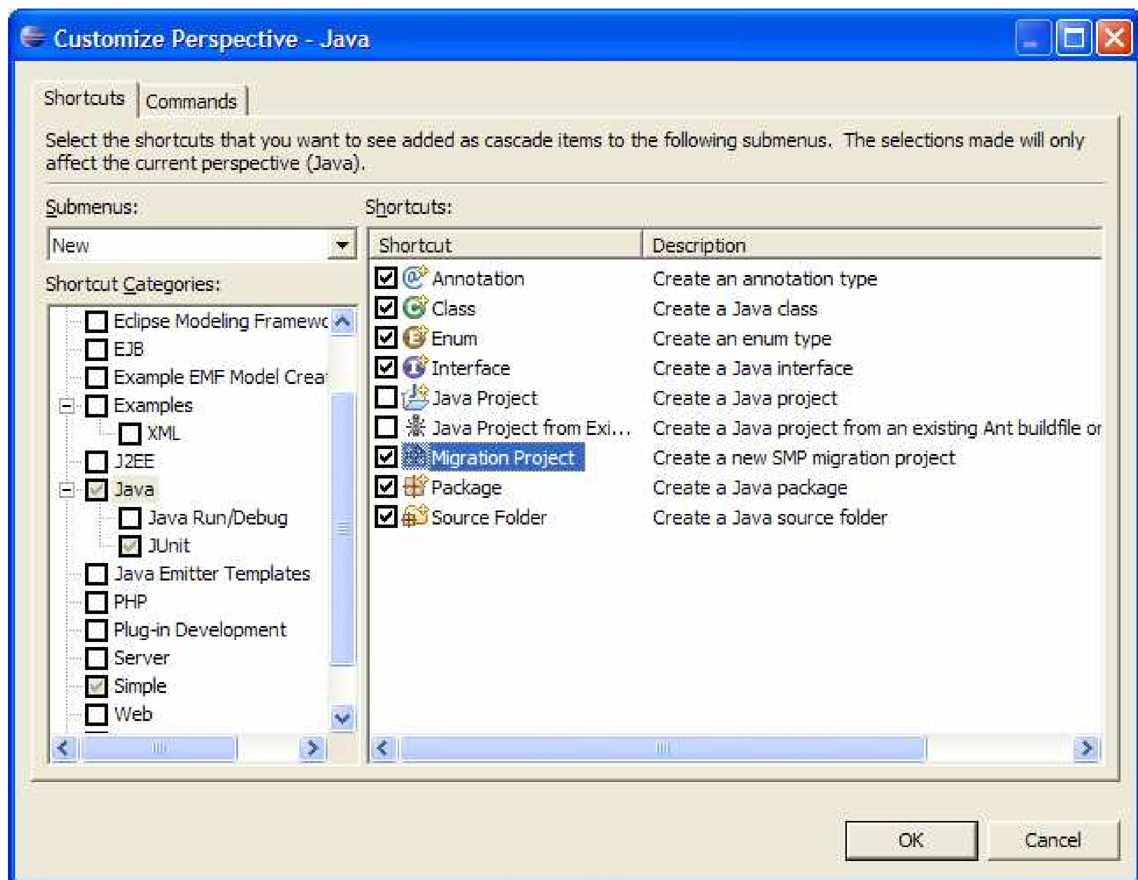


Abb. A.1.: Konfigurationsdialog von *Eclipse* zum Aktivieren von Menüeinträgen

Dieser Dialog wird über das *Eclipse*-Hauptmenü geöffnet (Menü Window → Customize Perspective…). Dort in den Untermenüs New auswählen und den *Java*-Knoten auswählen. Zur Aktivierung des *Migration Projects* in der Auflistung beim entsprechenden Eintrag den Haken setzen. Bei korrekter Aktivierung erscheint das *Migration Project* dann als Ressource im New-Menü von *Eclipse*.

Anhang B: Benutzung des Plug-Ins

In dieser Übersicht werden die im Plug-In zur Verfügung gestellten Funktionen und ihr Aufruf beschrieben.

Neues Migrationsprojekt erstellen

Im Dateimenü von *Eclipse* das Erstellen einer neuen Ressource aufrufen. Dort als Ressourcentyp *»Migration Project«* auswählen. Es öffnet sich ein Erstellungsdialog, der nacheinander das Ursprungs- und Zielprojekt der Migration abfragt. Nach Beenden des Dialogs wird in den ausgewählten Projekten jeweils eine Migrationsprojektdatei erzeugt, die die beiden Projekte einander zuordnet.

Segment erstellen

Um im Quelltext einen Abschnitt in ein Segment einzufassen, den entsprechenden Abschnitt im Editor selektieren und dann auf der Selektion das Kontextmenü öffnen. Im Kontextmenü die Funktion *»Add segment«* aufrufen. Der gewählte Quelltextabschnitt wird dann in Segmentkommentare eingefasst.

Befindet sich die Auswahl innerhalb eines schon vorhandenen Segments, wird dieses in drei neue Segmente aufgeteilt. Das erste Segment enthält dann den Bereich des ursprünglichen Segments vor der Selektion, das zweite Segment dann die Selektion selbst und das dritte Segment den verbleibenden Rest des ursprünglichen Segments.

Wird die Funktion auf Selektionen in PHP-Quelltextdateien aufgerufen, wird im selektierten Abschnitt nach Funktionssignaturen gesucht. Wird eine gefunden, wird die Signatur als Namensvorschlag in den Namen des Segments kopiert.

Segment zuordnen

Zum Zuordnen eines Segments im Zielprojekt zu einem Segment im Ursprungsprojekt im Quelltext des Zielsegments das Kontextmenü öffnen und dort die Funktion *»Implementation of…«* aufrufen. Daraufhin öffnet sich ein Zuordnungsdialog, in dem das Ursprungsprojekt in einer Baumansicht dargestellt wird. In dieser Baumansicht die Quelltextdatei suchen, in der sich das zuzuordnende Ursprungssegment befindet. Das entsprechende Segment auswählen und den Dialog beenden, um die Zuordnung abzuschließen. Die Zuordnungsinformationen werden am Zielsegment vermerkt und erscheinen im Quelltext.

Segment als obsolet markieren

Auf dem zu markierenden Segment im Quelltext das Kontextmenü öffnen und die Funktion *»Mark Segment as obsolete«* auswählen. Am Segment wird daraufhin das Schlüsselwort `<obsolete>` als Name gesetzt und das Segment als obsolet markiert.

Starten einer Migrationsanalyse

Im Package Explorer auf einem Projekt das Kontextmenü öffnen und die Funktion *»Analyse Migration Project«* aufrufen. Gehört das Projekt zu einem Migrationsprojekt, wird auf den beiden zur Migration gehörenden Projekten die Analyse der Segmentierungsabdeckung und –zustände gestartet. Die Ergebnisse dieser Analyse werden in einem *View* mit der Bezeichnung *Migration Status* angezeigt.

Für beide Projekte gibt es in der Auswertung jeweils eine baumartige Projektübersicht, ähnlich der Übersicht im Package Explorer oder Navigator, in der die Ergebnisse zusammengefasst sind. In dieser Baumansicht kann das Projekt bis hinunter zu einzelnen Segmenten durchlaufen werden. Ein Doppelklick auf ein Segment öffnet die dazugehörende Quelltextdatei und den Abschnitt im Editor.

Validieren eines Segments

Um ein Segment, dessen Ursprungssegment nachträglich verändert wurde, auf die Änderungen anzupassen und zu validieren, im zugehörigen Quelltextabschnitt das Kontextmenü öffnen und die Funktion *»Update source fingerprint«* aufrufen. Die aus dem Ursprungssegment berechnete

Prüfsumme wird dann neu berechnet und in das Zielsegment übernommen, das dann wieder als *migriert* gilt.

Aus einem Segment zum zugeordneten Ursprungssegment springen

Auf dem Segment im Quelltext das Kontextmenü öffnen und die Funktion *»Jump to…«* aufrufen. Ist dem Segment ein Ursprungssegment zugeordnet, werden zugehörige Quelltextdatei und –abschnitt im Editor geöffnet und angezeigt.

Automatisches Segmentieren einer Java-Datei

Zum automatischen Segmentieren einer *Java*-Datei im Package Explorer auf der entsprechenden Quelltextdatei das Kontextmenü aufrufen und die Funktion *»Segment source file«* auswählen. Die Methoden der in der Datei befindlichen Klasse werden dann automatisch in Segmente eingefasst.

Anhang C: Kommentarmarkenreferenz

Die folgende Tabelle enthält die Bezeichner der vom *Plug-In* verwendeten Segmentkommentare und eine Beschreibung der mit ihnen gespeicherten Informationen. Die Bezeichner und der Inhalt der Kommentare entsprechen der ersten Fassung des *Plug-Ins*, die im Zuge dieser Diplomarbeit entwickelt wurde und können sich in zukünftigen Versionen ändern. Die an der Migration beteiligten Projekte werden in dieser Aufzählung als Ursprungs- und Zielprojekt, die entsprechenden Segmente Ursprungs- und Zielsegmente genannt.

In den jeweiligen Kommentarzeilen befindet sich jeweils eine Kommentarmarke mit einem vorangestellten @ (angelehnt an die *JavaDoc*-Schreibweise), gefolgt von der darunter gespeicherten Information.

Kommentarmarke	Beschreibung
segment-begin	Leitet das Segment und den dazugehörigen Informationsblock ein. Im begin-Kommentar werden die fünfstellige Segment-ID und ein optionaler Name des Segments gespeichert. Obsolete Segmente werden im begin-Kommentar mithilfe des reservierten Schlüsselworts <obsolete> anstelle des Segmentnamens markiert.
segment-comment	Ein optionaler Freitext-Kommentar, der mit dem Segment gespeichert werden kann.
segment-end	Schließt das Segment ab. Um das Segmentende eindeutig einem Segment zuordnen zu können, wird die Segment-ID zusätzlich im end-tag gespeichert.
segment-sourcefile	Enthält den Pfad zur Quelltextdatei im Ursprungsprojekt, in der sich das dem Segment zugeordnete Ursprungssegment befindet. Dieser Eintrag ist nur in zugeordneten Segmenten (im Zielprojekt) enthalten.
segment-sourcesegmentid	Enthält die Segment-ID des dem Segment zugeordneten Ursprungssegments. Dieser Eintrag ist nur in zugeordneten Segmenten (im Zielprojekt) enthalten.
segment-sourcefingerprint	Enthält die aus dem Quelltext des zugeordneten Ursprungssegments berechnete Prüfsumme, mit deren Hilfe Änderungen am Ursprungsquelltext erkannt werden können. Dieser Eintrag ist nur in zugeordneten Segmenten (im Zielprojekt) enthalten.

Tab. C.1.: Übersicht über die Segmentkommentare

Literaturverzeichnis

[Beck 2003] BECK, Kent ; GAMMA, Erich: *Contributing to eclipse – Principles, Patterns und Plug-Ins*. 1. Aufl. Boston [u.a.] : Addison-Wesley, 2003

[Brück 1993] BRÜCK, Rainer: Migration: a model for design by modification. In: *Proceedings of the 1993 ACM/SIGAPP symposium on Applied computing: states of the art and practice* (1993), S. 71-76

[Brüssau 2006] BRÜSSAU, Kai (Hrsg.) ; WIDDER, Oliver (Hrsg.); BRÜCKNER, Herbert ; LIPPERT, Martin ; LÜBKEN, Matthias ; SCHWARTZ-REINKEN, Birgit ; WUNDERLICH, Lars: *Eclipse – Die Plattform – Enterprise-Java mit Eclipse 3.1*. 2. Aufl. Frankfurt : entwickler.press, 2006

[Clover 2007] *Clover*. URL http://www.cenqua.com – Aktualisierungsdatum: 17.07.2007 – Cenqua Pty Ltd. Level 1 - Sydney, Australia

[Cockburn 2005] COCKBURN, Alistair: *Agile Software Development*. 8. Aufl. Boston [u.a.] : Addison-Wesley, 2005.

[Commsy 2007] *CommSy*. URL http://www.commsy.net - Aktualisierungsdatum: 17.07.2007 - HiTec e.V., Universität Hamburg, Fachbereich Informatik

[Cordy 2006] CORDY, James R.: Source transformation, analysis and generation in TXL. In: *PEPM '06: Proceedings of the 2006 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation* (2006), S. 1-11

[EasyMock 2007] *EasyMock*. URL http://www.easymock.org – Aktualisierungsdatum: 17.07.2007 – Tammo Freese, 2001-2007

[Eclipse 2007] ECLIPSE FOUNDATION: *Eclipse – SDK*. URL http://www.eclipse.org, Aktualisierungsdatum: 07.07.2007 – Eclipse Foundation, Inc., Ottawa, Ontario, Canada.

[Fowler 2004] FOWLER, Martin: *Inversion of Control Containers and the Dependency Injection pattern*. URL http://martinfowler.com/articles/injection.html - Aktualisierungsdatum: 23.01.2004 - ThoughtWorks Inc. - Chicago, US

[Gamma 2004] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software*. 1. Aufl., [Neuaufl.] - Boston [u.a.] : Addison-Wesley, 2004

[JUnit 2007] *JUnit*. URL http://www.junit.org – Aktualisierungsdatum: 17.07.2007 – Object Mentor, Incorporated, 2001-2004

[Koponen 2005] KOPONEN, Timo ; HOTTI, Virpi: Open source software maintenance process framework. In: *Proceedings of the fifth workshop on Open source software engineering* (2005), S. 1-5

[Krause 2001] KRAUSE, Jörg: *PHP 4 – Grundlagen und Profiwissen. Webserver-Programmierung unter Windows und Linux*. 2. überarb. Aufl. – München : Hanser-Verlag, 2001.

[Kremers 2000] KREMERS, Mark ; VAN DISSEL, Han: Enterprise resource planning: ERP system migrations. In: *Commun. ACM* 43 (2000), Nr. 4, S. 53-56

[Link 2002] LINK, Johannes: *Unit Tests mit Java – Der Test-First-Ansatz.* 1.Aufl. Heidelberg : dpunkt.verlag, 2002.

[Lippert 2002] LIPPERT, Martin; ROOCK, Stefan; WOLF, Henning: *Software entwickeln mit eXtreme Programming*. 1.Aufl. Heidelberg : dpunkt.verlag, 2002.

[McAffer 2005] MCAFFER, Jeff ; LEMIEUX, Jean-Michel: *Eclipse – Rich Client Platform*. 1. Aufl. Boston [u.a.] : Addison-Wesley, 2005.

[Menezes 1997] MENEZES, Alfred ; VAN OORSCHOT, Paul ; VANSTONE, Scott: *Handbook of Applied Cryptography*. Boca Raton [u.a.] : CRC Press, 1997.

[Oppermann 1992] OPPERMANN, Reinhard: *Software-ergonomische Evaluation.* 2. Aufl. Berlin [u.a.] : De Gruyter Verlag, 1992

[PHP 2007] *PHP: Hypertext Preprocessor*. URL http://www.php.net - Aktualisierungsdatum: 07.07.2007 – The PHP Group.

[Pomberger 2002] POMBERGER, Gustav (Hrsg.) ; RECHENBERG, Peter (Hrsg.) et al.: *Informatik-Handbuch.* 3. Aufl. Wien : Carl Hanser Verlag, 2002

[Rivières 2006] DES RIVIÈRES, Jim: *Evolving Java-based APIs*. URL http://wiki.eclipse.org/Evolving_Java-based_APIs - Aktualisierungsdatum: 07.07.2007 - IBM Corporation

[Sebesta 2006] SEBESTA, Robert W.: *Concepts of Programming Languages*. 7. Aufl. Boston [u.a.] : Pearson/Addison Wesley, 2006

[Sun 2007] SUN MICROSYSTEMS: *Java*. URL http://java.sun.com – 1994-2007 – Aktualisierungsdatum: 07.07.2007 - Sun Microsystems, Inc. - Santa Clara, US

[Ullenboom 2002] ULLENBOOM, Christian: *Java ist auch eine Insel*. 2. Aufl. Bonn : Galileo Press GmbH, 2002

[Züllighoven 1998] ZÜLLIGHOVEN, Heinz: *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug- & Material-Ansatz.* 1. Aufl. Heidelberg : dpunkt-Verlag, 1998

Eidesstattliche Erklärung

Ich versichere, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht.

Fabian Dittberner

Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

Fabian Dittberner