Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Hauke Wittern

Entwicklung eines Assistenten für die modellgetriebene Softwareentwicklung

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Hauke Wittern

Entwicklung eines Assistenten für die modellgetriebene Softwareentwicklung

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Olaf Zukunft
Zweitgutachter: Prof. Dr. Jörg Raasch

Abgegeben am 20. Juni 2008

Hauke Wittern

Thema der Bachelorarbeit
Entwicklung eines Assistenten für die modellgetriebene Softwareentwicklung
Stichworte
Modellgetriebene Softwareentwicklung, MDSD, Model Driven Architecture, MDA, Assistenz, Wizard, Modellierung, Usability-Test
Kurzzusammenfassung
Diese Bachelorarbeit stellt einen Prototyp eines Assistenten vor, der Softwareentwickler bei der modellgetriebenen Softwareentwicklung unterstützt. Zunächst untersucht ein Usability-Test, einer für die modellgetriebene Softwareentwicklung typischen Werkzeugkombination, wo die Entwickler weitere Unterstützung bei der Modellierung benötigen. Anschließend wird ein Assistent vorgestellt, der den Entwicklern diese Unterstützung bietet. Der hier entwickelte Assistent ist ein Wizard, der den Entwicklern die Verwendung von Metamodellen erleichtert. Abschließend untersucht ein zweiter Usability-Test den Prototyp des Assistenten. Dieser zweite Test untersucht, ob der Prototyp den Entwicklern die Modellierung gegenüber dem ersten Test erleichtern konnte. Die Ergebnisse werden diskutiert und Empfehlungen für die Weiterentwicklung von Assistenten für die modellgetriebene Softwareentwicklung gegeben.

Hauke Wittern

Title of the paper
Development of an assistant for Model Driven Software Development
Keywords
Model Driven Software Development, MDSD, Model Driven Architecture, MDA, Assistance, Wizard, Modeling, Usability-Test
Abstract
This bachelor thesis introduces a prototype of an assistant, which assists software developers with Model-Driven Software Development (MDSD). First, a usability-test examines where developers need further assistance when they are using a typical combination of tools for MDSD. The afterwards introduced assistant tries to give that assistance to the developers. The introduced assistant is a wizard helping developers with using a metamodel. Concluding a second usability-test examines whether the prototype was abled to make modeling easier for the developers than in the first test. This bachelor thesis discusses the results and gives recommendations how assistants for Model-Driven Software Development can be improved in the future.

Inhaltsverzeichnis

1. Einleitung **2**
- 1.1. Vision und Zielsetzung . . . 3
- 1.2. Inhaltlicher Aufbau der Arbeit . . . 3

2. Grundlagen **5**
- 2.1. Modellgetriebene Softwareentwicklung . . . 5
  - 2.1.1. Definition . . . 5
  - 2.1.2. Modellierung . . . 6
  - 2.1.3. Model Driven Architecture (MDA) . . . 9
  - 2.1.4. Bewertung der modellgetriebenen Softwareentwicklung . . . 11
  - 2.1.5. Abgrenzung und Alternative Ansätze . . . 13
  - 2.1.6. Fazit . . . 14
- 2.2. Assistenten und Wizards . . . 15
  - 2.2.1. Motivation für Assistenz in Softwaresystemen . . . 15
  - 2.2.2. Merkmale von Assistenz-Systemen . . . 15
  - 2.2.3. Techniken von Assistenz-Systemen . . . 16
  - 2.2.4. Wizards . . . 17
  - 2.2.5. Konstruktion von Assistenz . . . 18
  - 2.2.6. Assistenz in der Softwareentwicklung . . . 18
- 2.3. Persistenz und OR-Zugriffsschichten . . . 21
  - 2.3.1. Definition von Persistenz und Persistenzschichten . . . 21
  - 2.3.2. OR-Zugriffsschichten . . . 21
  - 2.3.3. Entwicklung von Persistenzschichten . . . 22

3. Analyse **23**
- 3.1. Werkzeugauswahl . . . 23
  - 3.1.1. AndroMDA . . . 24
  - 3.1.2. objectiF . . . 25
  - 3.1.3. openArchitectureWare . . . 25
  - 3.1.4. Zusammenfassung und Auswahl eines Werkzeugs . . . 29
- 3.2. Usability-Test von openArchitectureWare . . . 31
  - 3.2.1. Testziel . . . 31
  - 3.2.2. Testumgebung . . . 31

3.2.3. Testobjekt . . . 32
3.2.4. Aufgabenstellung . . . 32
3.2.5. Testpersonen und Testablauf . . . 32
3.2.6. Testergebnisse . . . 33
3.2.7. Diskussion der Testergebnisse . . . 35
3.2.8. Fazit des Usability-Tests . . . 37

4. Prototyp eines Assistenten für die modellgetriebene Softwareentwicklung 38
4.1. Ziele und Motivation . . . 38
4.2. Rollen bei der Modellierung . . . 39
4.3. Konzept eines Assistenten in Form von Wizards . . . 40
4.3.1. Aufgaben und Schritte . . . 41
4.3.2. Mögliche Probleme . . . 41
4.3.3. Modellgetriebener Wizard . . . 42
4.3.4. Metawizard . . . 44
4.3.5. Dokumentationswizard . . . 44
4.3.6. Unterstützte Rollen . . . 44
4.3.7. Ausgeräumte Ursachen für Fehlhandlungen . . . 45
4.3.8. Alternative Formen von Assistenten . . . 46
4.4. Musterarchitektur für ein Assistenzsystem . . . 47
4.4.1. Statische Sicht . . . 47
4.5. Fachliche Architektur des Prototyps . . . 49
4.5.1. Statische Sicht . . . 50
4.5.2. Dynamische Sicht . . . 53
4.6. Technische Architektur des Prototyps . . . 54
4.7. Der Generator für Assistenten . . . 55
4.7.1. Vorgehen bei der Erstellung von Assistenz-Plugins . . . 55
4.7.2. Komponenten des Generators . . . 55
4.8. Nutzung des Metawizards . . . 57
4.9. Wizard für Persistenz-Modelle . . . 58

5. Usability-Test des Prototyps 60
5.1. Testkonzept . . . 60
5.1.1. Testziel . . . 60
5.1.2. Testumgebung . . . 60
5.1.3. Testobjekt . . . 60
5.1.4. Aufgabenstellung . . . 61
5.1.5. Testpersonen und Testablauf . . . 61
5.2. Testergebnisse . . . 61
5.2.1. Beobachtete Probleme . . . 61
5.2.2. Vergleich mit dem ersten Test . . . 63

5.3. Diskussion der Testergebnisse 65
5.3.1. Konsequenzen für die Weiterentwicklung des Prototyps 67
5.3.2. Handlungsempfehlungen für die Entwicklung von Assistenten und Wizards allgemein 68
5.3.3. Fazit 68
5.4. Erkenntnisse aus den Usability-Tests für Usability-Tests 69

6. Zusammenfassung und Ausblick **70**

Anhang 73

A. Aufgabenstellung der Usability-Tests von openArchitectureWare **73**
A.1. Aufgabenstellung für den ersten Usability-Test 73
A.2. Aufgabenstellung für den Usability-Test des Prototyps 76

B. Bei den Usability-Tests ermittelte Metriken **79**
B.1. Diagramme mit dem Verlauf der erfassten Daten 80

C. Dokumentation des Prototyps **84**
C.1. Metamodelle 84
C.1.1. Das Aufgaben Metamodell 84
C.1.2. Metamodell zum Dokumentieren der Aufgaben und Schritte 89
C.1.3. Metamodell zum Dokumentieren von Metamodell-Elementen 90
C.2. Wizard für Persistenzmodelle 90
C.3. Metawizard 95

D. Inhalt der beiliegenden DVD **98**

Abbildungsverzeichnis **100**

Tabellenverzeichnis **102**

Glossar **103**

Quellenverzeichnis **105**

1. Einleitung

Bei der Entwicklung von Softwaresystemen werden die Entwickler von zahlreichen Software-Werkzeugen unterstützt. Viele der Werkzeuge sind Bestandteil einer integrierten Entwicklungsumgebung (IDE). Diese Werkzeuge können die Entwickler in jeder Phase des Software-Lebenszyklus unterstützen, vom Beginn des Entwicklungsprozesses bis zum Betrieb und zur Wartung eines Softwaresystems.

Die Unterstützung durch die Werkzeuge kann sehr vielfältig sein. Es gibt u. a. Werkzeuge,

- die bei der Aufnahme und Verwaltung von Anforderungen helfen,
- die dem Projektmanagement dienen,
- die den Entwurf eines Systems unterstützen (z. B. Modellierungswerkzeuge, Refaktorisierungswerkzeuge),
- die bei der Implementierung bzw. Programmierung helfen (z. B. Wortvervollständigung, Werkzeuge zum Finden von Referenzen auf eine Klasse, Variable oder eine Methode, Refaktorisierungswerkzeuge),
- die die Qualitätssicherung unterstützen (z. B. Werkzeuge, die Testfälle generieren oder ausführen; Werkzeuge, welche die statische Architektur analysieren),
- die den Betrieb eines Systems überwachen,
- die die Wartung und Weiterentwicklung einer Software unterstützen (z. B. Versionierungswerkzeuge).

Diese Werkzeuge sollen die Softwareentwicklung für die Entwickler leichter handhabbar machen. Den Entwicklern soll durch Automatisierung Arbeit abgenommen werden. Automatisierte Abläufe sind außerdem weniger fehleranfällig. Ohne diese Werkzeuge ist die Entwicklung eines Softwaresystems nicht vorstellbar.

Ein noch relativ neues Vorgehen zur Entwicklung von Softwaresystemen ist die modellgetriebene Softwareentwicklung, bei der Modelle eine zentrale Rolle spielen. In der Praxis hat die modellgetriebene Softwareentwicklung noch keine große Bedeutung. Mittlerweile sind

aber einige Werkzeuge für die modellgetriebene Softwareentwicklung verfügbar. In der Praxis wurden schon einige Projekte erfolgreich durchgeführt, bei denen diese Werkzeuge eingesetzt wurden. Die Hersteller der Werkzeuge werben damit auf ihren Webseiten (siehe z. B. [URL:oAW] oder [URL:objectiF]). Es ist aber noch nicht sicher, ob die zurzeit verfügbaren Werkzeuge schon alle Bedürfnisse der Entwickler erfüllen. Es muss deshalb untersucht werden, ob und mit welchen Werkzeugen die modellgetriebene Softwareentwicklung noch weiter erleichtert werden kann. Mit einer umfassenden Werkzeugunterstützung könnte die modellgetriebene Softwareentwicklung in Zukunft an Bedeutung gewinnen.

In dieser Bachelorarbeit werden die Bedürfnisse der Entwickler bei der modellgetriebenen Softwareentwicklung untersucht und ein Prototyp eines Assistenten vorgestellt, der den Entwicklern die Entwicklung erleichtern soll.

1.1. Vision und Zielsetzung

Ein hoher Grad an Unterstützung durch Werkzeuge ist auch für die modellgetriebene Softwareentwicklung erstrebenswert. Die Vision dieser Arbeit ist deshalb eine Entwicklungsumgebung für die modellgetriebene Softwareentwicklung, welche die Entwickler potenziell bei jeder Tätigkeit mit angemessenen Werkzeugen unterstützt. Sie soll damit den Entwicklern ihre Arbeit so weit wie möglich erleichtern.

Natürlich kann diese Vision im Rahmen einer Bachelorarbeit nicht vollständig umgesetzt werden. Das Ziel dieser Arbeit ist, die Unterstützung der Entwickler in einem zentralen Punkt zu verbessern. Dieser Punkt ist möglicherweise eine umfangreiche Unterstützung der Entwickler bei der Arbeit mit Modellen. Die Entwicklung von Modellen ist eine der hauptsächlichen Tätigkeiten bei der modellgetriebenen Softwareentwicklung. Deshalb ist hier das Potenzial für Verbesserungen besonders groß.

Diese Bachelorarbeit greift diesen Punkt auf und untersucht, welche Probleme bei der Entwicklung von Modellen auftreten können. Das Ziel ist zu zeigen, wie ein Assistent für die modellgetriebene Softwareentwicklung aussehen könnte, der die Probleme lösen kann. Der Assistent soll den Entwicklern die Entwicklung von Modellen erleichtern. Ein Prototyp soll demonstrieren, wie das konkret aussehen kann. Als konkretes Beispiel soll der Prototyp die Erstellung eines Persistenzmodells unterstützen.

1.2. Inhaltlicher Aufbau der Arbeit

Nach dieser Einleitung erläutert das zweite Kapitel die wichtigsten Grundlagen, auf denen diese Arbeit basiert. Besonders ausführlich geht das zweite Kapitel auf die modellgetriebene

Softwareentwicklung ein. Die modellgetriebene Softwareentwicklung spielt in dieser Arbeit eine zentrale Rolle. Außerdem wird gezeigt, wie Assistenten die Benutzer unterstützten können. Der im Rahmen dieser Arbeit entwickelte Prototyp soll die Erstellung eines Persistenzmodells mit einem Assistenten demonstrieren. Der letzte Abschnitt des Grundlagen-Kapitels geht erläutert deshalb, was man unter Persistenz und Persistenzschichten versteht.

Das dritte Kapitel analysiert den Istzustand der modellgetriebenen Softwareentwicklung. Zuerst werden drei repräsentative Werkzeuge vorgestellt. Eines der Werkzeuge wird ausgewählt, um darauf aufbauend den Prototyp eines Assistenten zu entwickeln. Im zweiten Abschnitt des dritten Kapitels wird das zuvor ausgewählte Werkzeug einem Usability-Test unterzogen. Der Test soll Probleme bei der modellgetriebenen Softwareentwicklung aufzeigen, die der in dieser Arbeit entwickelte Assistent verbessern soll.

Das vierte Kapitel stellt einen Prototyp eines Assistenten für die modellgetriebene Softwareentwicklung vor. Der erste Abschnitt stellt das Konzept des Assistenten vor. Danach wird die Architektur des Prototyps erläutert. Zum Schluss wird gezeigt, wie der Prototyp konkret aussieht.

Wie gut der Prototyp tatsächlich die modellgetriebene Softwareentwicklung unterstützt, überprüft ein weiterer Usability-Test. Das fünfte Kapitel listet die Ergebnisse dieses Tests auf. Anhand der Ergebnisse wird bewertet, ob das entwickelte Konzept prinzipiell umsetzbar ist.

Das sechste Kapitel schließt diese Arbeit mit einer Zusammenfassung der wichtigsten Ergebnisse und einem Ausblick auf die Weiterentwicklung von Assistenten für die modellgetriebene Softwareentwicklung ab.

2. Grundlagen

In diesem Kapitel werden die wesentlichen Grundlagen vorgestellt, auf denen diese Arbeit basiert. Der erste Abschnitt führt die modellgetriebene Softwareentwicklung ein, die eine zentrale Rolle in dieser Arbeit spielt. Der in dieser Arbeit entwickelte Assistent unterstützt exemplarisch das Modellieren eines Persistenzmodells. Deshalb werden im zweiten Abschnitt die wichtigsten Grundlagen von Assistenten und Wizards vorgestellt und zum Schluss wird erläutert, was man unter Persistenz und Zugriffsschichten versteht.

2.1. Modellgetriebene Softwareentwicklung

Dieses Unterkapitel erklärt, was man unter modellgetriebener Softwareentwicklung versteht, und zeigt die wichtigsten Ziele sowie Vor- und Nachteile der modellgetriebenen Softwareentwicklung. Zum Schluss grenzt dieses Unterkapitel die modellgetriebene Softwareentwicklung von ähnlichen Konzepten ab und zeigt, wie man bei nicht modellgetriebener Softwareentwicklung die gleichen Ziele erreichen kann.

2.1.1. Definition

Bei modellgetriebener Softwareentwicklung (Model Driven Software Development, MDSD) nehmen Modelle überwiegend die Rolle von manuell geschriebenem Programmcode ein [Stahl u. a. (2007)]. Modelle werden hier zu Artefakten erster Klasse [Gruhn u. a. (2006)].

Modelle werden häufig als Dokumentation oder Bauplan genutzt, nach dem das System manuell implementiert wird. [Stahl u. a. (2007)] nennt diese Art der Nutzung von Modellen modellbasiert. Bei modellgetriebener Softwareentwicklung beschreiben die Modelle nicht nur die Architektur eines Systems, die Modelle sind zugleich die Architektur. Um dies zu erreichen, ist es notwendig, aus den Modellen automatisiert lauffähige Software zu erzeugen. Änderungen an der Systemarchitektur werden an den Modellen vorgenommen, nicht am Programmcode.

Es gibt drei zentrale und miteinander verknüpfte Punkte, mit denen sich das MDSD befasst:

- formale Modelle zur Beschreibung eines Systems,
- lauffähige Software, die den formalen Modellen entspricht,
- automatisierte Erzeugung lauffähiger Software.

Ein formales Modell beschreibt einen Aspekt der Software vollständig. UML-Modelle sind ein Beispiel für formale Modelle. Es gibt beliebige weitere Arten von formalen Modellen, die in grafischer oder textueller Form verfasst sein können. Da Modelle in der Regel nicht ein gesamtes System beschreiben können, sind auch bei MDSD manuell geschriebene Programmteile nötig. Diese implementieren Teile des Systems aber beeinflussen im Allgemeinen nicht die Architektur des Systems.

Ein Modell wird automatisiert durch ein Programm in lauffähige Software übersetzt und nicht, wie bei modellbasiertem Vorgehen, manuell durch einen Programmierer. Dies ist entscheidend für die modellgetriebene Softwareentwicklung.

Um aus den formalen Modellen lauffähige Software zu erzeugen, gibt es zwei grundlegende Ansätze. Ein Ansatz sind Generatoren, die Quelltext in einer Programmiersprache erzeugen. Der andere Ansatz sind Interpreter, die das Modell zur Laufzeit einlesen. Bei beiden Ansätzen führt ein Rechner das Modell aus. Der Unterschied ist, wann das Modell ausgewertet wird: zur Laufzeit oder davor.

2.1.2. Modellierung

Dieser Abschnitt gibt einen Überblick über die Modellierung beim MDSD und erläutert die wichtigsten Begriffe, die dabei auftreten. Abbildung 2.1 stellt den Zusammenhang zwischen den wichtigsten Begriffen dar.

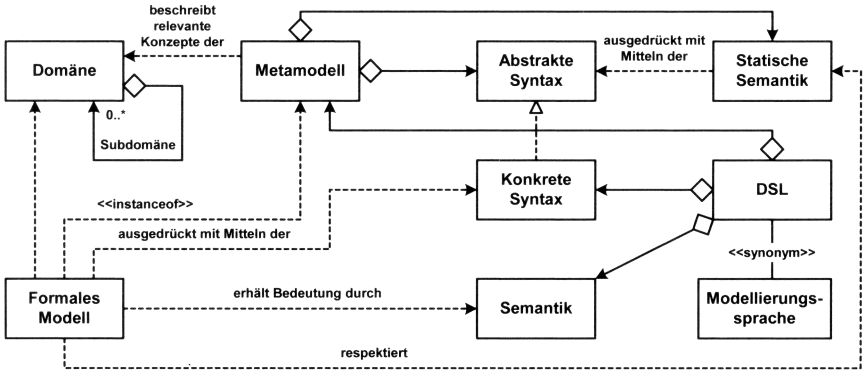


Abbildung 2.1.: Zusammenhang der Begriffe bei der Modellierung (aus [Stahl u. a. (2007)])

Domäne

Unter einer Domäne versteht man beim MDSD ein begrenztes Interessen- und Wissensgebiet. Innerhalb der Welt einer Domäne beschreibt ein Modell bestimmte Aspekte eines Systems. Domänen können in Subdomänen unterteilt werden und können technisch oder fachlich sein. Eine technische Domäne ist z.B. die statische Sicht auf eine Softwarearchitektur, die weiter in Klassen und Komponentenmodelle unterteilt werden kann. Die Domäne Kreditinstitut, mit Konzepten wie Konto und Kredit, ist ein Beispiel für eine fachliche Domäne. [Stahl u. a. (2007)]

Metamodell

Ein Metamodell ist eine formale Beschreibung einer Domäne. Es beschreibt mit einer abstrakten Syntax und einer statischen Semantik die relevanten Konzepte einer Domäne. [Stahl u. a. (2007)]

Abstrakte Syntax

Die abstrakte Syntax eines Metamodells definiert die Elemente und deren Beziehungen, mit denen ein Metamodell eine Domäne beschreibt. Die abstrakte Syntax eines UML-Klassendiagramms definiert zum Beispiel, dass es Klassen gibt, die Attribute und Operationen haben können und, dass zwischen zwei Klassen eine Generalisierungsbeziehung bestehen kann. [Stahl u. a. (2007)]

Konkrete Syntax

Die konkrete Syntax legt fest, wie die Elemente einer abstrakten Syntax tatsächlich aussehen. Für eine abstrakte Syntax können verschiedene konkrete Syntaxen definiert werden. Eine konkrete Syntax kann entweder grafisch oder textuell sein. Ein UML-Klassendiagramm hat eine grafische konkrete Syntax, die z.B. definiert, dass eine Generalisierungsbeziehung durch einen Pfeil dargestellt wird. Programmiersprachen sind textuelle konkrete Syntaxen. Die Sprache Java legt zum Beispiel fest, dass eine Generalisierung mit dem Schlüsselwort «extends» in der spezialisierenden Klasse angegeben wird.

Statische Semantik

Die statische Semantik eines Metamodells legt Bedingungen fest, die ein Modell erfüllen muss. [Stahl u. a. (2007)]

Domänenspezifische Sprache

Eine domänenspezifische Sprache (Domain Specific Language, DSL) ist eine Sprache, mit der Aspekte innerhalb einer Domäne formuliert werden können. Sie besteht aus einem Metamodell und einer konkreten Syntax. [Stahl u. a. (2007)]

Formale Modelle

Formale Modelle sind mit einer DSL beschriebene Aspekte eines Systems. Der Informationsgehalt eines formalen Modells ist auf die Domäne beschränkt. Ein formales Modell ist bei MDSD gleichzeitig auch ein Programm, da es, wie in Abschnitt 2.1.1 erwähnt, automatisiert zum Ausführen gebracht werden kann. [Stahl u. a. (2007)]

Metametamodell

Ein Metametamodell beschreibt die Elemente eines Metamodells. Damit ist ein Metametamodell ein Metamodell für ein Metamodell. Die Einteilung der Modelle in Modell, Metamodell und Metametamodell ist immer relativ zu einem Modell zu sehen. Für jedes Modell existiert ein Metamodell. Deshalb kann man prinzipiell beliebig viele Meta-Ebenen unterscheiden. In der Regel verwendet man aber nur eine dreistufige Hierarchie von Modellen, in der das Metametamodell sich selbst beschreibt. Abbildung 2.2 zeigt eine solche Hierarchie. Ein häufig verwendetes Metametamodell ist die Meta Object Facility (MOF), die u. a. von der UML verwendet wird. [Stahl u. a. (2007)]

Modellierung mit UML

Eine häufig verwendete Modellierungssprache ist die UML. Die UML kann auch bei MDSD als Metamodell verwendet werden. Die UML ist sehr allgemein gehalten, um möglichst viele Aspekte abzudecken. Bei MDSD sind jedoch spezielle Metamodelle sinnvoll, um ein System zu beschreiben [Stahl u. a. (2007)]. Deshalb hat die UML2.0 einen Erweiterungsmechanismus, mit dem sie an die speziellen Gegebenheiten einer Domäne angepasst werden kann.

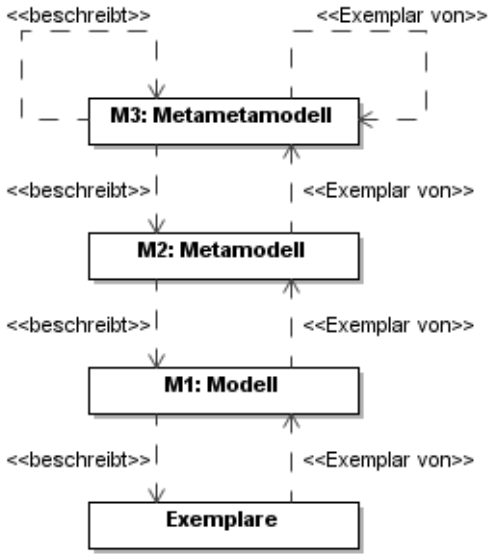


Abbildung 2.2.: Dreistufige Hierarchie von Modellen (nach [Stahl u. a. (2007)])

Mit Stereotypen kann das UML2.0-Metamodell um eigene Metamodell-Elemente erweitert werden. Ein Stereotyp ist eine Erweiterung eines Metamodell-Elements der UML. Für jedes Stereotyp kann man Attribute (Properties) definieren. Um ein Element eines Modells mit einem Stereotyp zu klassifizieren, fügt man das Stereotyp zu den, auf das Element angewandten, Stereotypen hinzu. Die Werte der Attribute eines Stereotyps werden für ein Modell-Element mit Tagged-Values gesetzt.

2.1.3. Model Driven Architecture (MDA)

Einen hohen Bekanntheitsgrad hat derzeit die Model Driven Architecture (MDA) [URL:MDA]. Sie ist ein von der Object Management Group[1] (OMG) vorgeschlagener Standard zur Umsetzung der modellgetriebenen Softwareentwicklung. Die MDA ist aber nicht mit MDSD gleichzusetzen.

Die MDA ist die Spezifikation der Architektur einer Infrastruktur, welche die modellgetriebene Softwareentwicklung umsetzt [Gruhn u. a. (2006)]. Zu dieser Infrastruktur gehören der unterliegende Software-Prozess sowie die verwendeten Hilfsmittel und Werkzeuge [Gruhn u. a. (2006)]. Das Wort «Architecture» in MDA steht also nicht für die Architektur des Systems, das entwickelt wird.

[1] http://www.omg.org

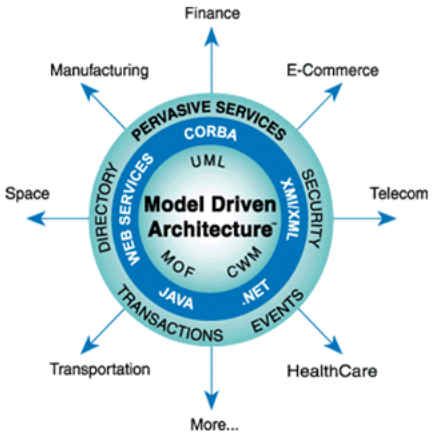


Abbildung 2.3.: Der schalenartige Aufbau der MDA (Quelle: [URL:MDA])

Die MDA deckt nicht alle Konzepte des MDSD ab. Sie spezialisiert das MDSD und schränkt es in einigen Punkten ein [Stahl u. a. (2007)]. Eines der Hauptziele der MDA ist die Interoperabilität von Werkzeugen [Stahl u. a. (2007)]. Um einheitliche Modelle zu schaffen, schreibt die MDA die Verwendung der Meta Object Facility (MOF) als Metametamodell vor. Die MOF ist ein sich selbst beschreibendes Metametamodell und führt zu einer Hierarchie mit drei Modell-Ebenen (siehe Abb. 2.2).

Die MDA unterscheidet zwischen den grundsätzlichen Abstraktionsebenen Platform Independent Model (PIM) und dem Platform Specific Model (PSM). Die Modellierung beginnt mit dem PIM, welches die Anwendungslogik beschreibt. Das PIM wird anschließend zu einem PSM verfeinert. Das PSM beschreibt die technische Umsetzung [Gruhn u. a. (2006)]. In die Kategorie der PSM gehört auch der generierte Programmcode. Ob ein Modell plattformspezifisch ist, ist immer relativ. Ein PIM ist relativ zu der Plattform unspezifisch, die in einem, dieses PIM spezialisierenden, PSM verwendet wird [Stahl u. a. (2007), Gruhn u. a. (2006)]. Ein PIM kann also durchaus technische Details enthalten. Zwischen den Modellen liegen ein oder mehrere Transformationsschritte.

Kritik an der MDA

Zum Teil wird die MDA heftig kritisiert. In [Thomas (2004)] wird die Umsetzung der MDA durch die OMG kritisiert. Die UML (ebenfalls ein Standard der OMG) sei aufgrund der allgemeinen und visuellen Notation gut geeignet, Software zu beschreiben. Die OMG habe es aber versäumt, die Interoperabilität der UML-Werkzeuge voranzutreiben. Beim UML-2.0 Standard vermisst [Thomas (2004)] eine verständliche Semantik und Referenzimplementierungen. Dies erschwere eine Implementierung von UML-Transformationswerkzeugen für die MDA.

2.1.4. Bewertung der modellgetriebenen Softwareentwicklung

Die Anwendung von modellgetriebener Softwareentwicklung verspricht eine Reihe von Vorteilen. MDSD bringt aber auch neue Probleme mit sich. Dieser Abschnitt stellt die wichtigsten Vor- und Nachteile des MDSD vor.

Hoher Abstraktionsgrad

Der wichtigste Grund für das MDSD sei laut [Stahl u. a. (2007)] die Möglichkeit, das System auf einer höheren Abstraktionsebene als der zugrundeliegenden Programmiersprache zu entwickeln. Dies ermöglicht es, das System näher an der fachlichen Domäne zu entwickeln und mit treffenderen Elementen zu beschreiben (z.B. Komponente oder Entität statt Klasse). Die Komplexität der Software-Artefakte soll dadurch reduziert werden. Details der technischen Umsetzung sind nicht im Modell enthalten und müssen dem Modell-Entwickler nicht bekannt sein [Stahl u. a. (2007)].

In den 1970er Jahren wurde die immer weiter steigende Komplexität von Softwaresystemen als großes Problem erkannt [Dijkstra (1972)]. Die Reduzierung der Komplexität durch den Einsatz von MDSD scheint deshalb ein großer Vorteil des MDSD zu sein. Laut [Stahl u. a. (2007)] teile sich die Gesamtkomplexität in die Abbildung vom Modell in die Zielsprache und die Komplexität, die im Modell übrig bleibt, auf. Beide Teile könne man separat angehen und verstehen [Stahl u. a. (2007)]. Allerdings bedeutet dies noch keine Reduzierung der Komplexität des Softwareentwicklungsprozesses insgesamt. Vielmehr führt der Einsatz von MDSD zu einer Verlagerung der Komplexität an andere Stellen im Entwicklungsprozess [Hailpern und Tarr (2006)]. Mit zunehmender Anzahl von System-Artefakten (Modelle und Programmcode) steigt auch die Zahl der Beziehungen zwischen den Artefakten. Dadurch steigt potenziell auch die Komplexität sowohl der Beziehungen zwischen den Artefakten als auch der Werkzeuge, die die Modelle bearbeiten. Dies kann sich negativ z.B. bei der Wartung, dem Debugging oder der Änderung von Software bemerkbar machen [Hailpern und Tarr (2006)].

Die Beziehungen zwischen den Artefakten führen zu Problemen, wenn Änderungen eines Artefakts des Systems sich auf andere Artefakte auswirken ([Hailpern und Tarr (2006)] bezeichnet dies als «rampant round-trip problems»). Zum Teil können die betroffenen Stellen durch Modelltransformationen automatisch angepasst werden, z.B. wenn eine Klasse in einem Klassendiagramm eine neue Methode erhält. Es können aber nicht alle Änderungen automatisch behandelt werden. Wenn die Implementierung einer Methode geändert wurde, muss ein Entwickler beurteilen, ob die Änderung Auswirkungen auf die Semantik der zugehörigen Modelle hat [Hailpern und Tarr (2006)].

Es gibt noch ein weiteres Problem bei den Beziehungen zwischen den Artefakten eines Systems. Um die Beziehungen zu verstehen, benötigt man Expertenwissen über die Domänen aller Modelle. Einerseits hilft der hohe Abstraktionsgrad der Modelle, den Spezialisten einer Domäne Modelle zu entwickeln, andererseits wird dadurch das Verständnis der Beziehungen zwischen Modellen erschwert. Um die Beziehungen und Auswirkungen einer Änderung beurteilen zu können, sind hinreichende Kenntnisse in verschiedenen Technologien und Terminologien nötig [Hailpern und Tarr (2006)].

Interoperabilität

Durch den hohen Abstraktionsgrad wird die Interoperabilität und Plattformunabhängigkeit potenziell erhöht. Es ist denkbar, ein und dasselbe Modell auf verschiedene Weise technisch zu realisieren, also z.B. einen Wechsel der Plattform von Java auf .Net vorzunehmen oder das verwendete Persistenz-Framework auszutauschen. In der Praxis ist die Plattformunabhängigkeit nur eingeschränkt gegeben. Gründe hierfür seien laut [Stahl u. a. (2007)] Details der Plattform, die sich auf gewisse Weise doch in den Modellen niederschlügen. Hinzu käme, dass das System in der Regel auch aus nicht generiertem Code bestehe, der von Hand geschrieben und plattformspezifisch ist.

Einheitliche Architektur

Wie in Abschnitt 2.1.1 erwähnt, werden beim MDSD Änderungen des Systems an den Modellen selbst vorgenommen. Dies hat den Vorteil, dass Inkonsistenzen zwischen dem Modell und der tatsächlichen Architektur vermieden werden [Stahl u. a. (2007)].

Das MDSD übersetzt die durch ein Modell beschriebenen Elemente automatisch. Dadurch ist gewährleistet, dass die übersetzten Elemente auf eine einheitliche Art implementiert werden und die Architektur des Systems nicht verfälscht wird [Stahl u. a. (2007)].

Produktivitätssteigerung

Die automatische Generierung von lauffähiger Software erspart das manuelle Schreiben von Quelltext. Die Zeitersparnis dadurch ist aber nur gering, da die Zeit zur Entwicklung einer Lösung eines Problems überwiegt [Stahl u. a. (2007)]. Eine wirkliche Produktivitätssteigerung ist bei MDSD nur mittelfristig realistisch, wenn Änderungen am System vorgenommen wurden [Stahl u. a. (2007)]. Die Zeitersparnis ergibt sich hier aus dem wiederholten automatisierten Generieren des geänderten Systems.

Fazit

Die in diesem Abschnitt vorgestellten Vorteile zeigen, dass sich der Einsatz von MDSD positiv auf einige Qualitätskriterien auswirken kann. Zum Teil können die Vorteile aber durch neue Probleme aufgewogen werden. Eine Aussage, ob der Einsatz von MDSD die Gesamtqualität eines Softwareprodukts steigert und an den Punkten erfolgreich ist, an denen andere Softwareentwicklungsansätze gescheitert sind, ist wegen der vielen Faktoren mit Vorsicht zu betrachten.

2.1.5. Abgrenzung und Alternative Ansätze

Es gibt eine Reihe von Ansätzen zur Softwareentwicklung, die dem Vorgehen beim MDSD ähneln oder ähnliche Ziele verfolgen, wie das MDSD. Dieser Abschnitt grenzt zunächst das Computer Aided Software Engineering (CASE) vom MDSD ab und stellt dann mit Quasar eine grundsätzlich andere Möglichkeit vor, Anwendung und Technik zu trennen.

Abgrenzung vom CASE

Die modellgetriebene Softwareentwicklung hat in einigen Punkten Gemeinsamkeiten mit dem Computer Aided Software Engineering (CASE). MDSD und CASE sind dennoch eigenständige Ansätze, die nicht miteinander verwechselt werden dürfen.

Computer Aided Software Engineering ist die Verwendung von Werkzeugen, die den Softwareentwicklungsprozess unterstützen. Solche Werkzeuge sind zum Beispiel Werkzeuge zum Refaktorisieren, erstellen von UML-Modellen und generieren von Code.

Die Codegenerierung aus Modellen ist eine Gemeinsamkeit von CASE mit dem MDSD. CASE ist im Gegensatz zur MDSD jedoch durch stellenweise Starrheit gekennzeichnet. So gibt ein CASE-Werkzeug zumindest einen der Punkte Modellierungssprache, Transformation oder Plattform fest vor [Stahl u. a. (2007)]. Außerdem ist die Generierung von Code mittels eines CASE-Werkzeugs meistens ein einmalig ausgeführter Vorgang. Bei MDSD wird nach jeder Änderung des Modells die Transformation erneut durchgeführt.

Quasar

Quasar (Qualitätssoftwarearchitektur) [Siedersleben (2004)] und MDSD haben beide das Ziel, Anwendung und Technik zu trennen und somit näher an der fachlichen Domäne entwickeln zu können. Der Unterschied von Quasar und MDSD liegt in der Umsetzung dieses Ziels.

Bei Quasar stehen Komponenten und Schnittstellen im Vordergrund. Die Trennung von Anwendung und Technik wird dadurch erreicht, dass eine Komponente sich entweder mit der Anwendungsdomäne oder der Technik beschäftigt, aber nicht beidem. Quasar ist also als Referenzarchitektur zu verstehen. MDSD gibt keine Vorgaben an die Architektur des zu entwickelnden Systems. Die Entwicklung einer Quasar konformen Architektur ist beim MDSD dennoch nicht ausgeschlossen.

Diese Gegenüberstellung hat gezeigt, dass es grundsätzlich verschiedene Wege gibt, nah an der Anwendungsdomäne zu entwickeln. Für dieses Ziel gibt Quasar eine Architektur vor, und MDSD gibt vor, wie modelliert wird.

2.1.6. Fazit

Die modellgetriebene Softwareentwicklung ist ein interessanter Ansatz, um ein System auf einer anwendungsnahen Abstraktionsebene zu entwickeln. Dennoch muss man für jedes Projekt individuell die Vor- und Nachteile des MDSD abwägen, um zu entscheiden, ob das MDSD einem nicht modellgetriebenen Vorgehen vorzuziehen ist. Zumindest heute ist noch offen, ob die modellgetriebene Softwareentwicklung die Probleme anderer Entwicklungsansätze nachhaltig lösen kann.

2.2. Assistenten und Wizards

2.2.1. Motivation für Assistenz in Softwaresystemen

Die Arbeit mit einem Softwaresystem sollte optimalerweise von jedem Benutzer ohne Probleme möglich sein. In ISO 9126 werden deshalb entsprechende Qualitätskriterien für Software definiert (Funktionalität, Benutzbarkeit etc.). Das Ziel, dass sowohl Anfänger als auch Experten unter intuitiver Benutzung eines Softwaresystems ihre Arbeit erledigen können, ist aber eine nicht erreichbare Idealvorstellung. Probleme der Benutzer müssen als normal gesehen werden. Zum einen wird es mit zunehmendem Funktionsumfang schwieriger, ein System den Benutzern in jedem Punkt und auf den ersten Blick verständlich zu machen. Zum anderen gibt es gar keine intuitive Benutzung einer Software [Raskin (2000)]. Deshalb müssen die Entwickler dafür sorgen, dass sowohl die Probleme der Benutzer minimiert werden, als auch im Falle von Problemen den Benutzern angemessene Unterstützung zur Lösung der Probleme angeboten wird. Dies kann durch den Einsatz eines Assistenz-Systems erreicht werden.

Die Grundprinzipien von Assistenz-Systemen sind die Bereitstellung von Informationen und Automation von Aufgaben [Ames (2001), Rech u. a. (2006)].

Die Bereitstellung von Informationen, die ein Benutzer zur korrekten Erledigung einer Aufgabe benötigt, kann die Bedienung einer Benutzungsoberfläche erleichtern. Die benötigten Informationen sind in einer Benutzungsoberfläche ohne Assistenten nur aus einer separaten Online-Hilfe oder einem Handbuch zu entnehmen. Ein Assistenz-System kann die benötigten Informationen aber mit der Benutzungsoberfläche verschmelzen. Dadurch ist es einem Assistenz-System möglich die Fragen der Benutzer zu beantworten, bevor sie gestellt werden. Das Assistenz-System nimmt den Benutzer an die Hand und kann damit vermeiden, dass er scheitert, eine Aufgabe zu erledigen. [Ames (2001)]

Zusätzlich kann durch Automation von einfachen und sich wiederholenden Aufgaben dem Benutzer viel Arbeit abgenommen werden [Rech u. a. (2006)]. Der Benutzer kann sich mehr auf fachliche Dinge konzentrieren als auf die handwerkliche Umsetzung. Außerdem sind automatisierte Abläufe weniger fehleranfällig. Ein allgemeines Beispiel für automatisierte Abläufe ist die weitverbreitete Undo-Redo Funktionalität.

2.2.2. Merkmale von Assistenz-Systemen

Es gibt vier Merkmale, nach denen man die Methoden, mit denen Assistenz angeboten wird, unterscheiden kann [Rech u. a. (2006)]:

- **Der Zeitpunkt, an dem Assistenz angeboten wird**
  Die Unterstützung kann vor oder nach einer Aktion eines Benutzers angeboten werden. Jede Eingabe mit einem Eingabegerät (z. B. Maus und Tastatur) kann eine Aktion sein. Eine Aktion kann auch eine Folge von Eingaben sein. Die einem Benutzer angebotene Assistenz ist entweder proaktiv oder reaktiv. Bei reaktiver Assistenz verlangt der Benutzer explizit nach Unterstützung, beispielsweise wenn er ein Problem hat. Proaktive Assistenz bietet dem Benutzer vor einer Aktion Unterstützung an, um Probleme zu vermeiden [Ames (2001)].
- **Die Art, wie Assistenz angeboten wird**
  Die Unterstützung kann visuell und akustisch erfolgen. Denkbar sind z. B. Text, Videos, Animationen oder Audio-Kommentare.
- **Der Ort, an dem Assistenz angeboten wird**
  Der Ort, an dem die Unterstützung für den Benutzer erkennbar wird, kann z. B. ein Fenster der Anwendung, ein Tooltip oder das Betriebssystem sein.
- **Die Absicht mit der Assistenz angeboten wird**
  Dem Benutzer könnte z. B. eine Lösung für ein Problem vorgeschlagen werden oder er könnte auf eine neue Funktion des Programms aufmerksam gemacht werden.

Mindestens eine der genannten Eigenschaften ist bei einem Assistent festgelegt. Die Übrigen kann der Assistent zur Laufzeit beeinflussen [Rech u. a. (2006)].

2.2.3. Techniken von Assistenz-Systemen

Es gibt zahlreiche Techniken, mit denen ein Assistenz-System dem Benutzer Informationen präsentieren kann. Einige der bekanntesten sind im Folgenden aufgelistet (nach [Ames (2001)]):

- Beschriftung der Elemente einer grafischen Benutzungsoberfläche.
- Funktionen, die die Durchführung einer Aufgabe demonstrieren (Do-It-For-You, «zeig mir» Funktionalität). Die in einem Hilfetext beschriebenen Schritte einer Aufgabe kann der Assistent an einem Beispiel ausführen.
- Wizards (Erläuterung in Abschnitt 2.2.4).
- «Information-Rich Interfaces»: Benutzungsoberflächen, die dem Benutzer viel mehr Informationen in Textform bieten, als klassische Benutzungsoberflächen. Die Informationen gehen über die Beschriftung der Elemente einer grafischen Benutzungsoberflächen hinaus. Solche Benutzungsoberflächen sind überwiegend bei Webseiten zu finden.

- Eingebettetes Hilfesystem. Die Benutzungsoberfläche eines Systems und das zugehörige Hilfesystem sind vereint und nicht zwei getrennte Anwendungen. Ein eingebettetes Hilfesystem ermöglicht es, dem Benutzer kontextabhängige Hilfe anzubieten.
- Intelligente Assistenten, die die Bedürfnisse der Benutzer proaktiv erkennen. Der Microsoft Agent aus Microsoft Office ist ein Beispiel dafür. Der Microsoft Agent wurde aber von vielen Benutzern als störend empfunden und ist deshalb gescheitert.

Während mit den ersten drei genannten Techniken die angezeigten Informationen in der Regel statisch festlegt werden, können mit den letzten drei Techniken die Informationen zur Laufzeit und speziell für einen Benutzer ermittelt werden.

2.2.4. Wizards

Wizards sind eine weitverbreitete Methode, einen Benutzer zu unterstützen. Sie sind besonders nützlich, um komplexe Aufgaben zu automatisieren. Ein Wizard wird durch eine Serie von Schritten, die Daten zum Erledigen einer bestimmten Aufgabe sammeln, gekennzeichnet. Dies erfolgt üblicherweise in einem Fenster einer grafischen Benutzungsoberfläche, kann aber auch in textueller Form über die Kommandozeile erfolgen. Wizards sind besonders geeignet für Aufgaben, die algorithmisch in einer linearen Folge von Schritten gelöst werden können [Dryer (1997)].

Es gibt verschiedene Möglichkeiten, die Schritte eines Wizard abzuarbeiten [Burton u. a. (1999)]:

- «vorwärts» und «rückwärts» Befehle, mit denen man von einem Schritt zum nächsten gelangt.
- Inhaltsverzeichnis an der linken oder rechten Seite des Wizardfensters. Alle Schritte sind aufgelistet und können in einer beliebigen Reihenfolge bearbeitet werden.
- Tabs an der oberen Seite des Fensters. Jeder Schritt ist einer Registerkarte zugeordnet. Die Reihenfolge der Schritte ist beliebig.
- Drop Down Box mit einer Liste der Schritte. Die Reihenfolge der Schritte ist beliebig.

Die Variante mit den «vorwärts» und «rückwärts» Befehlen ist weit verbreitet, hat aber den Nachteil, dass der Ablauf starr vorgegeben ist. Man kann diese Variante aber mit den anderen kombinieren, um dem Benutzer mehr Freiraum zu lassen. Laut der Studie [Burton u. a. (1999)] sei ein Wizard mit einem Inhaltsverzeichnis eindeutig am einfachsten zu bedienen. Die Aufgaben würden mit solch einem Wizard am häufigsten erfolgreich bearbeitet werden. Ein Wizard sollte dennoch die Möglichkeit bieten mit «vorwärts» und «rückwärts» Befehlen die Schritte abzuarbeiten [Burton u. a. (1999)].

2.2.5. Konstruktion von Assistenz

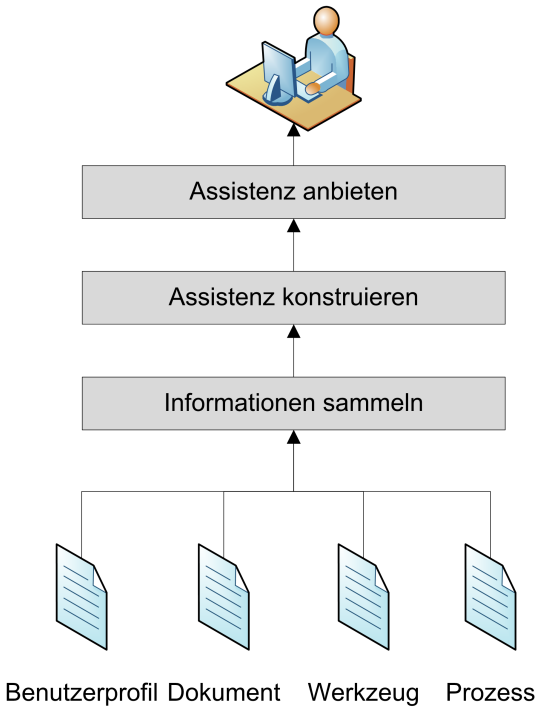


Abbildung 2.4.: Datenfluss bei Assistenzsystemen (nach Rech u. a. (2006))

Die den Benutzern angebotene Assistenz wird mit einem Konstruktionsalgorithmus erzeugt. Der Algorithmus muss beachten, an wen die Assistenz gerichtet ist. Gemäß dem User Centered Design (UCD) [Neumann (2005)] müssen die unterschiedlichen Bedürfnisse der Benutzer beachtet werden. Die angebotene Assistenz ist darüber hinaus abhängig von dem unterstützten Prozess [Rech u. a. (2006)]. Zum Beispiel sind beim Testen andere Aufgaben durchzuführen als beim Programmieren. Der Konstruktionsalgorithmus extrahiert die benötigten Informationen aus verschiedenen Quellen. Als Informationsquelle können z. B. Daten über den Benutzer, das bearbeitete Dokument, der Prozess oder die verwendeten Werkzeuge dienen. Den Datenfluss eines solchen Algorithmus veranschaulicht Abbildung 2.4.

2.2.6. Assistenz in der Softwareentwicklung

Die Entwicklung eines Softwaresystems ist komplex und durch wissensintensive Tätigkeiten geprägt. Während des Entwicklungsprozesses werden zahlreiche Dokumente (Anforderungen, Quelltexte, Diagramme, Spezifikationen etc.) erstellt und weitergegeben. Ein einzelnes Dokument kann viele Informationen enthalten. Die Arbeit mit den Dokumenten ist deshalb

wissensintensiv. Bei der Erstellung, Modifikation oder Wartung eines Dokuments benötigt ein Entwickler Informationen über das Arbeitsdokument selbst und Informationen über weitere Dokumente, von denen das Arbeitsdokument abhängt. Oft hat ein Entwickler die Dokumente nicht selbst erstellt, wodurch es für ihn schwer ist, alle Informationen zu überblicken. Assistenzsysteme für die Softwareentwicklung können es erleichtern, diesen Informationsbedarf zu erfüllen und die Komplexität des Entwicklungsprozesses zu handhaben. [Rech u. a. (2006)]

Assistenzsysteme können außerdem durch Automation viele Tätigkeiten bei der Softwareentwicklung erleichtern.

Assistenzsysteme können alle Phasen des Software-Lebenszyklus unterstützen. Die Unterstützung kann dabei, wie die folgenden Beispiele zeigen, in sehr unterschiedlichen Formen auftreten:

- Automation von Aufgaben bei der Softwareentwicklung.
  Bei der Modellierung und der Implementierung können die entwickelten Modelle bzw. Programme automatisch refaktorisiert werden. Ebenfalls können Teile der Qualitätssicherung automatisiert werden. Es gibt u. a. Testwerkzeuge, die Testfälle generieren und statische Analysen sowie dynamische Tests automatisch ausführen. [Rech u. a. (2006), Spillner und Linz (2005)]
- Wizards, welche die zum Ausführen einer Aufgabe benötigten Informationen sammeln und anschließend die Aufgabe automatisiert ausführen. Beispielsweise Wizards zum Anlegen eines Projekts oder dem Rumpf einer Klasse.
- Assistenz, die den Entwicklern über Querverweise und Visualisierungen einen Einblick in das laufende Projekt ermöglicht [Rech u. a. (2006)].
- Assistenz, welche die Interaktion zwischen den an einem Projekt beteiligten Personen ermöglicht [Rech u. a. (2006)].
- Die Quelltext-Editoren erleichtern das Schreiben von Quelltexten mit zahlreichen Funktionen:
  - Syntaxhighlighting erleichtert das Lesen von Quelltexten.
  - Wortvervollständigung erleichtert das Schreiben von Quelltexten.
  - Die Erzeugung von Code-Fragmenten erleichtert das Schreiben von Quelltexten.
  - Die Integration der Dokumentation von Programmbibliotheken erleichtert die Verwendung der Programmbibliotheken. Die Dokumentation kann beispielsweise mittels Tooltips verfügbar gemacht werden.
  - Hervorhebung von Syntaxfehlern und Verletzungen von Programmierrichtlinien.

- Eine Auswirkungsanalyse bei Änderungen bzw. Refaktorisierungen am Programm oder Modell ist besonders hilfreich, um Fehlerzustände zu vermeiden [Rech u. a. (2006)]. Sie kann proaktiv mögliche Fehlerzustände aufdecken, die durch eine Änderung entstehen würden. Ein Entwickler kann beispielsweise gewarnt werden, wenn er eine Variable umbenennt und es in einem übergeordneten Gültigkeitsbereich eine Variable mit dem gleichen Namen gibt.

Eine Untersuchung der Assistenz bei der Softwareentwicklung in [Rech u. a. (2006)] hat ergeben, dass insbesondere bei der Aufnahme sowie Analyse von Anforderungen, dem Systementwurf, der Programmierung und der Projektplanung ein hoher Bedarf von Assistenz besteht. Bei den anderen Tätigkeiten bestehe laut [Rech u. a. (2006)] weniger Bedarf, weil diese meistens von Spezialisten durchgeführt werde.

2.3. Persistenz und OR-Zugriffsschichten

Der im Rahmen dieser Bachelorarbeit entwickelte Prototyp demonstriert einen Assistenten, der die Entwickler bei der Entwicklung eines Persistenzmodells unterstützt. Dieser Abschnitt erläutert deshalb kurz, was man in der Informatik unter Persistenz versteht.

2.3.1. Definition von Persistenz und Persistenzschichten

Unter Persistenz versteht man in der Informatik das dauerhafte Speichern von Daten in nicht flüchtigem Speicher. Dies ist nötig, wenn Daten nicht verloren gehen dürfen oder wenn die Kapazität des Hauptspeichers nicht ausreicht [Starke (2005)].

Die Implementierung der Datenspeicherung wird üblicherweise in einer Persistenzschicht gekapselt, um sie von der Anwendungslogik zu trennen [Bauer und King (2006), Starke (2005)]. Nach außen bietet eine Persistenzschicht Schnittstellen mit Operationen zum Speichern, Löschen, Ändern und Abfragen von Objekten an.

2.3.2. OR-Zugriffsschichten

Als dauerhafter Speicher sind heutzutage relationale Datenbank-Management-Systeme (DBMS) am weitesten verbreitet. In objektorientierten Systemen die ein relationales DBMS verwenden bezeichnet man eine Persistenzschicht auch als objektrelationale Zugriffsschicht (OR-Zugriffsschicht). Die Konzepte von objektorientierten Systemen und relationalen Datenbanken sind von Grund auf verschieden. Zwischen ihnen besteht ein Strukturbruch, den man als impedance mismatch bezeichnet [Starke (2005)]. Ein Objekt hat eine Identität, einen Zustand und ein Verhalten. Ein Objekt mit einer Identität bezeichnet man auch als Entität. Eine relationale Datenbank, deren Grundlage die relationale Algebra ist, kennt diese Eigenschaften nicht. Relationale Datenbanken speichern die Relationen in einer Sammlung von Tabellen und Tupeln (Zeilen). Das Speichern von Objekten in relationalen Datenbanken bringt deshalb eine Reihe von Problemen mit sich. Die Wichtigsten sind im Folgenden aufgelistet.

- Zeilen in einer relationalen Datenbank werden durch einen Primärschlüssel identifiziert, sie haben im Gegensatz zu Objekten jedoch keine Identität. Liest man die selbe Zeile aus einer Datenbank, ist es möglich, daraus verschiedene Objekte zu erzeugen [Siedersleben (2004)].
- Relationale Datenbanken kennen keine Vererbung. Klassen müssen deshalb auf Tabellen abgebildet werden [Siedersleben (2004)].

- Aufgrund des fehlenden Vererbungskonzepts sind polymorphe Abfragen in relationalen Datenbanken schwierig umzusetzen [Siedersleben (2004)]. Eine polymorphe Abfrage liegt vor, wenn eine Abfrage auch die Objekte aller Unterklassen zurückgeben soll, die den Abfragekriterien entsprechen.
- Objekte haben eine Reihe von Attributen und Assoziationen zu anderen Objekten. Aus Performanzgründen ist es oft nicht sinnvoll, ein Objekt vollständig zu laden [Siedersleben (2004)].
- Bei der Entwicklung eines neuen Systems muss entschieden werden, ob sich das Datenbankschema an dem Klassenmodell orientiert oder umgekehrt. Wenn eine bereits vorhandene Datenbank verwendet wird, kann das Transformieren von Zeilen zu Objekten schwierig sein [Siedersleben (2004)].

Die Lösung dieser Probleme wird in der Regel in einer Persistenzschicht gekapselt. Dadurch ist es möglich, den Strukturbruch an zentraler Stelle zu behandeln und vor den anderen Systemteilen zu verbergen.

2.3.3. Entwicklung von Persistenzschichten

Persistenzschichten bieten im Allgemeinen zwei Schnittstellen nach außen an, mit denen die Persistenzschicht in den darüber liegenden Schichten verwendet werden kann. Eine Pool-Schnittstelle zum Anlegen und Löschen von Objekten und eine Query-Schnittstelle zum Abfragen von persistenten Objekten [Siedersleben (2004)]. Diese Schnittstellen sollten möglichst viele Details der Implementierung verstecken, um möglichst wenig Abhängigkeiten zwischen der Anwendungslogik und der Persistenzschicht zu haben [Siedersleben (2004), Bauer und King (2006)].

Bei der Implementierung dieser Schnittstellen in der Persistenzschicht kann man auf Middleware zurückgreifen, die dem Entwickler das Verwalten der Objekte im Pool und das Abfragen von Objekten aus der Datenbank abnimmt. Ein bekanntes Beispiel für solch eine Middleware ist das Persistenz-Framework Hibernate (URL:Hibernate). Selber baut man eine solche Middleware wegen des hohen Aufwandes üblicherweise nicht [Siedersleben (2004)].

Damit die Objekte einer Anwendung in einer Datenbank gespeichert werden können, muss die Middleware Funktionalität zum Definieren von Abbildungen von Objekten auf Relationen einer Datenbank anbieten. Dies bezeichnet man als Objekt-relationales Mapping (OR-Mapping) [Starke (2005)]. Der Entwickler der Persistenzschicht definiert dieses Mapping in einer Programmiersprache oder über Konfigurationsdateien. Bei Hibernate ist beides möglich.

3. Analyse

In diesem Kapitel werden zunächst drei Werkzeuge für die modellgetriebene Softwareentwicklung vorgestellt und eines von diesen als Basis für den Prototyp ausgewählt, der im Rahmen dieser Arbeit entsteht. Das ausgewählte Werkzeug wird im zweiten Teil dieses Kapitels einem Usability-Test unterzogen, um Schwächen beim Einsatz des Werkzeugs aufzuzeigen. Die Resultate der Analysen und insbesondere der Usability-Tests sind die Grundlage für die Verbesserungsmöglichkeiten, die in Kapitel 4 vorgeschlagen und von einem Prototyp umgesetzt werden.

3.1. Werkzeugauswahl

Dieser Abschnitt gibt einen Überblick über die wichtigsten Funktionen und Merkmale, die Werkzeuge für das MDSD haben. Die drei im Folgenden vorgestellten Werkzeuge wurden wegen ihrer Verbreitung ausgewählt und weil sie unterschiedliche Funktionen besitzen. Der Fokus der Untersuchung liegt insbesondere auf der Bedienung der Werkzeuge, wie modelliert wird und ob bzw. wie der Benutzer von den Werkzeugen bei der Modellierung unterstützt wird.

Ein weiterer Aspekt der Untersuchung war, wie gut sich die Werkzeuge als Basis für den im weiteren Verlauf dieser Arbeit vorgestellten Prototyp eignen. Die Anforderungen dafür sind:

- Die Benutzungsoberfläche der Werkzeuge muss mit Plugins erweiterbar sein.
- Die Konzepte des MDSD müssen, wie in Kapitel 2.1 dargestellt, umgesetzt sein.
- Die Modellierung muss mit der UML möglich sein.
- Die Zielsprache muss eine der gängigen objektorientierten Sprachen Java oder C# sein.

Zum Schluss wird eines der Werkzeuge als Basis für den Prototyp ausgewählt.

3.1.1. AndroMDA

AndroMDA[2] ist ein Open-Source Generator-Framework für die MDA. Es kann über Kommandozeilenbefehle bedient werden oder über eine grafische Benutzungsoberfläche eines Plugins für die Entwicklungsumgebung. Letzteres ist für den Benutzer deutlich komfortabler.

Für die Eclipse Entwicklungsumgebung ist das Android Plugin verfügbar und für Visual Studio das Android/VS Plugin. Die Plugins befinden sich derzeit noch in einem nicht ausgereiften Entwicklungszustand und unterscheiden sich bei den angebotenen Funktionen. Die Unterstützung für den Benutzer beschränkt sich bei beiden auf Konfigurationstätigkeiten und das Anstoßen des Transformationsvorgangs.

Beide Plugins bieten einen Wizard zur Erzeugung eines neuen Projekts. In diesem Wizard werden die grundsätzlichen Einstellungen des Projekts abgefragt (siehe Abbildung 3.1). Negativ an diesen Wizards ist das frühzeitige Festlegen auf die verwendeten Technologien (z.B. das verwendete DBMS).

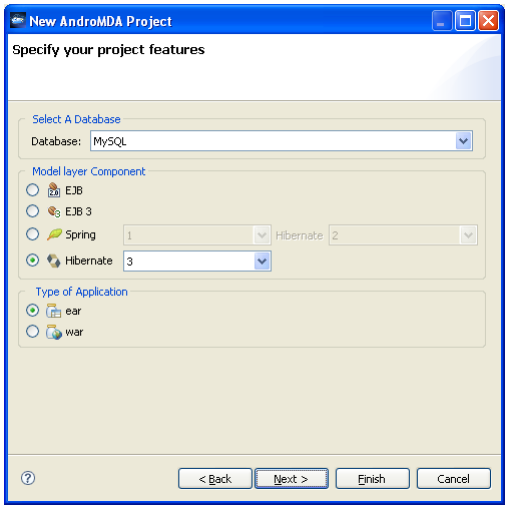


Abbildung 3.1.: Android Projekt-Wizard: Konfiguration der verwendeten Komponenten

Android für Eclipse bietet zudem einen Editor mit grafischer Benutzungsoberfläche für die projektspezifische XML-Konfigurationsdatei für AndroMDA. Mit diesem Editor können unter anderem die Einstellungen der Cartridges vorgenommen werden. Der Benutzer wird allerdings nicht dabei unterstützt, die richtigen Konfigurationswerte zu finden.

Die Modellierung des fachlichen und technischen Modells wird bei AndroMDA mit der UML vorgenommen. Hierzu ist der Modellaustausch mit einem UML-Werkzeug über XMI erforderlich. Der Benutzer wird bei der Modellierung nicht unterstützt. Ob ein entwickeltes Modell

[2] http://www.andromda.org

von den Transformations-Komponenten verarbeitet werden kann, zeigt sich erst während der Transformation.

Nach der Modellierung erfolgt der Transformationsvorgang. Dieser wird durch einen Kommandozeilenbefehl gestartet. In Visual Studio wird dies durch einen Klick auf einen Button vereinfacht. Bei Eclipse ist dies noch nicht implementiert.

Negativ fielen bei AndroMDA die Abhängigkeiten zu Werkzeugen von Drittanbietern und der aufwendige Installationsprozess auf.

3.1.2. objectiF

objectiF[3] ist ein kommerzielles MDA-Werkzeug mit ausschließlich grafischer Benutzungsoberfläche. Es bietet Integration in die Entwicklungsumgebungen Eclipse und Visual Studio. Die Arbeit mit objectiF findet aber in von der Entwicklungsumgebung getrennten Fenstern statt (Abb. 3.2). Die Entwicklungsumgebung dient lediglich zum Vervollständigen des generierten Programmcodes, wobei dies auch über den Editor von objectiF möglich ist.

Bei der Erstellung eines neuen Projekts ist ein Wizard behilflich, in dem eine Projektvorlage ausgewählt werden kann (Abb. 3.3). Die Modellierung mit objectiF wird mittels des integrierten UML Werkzeuges vorgenommen. Der Import von Modellen über XMI ist möglich. Die grafische Benutzungsoberfläche trennt bei der Modellierung explizit zwischen fachlichem Modell, technischem Modell und dem erzeugten System (siehe Abb. 3.2). Der Wechsel zwischen den Modellen geschieht einfach über das Hauptfenster oder Kontextmenüs. Diese klare Trennung der Modelle erleichtert es dem Benutzer, sich zurechtzufinden. Bei herkömmlichen UML-Werkzeugen ist die Trennung der Modelle nicht so offensichtlich. Eine konstruktive Unterstützung des Benutzers bei der Modellierung bietet aber auch dieses Werkzeug nicht. Mit dem Modell-Verifier kann lediglich das fertige Modell ausgewertet werden.

Das Transformieren der Modelle wird in objectiF durch Befehle in den Kontextmenüs ausgeführt.

3.1.3. openArchitectureWare

openArchitectureWare[4] (oAW) ist ein Open-Source MDA-Generator-Framework. oAW ist vollständig in die Eclipse Entwicklungsumgebung integriert (Abbildung 3.4). Andere Entwicklungsumgebungen werden von oAW nicht unterstützt.

[3] http://www.microtool.de/objectiF
[4] http://www.openarchitectureware.org

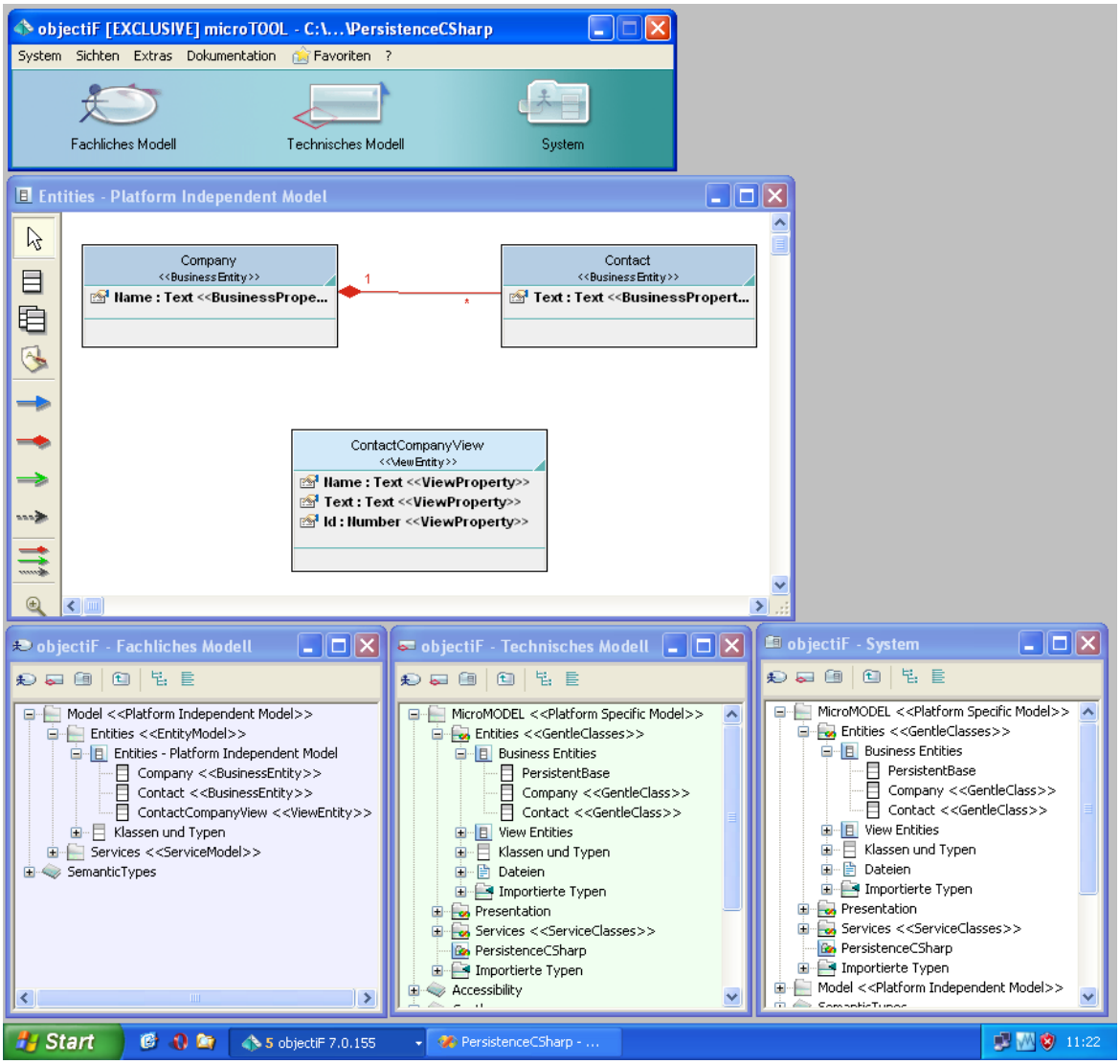


Abbildung 3.2.: objectiF Entwicklungsumgebung

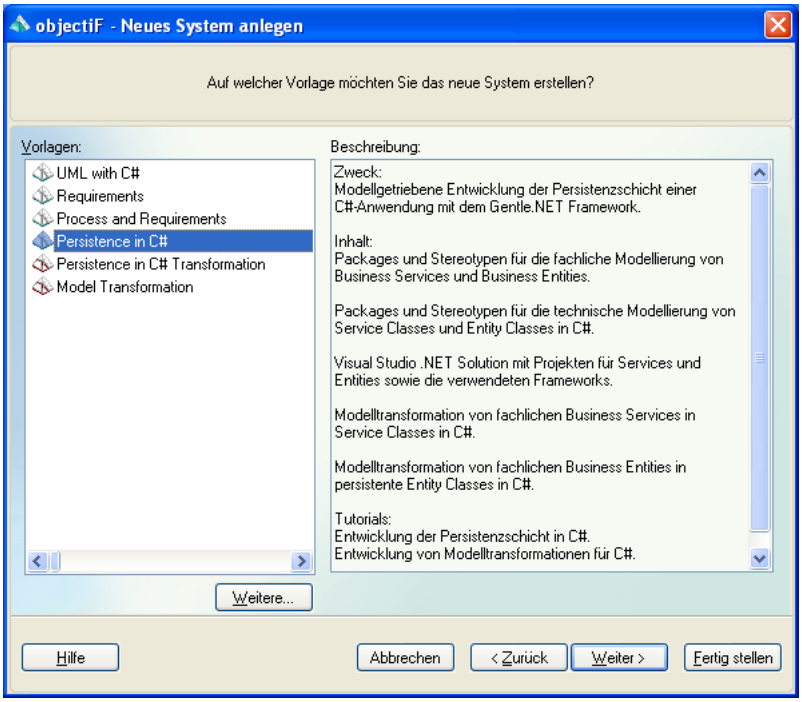


Abbildung 3.3.: objectiF “Neues System Anlegen” Wizard: Seite2

Im Gegensatz zu den zuvor vorgestellten Werkzeugen ist die Modellierung nicht nur mit UML-Metamodellen sondern vorzugsweise auch mit auf Essential MOF (EMOF) basierenden Ecore-Metamodellen. EMOF ist eine Untermenge der Meta Object Facility (MOF), die besonders zur Beschreibung eines Systems geeignet ist, wenn die Implementierungssprache eine objektorientierte Programmiersprache ist [OMG (2006)]. Die Verwendung des Ecore-Metametamodells hat den Vorteil, dass man Metamodelle komplett selbst entwickeln kann. Die Metamodelle enthalten dann nur die zur Domäne passenden Abstraktionen [Stahl u. a. (2007)] und keine Abhängigkeiten zu der UML [Wanner und Siegl (2007)].

Zur Überprüfung von Modellen bietet oAW in der Sprache Check formulierte Constraints an. Darüber hinaus ist es mit sogenannten Recipes möglich, die Implementierungsklassen auf Korrektheit zu überprüfen.

Für die Check-Constraints und Templates bietet oAW textbasierte Editoren an, die in die Eclipse-IDE integriert sind. Diese Editoren bieten eine ähnliche gute Unterstützung, wie man sie beim Schreiben von Quelltext gewohnt ist. So sind z.B. Syntaxhighlighting, Wortvervollständigung und Hervorhebung von Fehlern vorhanden. Die Recipes werden in Java geschrieben. In [Wanner und Siegl (2007)] wird diese Integration als derzeit optimal gelobt. Darüber hinausgehende Unterstützung bietet oAW dem Benutzer allerdings nicht an. Modelle können mit den in Eclipse integrierten Ecore und UML-Editoren oder einem externen

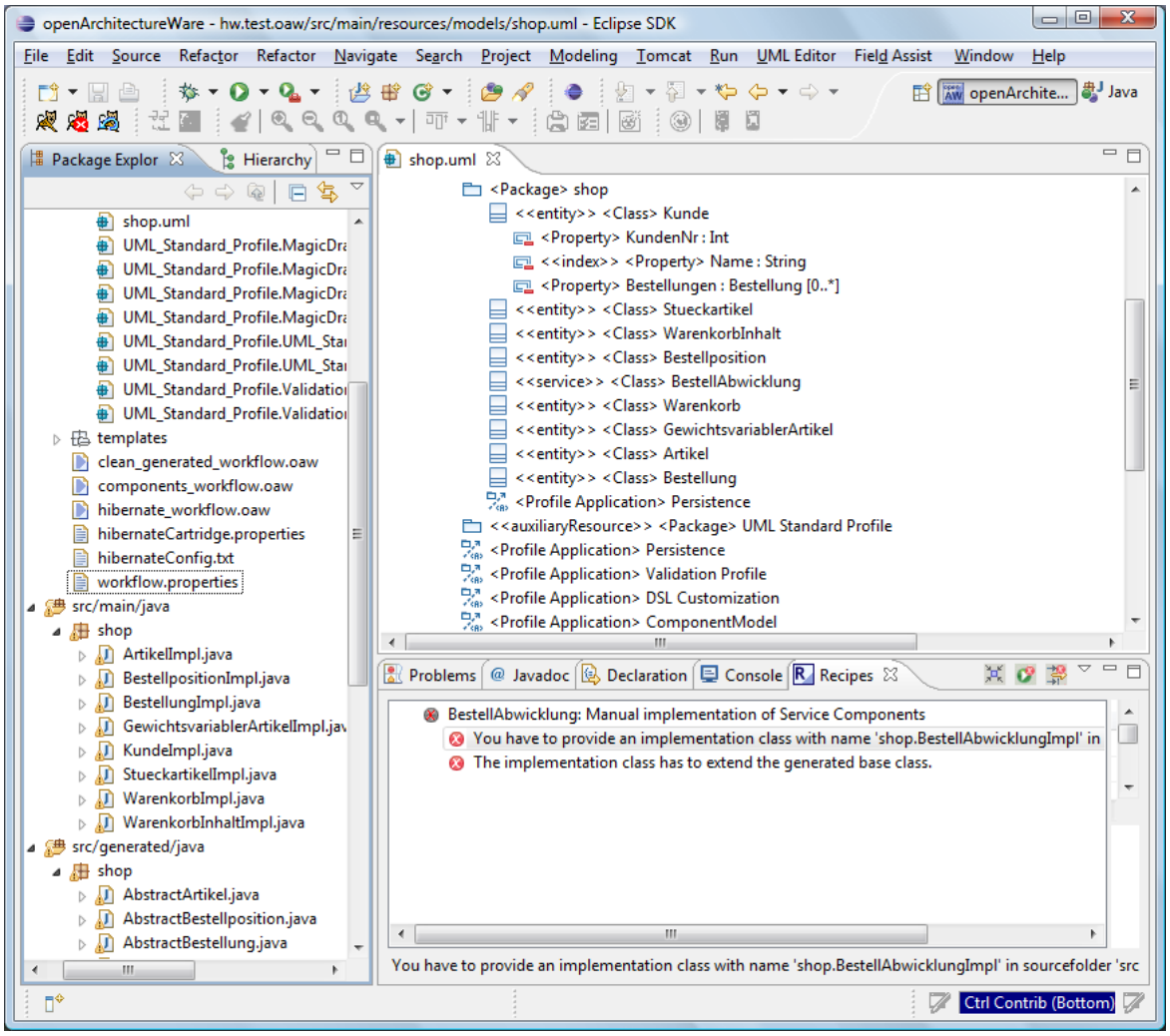


Abbildung 3.4.: openArchitectureWare mit Eclipse

UML-Werkzeug erstellt werden. Der Benutzer wird bei der Modellierung aber nicht von oAW unterstützt. oAW bietet keine Wizards oder Editoren zum Erleichtern der Konfiguration. Die Konfigurationsdateien müssen in einem XML bzw. Texteditor bearbeitet werden.

Der Import von UML-Modellen wird bei oAW vorzugsweise über das Eclipse-UML2[5] Format vorgenommen, es gibt aber auch XMI-Adapter für einige gängige Werkzeuge. Das Eclipse UML2 Format hat gegenüber dem XMI-Format den Vorteil, dass es einheitlich ist.

Mit oAW werden keine Standard Cartridges geliefert. Der Anwender ist deshalb auf Cartridges von Drittanbieter angewiesen oder muss diese selbst entwickeln. Für Metamodelle, die man selbst entwickelt hat, muss man die Modelltransformationen auf jeden Fall selbst erstellen.

3.1.4. Zusammenfassung und Auswahl eines Werkzeugs

Dieser kurze Überblick hat gezeigt, dass keines der Werkzeuge die Entwickler bei der Modellierung unterstützt, wie in Kapitel 1.1 vorgeschlagen. Die Unterstützung für die Entwickler beschränkt sich auf Wizards und Editoren, die bei der Konfiguration helfen sowie Editoren für die Templates. Eine Überprüfung der Modelle findet erst nach der Erstellung eines Modells im Rahmen eines Transformationsschrittes statt. Maßnahmen, die eine fehlerhafte Modellierung verhindern könnten, sind in keinem der vorgestellten Werkzeuge vorhanden. Die wichtigsten Eigenschaften der Werkzeuge sind in Tabelle 3.1 zusammengefasst.

Als Basis für den Prototyp wurde oAW ausgewählt. objectiF ist für den Prototyp nicht geeignet, da die Benutzungsoberfläche nicht wie gefordert mit Plugins erweiterbar ist. Es kann zwar die Entwicklungsumgebung (Eclipse oder Visual Studio), in die objectiF integriert ist, erweitert werden, die Benutzungsoberfläche von objectiF ist davon aber getrennt. Außerdem bietet objectiF keine dokumentierte Programmierschnittstelle (API), sodass der Datenaustausch mit dem Prototyp auf den Import und Export von Modellen im XMI-Format beschränkt wäre. Alle Werkzeuge erfüllen die weiteren genannten Anforderungen. Die Entscheidung zugunsten von oAW viel insbesondere wegen der Reife des Werkzeugs. Die Integration von AndroMDA in Eclipse und Visual Studio hat noch keinen ausgereiften Zustand. AndroMDA hat außerdem mehrere Abhängigkeiten zu Werkzeugen von Drittanbietern und lies sich nur umständlich installieren. openArchitectureWare wird deshalb im weiteren Verlauf dieser Arbeit als Werkzeug für die modellgetriebene Softwareentwicklung verwendet. Im nächsten Unterkapitel wird die Benutzbarkeit von oAW mit einem Usability-Test untersucht.

[5] http://www.eclipse.org/uml2

	AndroMDA	openArchitectureWare	objectiF
IDE Integration	Eclipse, Visual Studio	Eclipse	Eclipse, Visual Studio
Wizard für neue Projekte	✔	-	✔
Unterstützung bei der Konfiguration (Editoren)	✔	-	-
Integrierter UML-Editor	-	-	✔
Unterstützung bei der Modellierung	-	-	-
Frei definierbares Metamodell	-	✔ (Ecore)	-
Metamodell auf UML-Basis (UML-Profile)	✔	✔	✔
Modell zu Modell Transformation	-	✔	-
Modellverifikation	✔	✔	✔
XMI-Import	✔	✔	✔
Generierung für Hibernate	✔	✔[6]	✔
Erweiterbarkeit der Entwicklungsumgebung	✔	✔	-
Ziel Plattform	Java, .Net, u.A.[5]	Java, u.A.[5]	Java, .Net
Ziel Sprache	Java, C#, u.A.[5]	Java, u.A.[5]	C#, C++, Java
Lizenz	Open source	Open source	Kommerziell, Kostenlos mit eingeschränkter Funktionalität
Webseite	www.andromda.org	openarchitectureware.org, www.eclipse.org/gmt/oaw	www.microtool.de/objectiF

Tabelle 3.1.: Werkzeugübersicht

[5] Durch die Verwendung von benutzerdefinierten Templates ist auch die Generierung für andere Plattformen und Programmiersprachen möglich

[6] Bei oAW werden keine Generatoren mitgeliefert. Es wird deshalb ein Cartridge eines Drittanbieters benötigt. Für Hibernate gibt es ein Cartridge bei [URL:FornaxPlatform]

3.2. Usability-Test von openArchitectureWare

Das im vorigen Abschnitt ausgewählte MDA-Werkzeug openArchitectureWare wird in diesem Abschnitt einem Usability-Test unterzogen. Im Folgenden wird zunächst das Testkonzept erläutert. Die Ergebnisse des Tests werden anschließend präsentiert und zum Schluss diskutiert.

3.2.1. Testziel

Das Ziel des Tests ist es, Schwächen in der Benutzbarkeit bei modellgetriebener Softwareentwicklung mit openArchitectureWare aufzuzeigen. Das Augenmerk liegt dabei sowohl auf den speziellen Schwächen von openArchitectureWare als auch Problemen, die bei der modellgetriebenen Softwareentwicklung allgemein auftreten. Die entsprechenden Qualitätskriterien sind nach ISO 9126 die Benutzbarkeit mit den Punkten Verständlichkeit, Erlernbarkeit und Bedienbarkeit sowie die Ergonomie nach ISO 9241.

3.2.2. Testumgebung

Die Usability-Tests fanden im Usability-Labor der Hochschule für Angewandte Wissenschaften Hamburg [URL:HAW-Uselab] statt. Zu der Ausstattung des Labors gehören:

- sechs Kameras, die das Verhalten des Benutzers aufzeichnen,
- Mikrofone, die das laute Denken oder Fragen des Benutzers aufzeichnen,
- ein Eyetracker, der den Blickverlauf des Benutzers aufzeichnet,
- Software zum Aufzeichnen der Benutzereingaben mit Maus und Tastatur,
- Software zum Aufzeichnen des Bildschirminhalts des Testrechners.

Mit diesen Geräten werden die Aktionen und Reaktionen des Benutzers beim Lösen einer vorgegebenen Aufgabe mit einem Computer aufgezeichnet. Mit den aufgezeichneten Daten kann das Testobjekt bewertet werden. Es ist zu betonen, dass das Testobjekt nicht der Benutzer sondern die verwendete Hardware oder Software ist.

3.2.3. Testobjekt

Das Testobjekt ist das in die Eclipse-IDE (Version 3.3) integrierte MDA-Werkzeug openArchitectureWare (Version 4.2) zusammen mit MagicDraw (Version 15) als UML-Werkzeug. openArchitectureWare wurde in Kapitel 3.1 als das geeignete Werkzeug ausgewählt.

3.2.4. Aufgabenstellung

Bei diesem Test bekamen die Testpersonen die Aufgabe, das Persistenzmodell eines fiktiven Onlineshops zu entwickeln und daraus lauffähige Software zu erzeugen. Die Klassen des Onlineshops waren in der Aufgabe vorgegeben. Um die Aufgabe zu lösen, mussten die Testpersonen das Modell um die Persistenzaspekte erweitern. Dazu mussten sie den Klassen im Modell Stereotypen zuweisen und Tagged-Values setzen. Wie die Persistenzaspekte modelliert werden, mussten die Testpersonen anhand der Dokumentation des verwendeten Cartridges für Hibernate herausfinden. Die genaue Aufgabenstellung befindet sich im Anhang A.1 (Seite 73).

3.2.5. Testpersonen und Testablauf

Die Testpersonen waren zwei Studenten aus höheren Semestern des Bachelor-Studiengangs «Angewandte Informatik» und ein Absolvent des Master-Studiengangs «Informatik» an der Hochschule für Angewandte Wissenschaften Hamburg. Mehr Testpersonen wären an sich wünschenswert gewesen, allerdings waren nicht mehr Studenten mit den entsprechenden Voraussetzungen bereit zur Teilnahme. Drei Teilnehmer reichen dennoch aus, um die grundsätzlichen Probleme aufzudecken.

Die Testpersonen gaben alle an, gute UML-Kenntnisse zu haben. Eine Testperson gab an, zu wissen, was MDSD ist. Es hatte aber noch keine Testperson das MDSD angewandt. Deshalb wurde den Testpersonen vor den Tests kurz erklärt, was MDSD ist. Zusätzlich bekamen sie eine Einführung in die Modellierung mit Magic Draw und das Generieren von Code mit oAW. Außerdem hatten die Testpersonen vor Testbeginn Zeit, die Aufgaben durchzulesen und Fragen zu stellen, um sicherzustellen, dass keine grundsätzlichen Verständnisprobleme vorliegen.

Im Anschluss an die Tests fand jeweils eine kurze Nachbesprechung mit der Testperson und dem Testleiter statt.

3.2.6. Testergebnisse

Bei den Usability-Tests wurden eine Reihe von Problemen beobachtet. Die wesentlichen Probleme sind im Folgenden aufgelistet. Im darauf folgenden Abschnitt werden die Ergebnisse diskutiert.

1. **Ergonomie von Magic Draw**
   Die Benutzungsoberfläche von Magic Draw ist an einigen Stellen nicht ergonomisch. Die Testpersonen hatten Schwierigkeiten, in den Menüs und Fenstern die relevanten Funktionen zu finden, mit denen Stereotypen und Tagged-Values gesetzt werden. Zusätzlich wurde die Bedienung des Stereotypen-Menüs durch die hohen Anforderungen an die Motorik bei der Mausbedienung erschwert.

2. **Ergonomie von oAW**
   Die Ergonomie von oAW wird durch die eingeschränkte Erwartungskonformität bei der Behandlung von Fehlermeldungen im «Recipe» Fenster beeinträchtigt. Zwei der Testpersonen versuchten, durch einen Doppelklick auf eine Fehlermeldung zum Ort des Fehlerzustands zu gelangen. Die Testpersonen waren dieses Vorgehen von den Fehlermeldungen im «Problems» Fenster von Eclipse gewöhnt. Das Recipe-Fenster bietet diese Funktionalität jedoch nicht.

3. **Qualität der Dokumentation**
   Die den Testpersonen zur Verfügung gestellte Dokumentation des Hibernate Cartridges beinhaltet alle Informationen, die zur Modellierung des Persistenzmodells gemäß der Aufgabenstellung nötig sind. Dennoch benötigten alle Testpersonen zusätzliche Hilfestellung vom Testleiter, um die Aufgaben zu lösen. Gründe dafür sind, dass die Formulierung der Dokumentation für die Testpersonen nicht immer verständlich war und dass die Dokumentation an einigen Stellen sehr kurz gefasst ist. Ein großes Problem für die Testpersonen war, dass einige Details in der Dokumentation nur indirekt enthalten sind. Um eine Klasse persistent machen zu können, muss diese das Stereotyp «Entity» haben. Die Klassen, die von einer Entität erben, sind nicht automatisch ebenfalls eine Entität. Letzteres steht aber nicht explizit in der Dokumentation.

4. **Vermutungen und falsche Annahmen**
   Alle Testpersonen haben bei mehreren Teilaufgaben nicht zutreffende Vermutungen über die Lösung gemacht. Die Testpersonen haben aber niemals in der Dokumentation nachgesehen, ob die Vermutung zutrifft. Die Lösung der Aufgaben war in diesen Fällen meistens nicht korrekt. Der Fehlerzustand im Modell wurde entweder viel später bei der Modelltransformation oder gar nicht entdeckt, weil keine Fehlerwirkung auftrat.

   Die Aufgabe forderte, dass die Assoziations- und Generalisierungs-Beziehungen zwischen den Klassen im Persistenzmodell korrekt abgebildet werden müssen. Die spontane und korrekte Annahme der Testpersonen war, dass dazu keine Änderungen im

Klassendiagramm nötig sind. Von den Testpersonen wurde dies als selbstverständlich betrachtet, führte aber bei einer Testperson aufgrund der expliziten Forderung in der Aufgabenstellung zu großer Verwirrung. Die Dokumentation erläutert zwar Assoziationen und Generalisierung, die Testpersonen waren sich trotzdem nicht sicher, keine Änderung am Modell vornehmen zu müssen.

5. **Trial and Error**
   Ab und zu haben die Testpersonen versucht, durch Ausprobieren eine Lösung zu finden. Eine der Testpersonen hat immer wieder in der Liste der Stereotypen geblättert, um einen passend erscheinendes Stereotyp zu finden. Beim Definieren eines Index für ein Attribut hat die Testperson zwar das richtige Stereotyp «Index» gefunden, aber sich nicht mit der Dokumentation vergewissert. Die Testperson hat deshalb nicht bemerkt, dass mit einem Tagged-Value ein Name für den Index angegeben werden muss.

6. **Überfliegen und übersehen**
   Beim Lesen der Dokumentation haben die Testpersonen wichtige Details übersehen. Alle Testpersonen haben die Dokumentation nicht vollständig gelesen, sondern nur überflogen. Auch die ihnen wichtig erscheinenden Abschnitte haben sie meistens nicht bis zum Ende gelesen, weil sie davon ausgingen, dass die gelesenen Informationen für die Lösung der Aufgaben ausreichen. Beim Anlegen eines Index haben zwei Testpersonen bis zu der Stelle in der Dokumentation gelesen, an der steht, dass sie ein «Index» Stereotypen verwenden müssen. Dass ein Index einen Namen hat, haben sie nicht mehr gelesen. Ebenso haben alle Testpersonen übersehen, dass künstliche Schlüssel automatisch generiert werden, wenn kein Attribut als Primärschlüssel gekennzeichnet ist.

7. **Überspringen und übersehen**
   Für die Lösung der Aufgaben wichtige Details können auch dann übersehen werden, wenn Aufgaben übersprungen werden. Eine Testperson hatte zunächst keinen Ansatz, wie eine Klasse als persistierbar gekennzeichnet wird. Sie stellte diese Aufgabe zurück, um zuerst die Leichten zu bearbeiten. Nach Erledigung aller anderen Aufgaben vergaß die Testperson aber noch, die übersprungene Aufgabe zu erledigen. Beim Generieren des Codes viel der Testperson schließlich auf, dass nicht alle Klassen generiert wurden, sie konnte den Grund dafür aber nicht ohne Hilfe des Testleiters finden.

8. **Verwechslung von Stereotypen mit gleichem Namen**
   Wenn verschiedene UML-Profile verwendet werden, die Stereotypen mit gleichem Namen definieren, besteht die Gefahr, dass die Stereotypen verwechselt werden. Beim Setzen des Stereotyps «Entity» wählten zwei der Testpersonen ein falsches Stereotyp mit dem gleichen Namen aus einem Standard UML-Profil aus.

9. **UML-Kenntnisse**
   Die Testpersonen gaben an gute UML-Kenntnisse zu haben. Erweiterungen der UML

mit Stereotypen und Tagged-Values hatten sie bei der Modellierung aber noch nicht verwendet. Vor dem Test wurde ihnen deshalb erklärt, wie das UML-Metamodell mit Stereotypen und Tagged-Values erweitert wird. Dennoch hatten zwei Testpersonen Probleme bei der Verwendung von Stereotypen und Tagged-Values, weil sie sich nicht sicher waren, das Konzept verstanden zu haben. Im Laufe des Tests gewannen die Testpersonen an Sicherheit bei der Verwendung von Stereotypen und Tagged-Values. In der Nachbesprechung gaben die Testpersonen aber an, das Konzept nicht vollständig verstanden zu haben.

10. **Fehlender Überblick**
    Die Tests haben gezeigt, dass die Testpersonen Schwierigkeiten hatten, den Entwicklungsprozess zu überblicken. Dies äußerte sich in folgenden Punkten:

    - Zu beginn der Tests fehlte den Testpersonen ein Ansatz zur Lösung der Aufgaben. Es dauerte mehrere Minuten, bis sich die Testpersonen einen Überblick über das Klassendiagramm (welches sie schon in der Vorbesprechung gesehen und verstanden hatten), die Werkzeuge und insbesondere die Dokumentation verschafft hatten. Erst danach konnten sie damit beginnen, zielgerichtet in der Dokumentation nach Informationen zur Lösung der Aufgaben zu suchen.
    - Einer Testperson war nicht klar, wofür welches Werkzeug zuständig ist. Sie versuchte mit Magic Draw aus dem Modell Code zu generieren.
    - Hinzu kommt, dass die Testpersonen kein Gefühl dafür hatten, wie weit sie von der Lösung der Aufgaben entfernt waren.

11. **Fehlerquelle Verwendung anderer Klassennamen als im Modell**
    Das Hibernate Cartridge generiert für jede im Modell vorhandene Entität ein Interface (Name wie die Entität), eine abstrakte Klasse (Klassenname mit Präfix «Abstract») und eine Implementierungsklasse (Klassenname mit Suffix «Impl»). Der generierte Code entspricht also nicht dem Klassendiagramm. Dies führte dazu, dass alle Testpersonen bei der manuellen Implementierung der «Bestellen» Methode nicht auf Anhieb ein Objekt der richtigen Klasse erzeugten.

3.2.7. Diskussion der Testergebnisse

Die Testergebnisse zeigen, dass viele Faktoren die Testpersonen behindert haben. Die Ursachen sind sowohl Mängel in der Benutzbarkeit der Werkzeuge im Speziellen als auch Probleme, die bei der modellgetriebenen Softwareentwicklung im Allgemeinen auftreten. Zum Teil führte die mangelhafte Benutzbarkeit sogar zu Fehlhandlungen der Benutzer und damit zu Fehlerzuständen im Model.

Mangelhafte Aufgabenangemessenheit der Werkzeuge

Die Testergebnisse zeigen als Ganzes betrachtet, dass sich die Testpersonen weniger damit beschäftigt haben, was sie modellieren, sondern vielmehr, wie sie modellieren. Die Werkzeuge sind damit nicht für die im Usability-Test durchgeführten Aufgaben angemessen.

Anstatt die eigentliche Aufgabe zu lösen, stießen die Testpersonen auf immer weitere Probleme. Ihnen war klar, dass sie ein Persistenzmodell modellieren und was aus diesem hervorgehen soll. Wie sie das mit den vorgegebenen Werkzeugen umsetzen können, mussten sie aber erst einmal herausfinden. Dazu mussten die Testpersonen sich erst mit dem ihnen fremden Metamodell vertraut machen, das in der Dokumentation beschrieben ist. Als zweitens mussten sie herausfinden, wie man das Metamodell mit UML umsetzt. Dann mussten sie noch herausfinden, wie man das mit den Werkzeugen umsetzt. Auf die Benutzbarkeit wirkt sich das negativ aus, denn die Aufmerksamkeit des Benutzers wird von dem eigentlichen Problem immer weiter abgelenkt. Der in [Raskin (2000)] als «locus of attention» bezeichnete Punkt der Aufmerksamkeit ist nicht mehr bei dem, was modelliert wird. Zwar können Benutzer durch wiederholte Anwendung lernen, die Funktionen der Werkzeuge und UML-Konzepte automatisch und unterbewusst anzuwenden, sodass ihre Aufmerksamkeit bei der Aufgabe bleibt [Raskin (2000)]. Wie die Testergebnisse zeigen, wird die Lernphase aber durch die Komplexität der Modellierung erschwert. Außerdem ist es nicht realistisch, dass sich ein Benutzer alle Details merken kann. Auch Experten benutzten eine Dokumentation, um spezielle Details nachzuschauen.

Diese Beobachtungen offenbaren die mangelhafte Aufgabenangemessenheit der Werkzeuge.

Fehlerzustände im Modell werden nicht verhindert

Bei den Tests wurde beobachtet, dass Fehlerzustände erst spät oder gar nicht entdeckt wurden (siehe Punkte 4, 5, 6, 7, 8). Ein Fehlerzustand sollte jedoch möglichst früh gefunden werden. Spät gefundene Fehler können zu Verzögerungen des Projekts und hohen Kosten bei der Behebung des Fehlerzustands führen [Spillner und Linz (2005)]. Die bei den Tests beobachteten Fehler können auch durch Experten gemacht werden.

Die Werkzeuge sind nur von Experten benutzbar

Die Tests haben gezeigt, dass die verwendeten Werkzeuge nicht ergonomisch sind (siehe Punkte 1, 2). Die Werkzeuge sind, wie zuvor erwähnt, nicht ausreichend für die Modellierung mit Metamodellen geeignet und ihre Selbstbeschreibungsfähigkeit ist gering. Nur Experten können diese Werkzeuge erfolgreich benutzen.

Das ist nicht akzeptabel. Eine Benutzungsoberfläche sollte nämlich von jedem, egal ob Anfänger oder Experte, einfach zu bedienen sein. Einfache Aufgaben sollten auch einfach durchzuführen sein [Raskin (2000)]. Dafür gibt es viele Argumente. Ein Argument ist, dass die Benutzbarkeit Einfluss darauf hat, ob ein System von den Benutzern akzeptiert wird. Bei fehlender Benutzerakzeptanz kann die Einführung eines neuen Systems scheitern [Spillner und Linz (2005)]. Dies gilt auch für die Softwareentwicklung und damit auch für das MDSD. Akzeptieren die Entwickler die zur Softwareentwicklung verwendeten Werkzeuge nicht, kann ein Projekt scheitern. Ein weiteres Argument ist die Produktivität. Wenn die verwendeten Entwicklungswerkzeuge die Produktivität des Entwicklungsprozesses negativ beeinflussen, könnte ein anderer Entwicklungsansatz vorzuziehen sein.

3.2.8. Fazit des Usability-Tests

Die Testergebnisse haben gezeigt, dass es Bedarf für Verbesserungen an den Werkzeugen für die modellgetriebene Softwareentwicklung gibt. Die Benutzbarkeit der Werkzeuge ist mangelhaft und es ist schwer, deren Bedienung zu erlernen. Auch wenn man ein Experte ist, ist die modellgetriebene Softwareentwicklung mit den verwendeten Werkzeugen nicht einfach, da von den Benutzern sehr viel gefordert wird. Die Entwickler müssen das System, das sie entwickeln, verstehen. Das ist in der Regel schon komplex. Hinzu kommt, dass sie sehr gute Kenntnisse der MDSD-Konzepte haben müssen. Für die Modellierung sind insbesondere UML-Kenntnisse nötig, die die Testpersonen in ihrem Informatik Studium nicht gelernt hatten. Ebenso müssen sich die Benutzer sehr gut mit allen verwendeten Werkzeugen auskennen. Die Konzepte des MDSD ermöglichen es zwar ein System anwendungsnah zu entwickeln, die Werkzeuge unterstützen dies aber nicht angemessen. Statt dessen bringen sie zusätzliche Probleme mit sich, die von der Entwicklung ablenken. Die Kriterien Verständlichkeit, Erlernbarkeit und Bedienbarkeit des Qualitätskritereriums Benutzbarkeit nach ISO 9126 sowie die Ergonomie nach ISO 9241 sind nicht ausreichend erfüllt.

Um die genannten Qualitätskritererien zu erfüllen, müssen zum einen die einzelnen Arbeitsschritte einfacher sein. Zum anderen müssen die Zusammenhänge leichter zu verstehen sein und die Modellierung leichter zu erlernen sein. Außerdem sollten Fehler im Modell möglichst verhindert oder zumindest schneller gefunden werden können. Wie das umgesetzt werden kann, zeigt der im nächsten Kapitel vorgestellte Prototyp.

4. Prototyp eines Assistenten für die modellgetriebene Softwareentwicklung

Der Usability-Test in Kapitel 3.2 hat gezeigt, dass beim MDSD Probleme auftreten können. Dieses Kapitel erläutert, wie ein Teil der Probleme mit einem Wizard gelöst werden kann. Dazu wird zunächst das Konzept des Wizards vorgestellt und anschließend die Realisierung. Der letzte Abschnitt dieses Kapitels stellt als exemplarisches Beispiel einen Wizard vor, der den Benutzer beim Erstellen von Persistenzmodellen unterstützt.

4.1. Ziele und Motivation

Ein zentrales Problem bei der modellgetriebenen Softwareentwicklung ist die Komplexität zwischen den Modellen, die der Entwickler in den Griff bekommen muss. Dies wurde in [Hailpern und Tarr (2006)] kritisiert (siehe Kapitel 2.1.4) und hat sich auch im Usability-Test (Kapitel 3.2) gezeigt. Ein Entwickler benötigt detaillierte Kenntnisse und ein tief greifendes Verständnis über alle Metamodelle, die er verwendet. Die Erfüllung dieser Anforderungen an den Entwickler ist nicht selbstverständlich, da er in der Regel nicht alle Modelle selbst erstellt. Ganz im Gegenteil ist es bei der Softwareentwicklung üblich, große Projekte in kleine und handhabbare Unterprojekte aufzuteilen und Komponenten wiederzuverwenden. Die Vorteile dieser Arbeitsteilung gehen jedoch verloren, wenn die verborgenen Details bei der (Wieder-)Verwendung von Modellen wieder von Bedeutung sind. Metamodelle, die wiederverwendet werden, sind zum Beispiel das UML-Metamodell und das Persistenz-Metamodell, welches beim Usability-Test in Kapitel 3.2 verwendet wurde.

Das Ziel des im Folgenden vorgestellten Wizard-Konzepts ist es, dieses Problem zu lösen. Ein Entwickler soll bei der Verwendung eines Modells kein Experte des Modells sein müssen. Die Kenntnisse, die er über das Modell haben muss, sollen minimal sein.

4.2. Rollen bei der Modellierung

Bei modellgetriebener Softwareentwicklung arbeiten die Entwickler in verschiedenen Rollen mit den Modellen. Die Rollen der Entwickler müssen bei der Entwicklung von Assistenten beachtet werden. Die grundsätzlichen Rollen sind Ersteller, Dokumentierer oder Validierer. Das Arbeiten mit einem Modell ist immer relativ zu einem Metamodell. Deshalb kann man die Rollen ebenfalls relativ zu einem Metamodell unterscheiden (siehe auch Abb. 4.1):

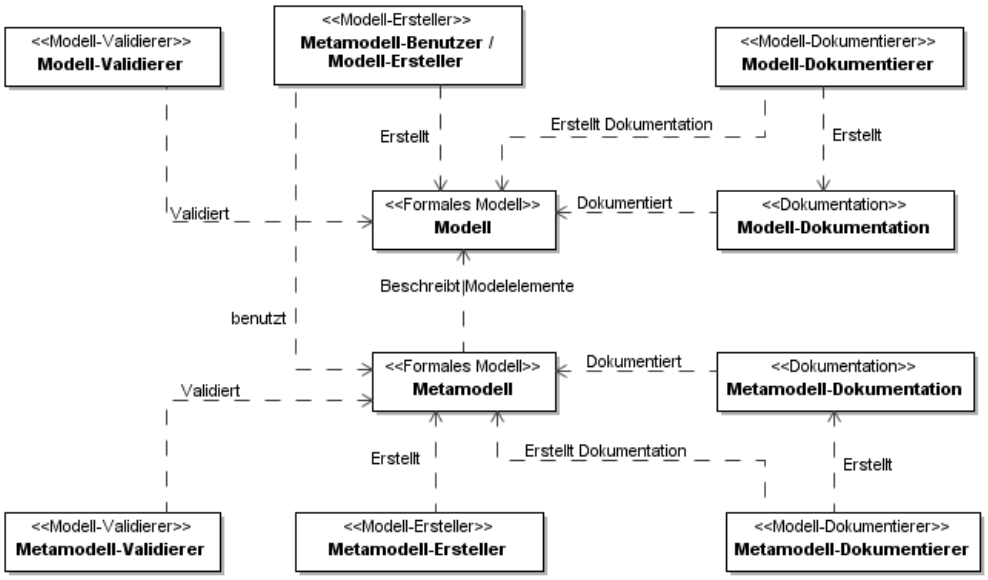


Abbildung 4.1.: Rollen bei der Modellverwendung

- **Modell-Ersteller bzw. Metamodell-Benutzer**
  Ein Entwickler einer Software, der ein Metamodell benutzt, um die Elemente einer Domäne in einem Modell zu beschreiben. Das Modell beschreibt die Fachlichkeit einer Anwendung.
- **Modell-Dokumentierer**
  Ein Entwickler, der eine Dokumentation für ein Modell erstellt. Das Modell wurde von den Modell-Erstellern erstellt. Die Dokumentation ermöglicht die weitere Verwendung und Wartung des Modells und kann als Testbasis bei der Qualitätssicherung dienen.
  Wenn die Dokumentation ein formales Modell ist (z. B. in Form von XML oder ähnlich wie JavaDoc), ist ein Modell-Dokumentierer ein Modell-Ersteller eines Dokumentations-Modells.
- **Metamodell-Ersteller**
  Ein Entwickler, der eine Domäne analysiert und für diese ein Metamodell erstellt.

- **Metamodell-Dokumentierer**
  Ein Entwickler, der eine Dokumentation für ein Metamodell erstellt. Seine Aufgaben sind die eines Modell-Dokumentieres, nur das es hier um ein Metamodell geht. Er erstellt eine Dokumentation, damit das Metamodell von den Metamodell-Benutzern verstanden und verwendet werden kann.
- **Metamodell-Validierer**
  Eine Person, die ein Metamodell auf Korrektheit und Vollständigkeit überprüft. Diese Rolle kann sowohl von einem Entwickler als auch von dem Benutzer der zu erstellenden Software eingenommen werden.
- **Modell-Validierer**
  Eine Person, die überprüft, ob ein von einem Modell-Ersteller erstelltes Modell korrekt ist. Diese Rolle kann von einem Entwickler oder dem Benutzer bzw. Auftraggeber der zu erstellenden Software eingenommen werden.

Die weiteren Rollen, die bei der Entwicklung einer Software vorkommen können, sind an dieser Stelle nicht von Bedeutung.

4.3. Konzept eines Assistenten in Form von Wizards

Um das zuvor erwähnte Problem zu lösen, wird in dieser Arbeit ein Assistent in Form von Wizards (siehe Kapitel 2.2.4) verwendet. Die Wizards sollen die Benutzer (bzw. die Modell-Ersteller, s. o.) durch die Modellierungstätigkeiten führen. Das Ergebnis der Wizards sollen fertige Modelle sein, welche für eine Modelltransformation verwendet werden können. Ein mit einem Wizard erstelltes Modell entspricht dem Modell, welches man mit einem herkömmlichen Modellierungswerkzeug erhalten würde.

Ein solcher Wizard soll dem Benutzer die Semantik eines Metamodells und seiner Elemente auf verständliche Weise präsentieren und erläutern. Bei der Verwendung eines klassischen Modellierungswerkzeugs wird die Semantik eines Metamodells nur durch die Dokumentation des Metamodells deutlich. Der Wizard kann hingegen die Semantik dem Benutzer in der Benutzungsoberfläche deutlich machen. Damit soll es dem Benutzer erleichtert werden, Entscheidungen bei der Modellierung zu treffen. Eine Dokumentation des Metamodells soll bei der Verwendung des Wizards nicht nötig sein. Sie soll von dem Wizard möglichst vollständig ersetzt werden. Das heißt jedoch nicht, dass das Metamodell undokumentiert sein darf.

Falls zwischen Metamodellelementen Abhängigkeiten bestehen, hilft der Wizard, sie in der richtigen Reihenfolge zu verwenden. Der Wizard kann dadurch dem Entwickler alle Modellierungsmöglichkeiten zeigen.

Eine fachlich falsche Modellierung kann der Wizard zwar nicht verhindern, aber die fehlerhafte Verwendung des Metamodells kann vermieden werden. Der Wizard fördert das Verständnis und erleichtert damit die Wahl der richtigen Metamodell-Elemente. Außerdem kann der Wizard vorbeugen, dass der Benutzer bei der Modellierung wichtige Details vergisst. Durch den schrittweisen Ablauf des Wizards bekommt der Benutzer einen Überblick über die möglichen und unbedingt nötigen Optionen. Falls der Benutzer in einem Schritt etwas vergisst, kann der Wizard den Benutzer sofort darauf aufmerksam machen.

4.3.1. Aufgaben und Schritte

Die Erstellung eines Modells hat einen formalisierbaren Ablauf. Es werden bestimmte Aufgaben mit einer Reihe von Schritten durchgeführt und das Modell so um Informationen angereichert. Falls zwischen Metamodellelementen Abhängigkeiten bestehen, bestehen auch Abhängigkeiten zwischen den Schritten. Der vorgegebene Ablauf eines Wizards hilft, die Schritte in der richtigen Reihenfolge durchzuführen. Bei der Erstellung eines Persistenzmodells für Hibernate müssen z. B. erst die zu persistierenden Klassen ausgewählt werden, indem man auf sie das Stereotyp «Entity» anwendet. Danach können weitere Einstellungen vorgenommen werden, wie z. B. die Tabellennamen durch das Setzen eines TaggedValue.

Für jede Aufgabe wird ein eigener Wizard erstellt mit Seiten für jeden einzelnen Schritt. Für jede Aufgabe müssen eine Dokumentation und eine Software für die Anwendungslogik des Wizards erstellt werden. Die Software verarbeitet die Dokumentation und bereitet sie für die grafische Benutzungsschnittstelle des Wizards auf.

4.3.2. Mögliche Probleme

Der Wizard verspricht eine leichtere Verwendung von Metamodellen. Dennoch besteht die Gefahr, dass der Einsatz solcher Wizards nicht realisierbar ist. Unverständliche Erklärungen und eine schlechte Qualität der Benutzungsoberfläche können einen Wizard unbenutzbar machen.

Die für einen Wizard benötigte Dokumentation und Software müssen individuell für das Metamodell erstellt werden. Die Erstellung der Dokumentation des Metamodells ist kein zusätzliches Problem. Eine Dokumentation ist zwar nicht einfach zu erstellen, sie sollte aber auch ohne den Wizard erstellt werden. Die Erstellung der Wizardsoftware und insbesondere der grafischen Oberfläche können hingegen einen großen zusätzlichen Aufwand bedeuten. Damit droht die Komplexität nicht gelöst zu werden, sondern nur unhandhabbar in die Erstellung des Wizards verschoben zu werden. Wenn das Erstellen eines Wizards kompliziert und

zeitaufwendig ist, wäre dies ein Nachteil, der die Vorteile überwiegt. Besonders gravierend daran ist, dass das Metamodell während der Erstellung des Wizards noch nicht verwendbar ist. Dadurch kann es zu Leerlaufzeiten im Projekt kommen und damit zu Verzögerungen des Projekts insgesamt.

Hinzu kommt, dass sich auch Metamodelle ändern können und der Wizard dadurch nicht mehr aktuell und einsetzbar sein könnte. Solche Änderungen kommen insbesondere bei iterativen Vorgehensmodellen vor. Es muss deshalb damit gerechnet werden, dass die Phase des Erstellens eines Wizard mehrfach durchlaufen werden muss.

Im folgenden Abschnitt wird ein Ansatz gezeigt, wie diese Probleme gelöst werden können.

4.3.3. Modellgetriebener Wizard

Die zuvor genannten Probleme lassen sich lösen, indem die Wizards modellgetrieben entwickelt werden. Damit das möglich ist, müssen die zur Generierung benötigten Modelle in einer formalen Beschreibung vorliegen (siehe Kapitel 2.1). Zur Generierung eines Wizards werden drei Modelle benötigt: das Metamodell, ein Aufgabenmodell und ein Dokumentationsmodell.

Das Aufgabenmodell definiert die Aufgaben und Schritte, die bei der Verwendung eines bestimmten Metamodells durchgeführt werden können. Die Aufgaben und Schritte des Aufgabenmodells können von einem Wizard repräsentiert werden. Das Dokumentationsmodell dokumentiert die Elemente des Metamodells. Es wird benötigt, um in der Benutzungsoberfläche Informationen zu den Metamodell-Elementen anzeigen zu können.

Das Dokumentationsmodell ist ein Modell, welches das Metamodell formal dokumentiert. Es ist aber nicht mit einem Meta-Metamodell zu verwechseln. Die Elemente des Metamodells werden von dem Dokumentationsmodell referenziert. Das Dokumentationsmodell wird mit einem speziellen Dokumentations-Metamodell beschrieben.

Während der Ausführung ermittelt der Wizard anhand des vorläufig erstellten Modells, welche Handlungsmöglichkeiten in den Schritten verfügbar sind und dem Benutzer angeboten werden.

Wegen der modellgetriebenen Entwicklung der Wizards sind Änderungen an den Metamodellen kein Problem. Bei einer Änderung muss nur die Dokumentation der geänderten Metamodell-Elemente angepasst werden und der Wizard anschließend neu generiert werden.

Um die Erstellung der grafischen Benutzungsoberfläche kümmern sich generische Wizard- und Aufgaben-Komponenten. Diese bieten für alle Anwendungsfälle fertige Bausteine, die

bei der Generierung eines Wizards zusammengesetzt werden. Auch die Komplexität, die in der Anwendungslogik der Wizards steckt, wird von den Wizard- und Aufgaben-Komponenten bewältigt. Die Entwickler Metamodell-Ersteller und Metamodell-Dokumentierer können sich damit auf die wesentlichen Dinge konzentrieren: das Metamodell und dessen Dokumentation.

Aufgaben-Modell und Aufgaben-Metamodell

Die Semantik des Aufgaben-Modells ist, welche Aufgaben und Schritte durchgeführt werden können, wenn ein bestimmtes Metamodell bei der Modellierung verwendet wird. Das Aufgaben-Modell wird mit dem Aufgaben-Metamodell beschrieben.

Das Aufgaben-Metamodell enthält Metaklassen für Aufgaben und Schritte sowie deren Dokumentation. Für jede Art von Schritt gibt es eine eigene Metaklasse. Die genaue Beschreibung der Elemente des Aufgaben-Metamodells befindet sich im Anhang (Abschnitt C.1.1, Seite 84). Das Aufgaben-Modell wird von einem Metamodell-Dokumentierer erstellt.

Ein Benutzer benötigt beim Ausführen einer Aufgabe mit einem Wizard Informationen, die ihn anleiten. Diese Informationen werden aus der Dokumentation der Aufgabe und der zugehörigen Schritte gewonnen. Die Dokumentation der Aufgaben und Schritte ist eine Subdomäne des Aufgaben-Modells und ist bewusst von den Aufgaben und Schritten selbst getrennt. Sie hätte auch in Attributen der Aufgaben und Schritte gespeichert werden können, dann würde aber der Dokumentations-Aspekt nicht deutlich.

Die Dokumentation des Aufgaben-Modells ist eigentlich nur für die Verwendung im Wizard vorgesehen. Sie kann aber auch verwendet werden, um daraus eine Dokumentation für die Benutzer des Wizards zu generieren (z.B. Online-Hilfe o. ä.).

Dokumentations-Modell und Dokumentations-Metamodell

Der Wizard manipuliert ein Modell, das mit einem Metamodell beschrieben ist. Damit der Wizard die Elemente des Metamodells dem Benutzer verständlich präsentiert kann, müssen auch alle Elemente des Metamodells dokumentiert werden.

Die UML bietet die Möglichkeit, die Elemente eines Modells mit Kommentaren zu versehen. Dies reicht aber nicht aus, um ein Metamodell so zu dokumentieren, wie von einem Wizard benötigt. Der Grund dafür ist, dass die UML-Kommentare erstens keine Semantik haben und zweitens nicht strukturiert sind. Deshalb ist ein spezielles Metamodell notwendig, mit dem die Elemente eines Modells dokumentiert werden können: das Dokumentations-Metamodell.

Mit dem Dokumentations-Metamodell kann ein Metamodell-Dokumentierer ein Dokumentations-Modell für ein Metamodell erstellen. Ein solches Dokumentations-Modell ist geeignet, um daraus die vom Wizard benötigten Informationen zu sammeln.

Wie beim Aufgaben-Modell ist es denkbar diese Dokumentation auch für andere Zwecke zu verwenden als für Wizards. Aus dem Dokumentations-Modell könnte auch eine Dokumentation für die Entwickler generiert werden (z.B. im HTML Format, JavaDoc o. ä.).

Die genaue Beschreibung der Elemente des Dokumentations-Metamodells befindet sich im Anhang (C.1.3, Seite 90).

4.3.4. Metawizard

Das Aufgaben-Modell kann mit einem herkömmlichen Modellierungswerkzeug erstellt werden. Effizienter lässt sich das Aufgaben-Modell aber mit einem Wizard erstellen. Ein solcher Wizard kann abschließend aus dem Aufgaben-Modell einen neuen Wizard generieren und ist damit ein Metawizard. Der Metawizard unterstützt die Rolle der Metamodell-Dokumentierer.

In dem Metawizard dient das Aufgaben-Metamodell als Metamodell des zu erzeugenden Aufgaben-Modells. Bis auf diesen Sonderfall unterscheidet sich der Metawizard konzeptionell nicht von einem anderen Wizard.

4.3.5. Dokumentationswizard

Analog zum Metawizard kann das Dokumentationsmodell mit einem Wizard erstellt werden: mit dem Dokumentationswizard. Der Dokumentationswizard unterstützt die Metamodell-Dokumentierer bzw. Modell-Dokumentierer bei der Erstellung der Dokumentation.

4.3.6. Unterstützte Rollen

Das Ziel des Wizards ist im Wesentlichen die Unterstützung der Modell-Ersteller (siehe Abschnitt 4.2). Da ein Metamodell ebenfalls ein formales Modell ist, kann der Wizard genauso die Metamodell-Ersteller unterstützen. Wenn die Dokumentation, die ein Modell-Dokumentierer erstellt, ein formales Modell ist, kann das Erstellen der Dokumentation vom Dokumentationswizard unterstützt werden.

Metamodell-Dokumentierer können hier auch als Wizard-Ersteller betrachtet werden. Sie können, wie in Abschnitt 4.3.4 erläutert, ebenfalls bei der Erstellung eines Wizards unterstützt werden.

Die Rollen Modell- und Metamodell-Validierer werden vom Wizard nicht unterstützt, da sie kein Modell erstellen.

4.3.7. Ausgeräumte Ursachen für Fehlhandlungen

Der Assistent ermöglicht mit Wizards, viele der im Usability-Test beobachteten Fehlhandlungen zu vermeiden:

- Ein Vorgehen nach dem Trial and Error Prinzip soll durch den vorgegebenen Ablauf verhindert werden. Außerdem fördern die Erläuterungen das Verständnis.
- Der Benutzer wird nicht mit den Details des verwendeten Metamodells überfordert. Es werden keine Vorkenntnisse über das Metamodell benötigt.
- Falsche Vermutungen der Benutzer bzw. Entwickler sollen durch die Erläuterungen verhindert werden.
- Der Wizard überprüft während der Ausführung das erstellte Modell. Der Wizard kann dadurch verhindern, dass der Benutzer bei der Modellierung etwas vergisst. Der Wizard kann aber nur das überprüfen, was aus dem Metamodell hervorgeht. Er kann überprüfen, ob alle benötigten Attribute gesetzt sind und, ob die Constraints des Metamodells eingehalten werden. Er kann aber nicht überprüfen, ob alle funktionalen Anforderungen umgesetzt sind. Die Überprüfung erfolgt zum frühestmöglichen Zeitpunkt und nicht erst bei der Modelltransformation.
- Allgemein kann die Verwechslung von Metamodell-Elementen verhindert werden. Bei Verwendung der UML ist die Verwechslung von Stereotypen ausgeschlossen, da nur das relevante UML-Profil vom Wizard verwendet wird.
- Der Benutzer kann überblicken, was er mit dem Wizard modellieren kann. Keine Optionen sind versteckt.
- Es werden weniger UML-Kenntnisse gefordert. Das Konzept von Profilen, Stereotypen und Tagged-Values muss der Benutzer nicht kennen.
- In einer Dokumentation können Informationen nur indirekt enthalten sein und damit nicht gefunden werden. Bei der Entwicklung des Wizards wird der Entwickler des Wizards stärker gezwungen, alle nötigen Informationen direkt verfügbar zu machen. Beim Ausführen des Wizards fallen fehlende Informationen eher auf, als beim Lesen der Dokumentation.

Inwieweit diese erhofften Verbesserungen tatsächlich zum Tragen kommen, überprüft der zweite Usability-Test (siehe Kapitel 5).

4.3.8. Alternative Formen von Assistenten

Die vorgeschlagenen Wizards sind nicht die einzig mögliche Form, die Entwickler beim MDSD zu unterstützen. Folgende Assistenten sind alternativ oder zusätzlich vorstellbar:

- Unterstützung in einer grafischen Benutzungsoberfläche, die ein Diagramm eines Modells anzeigt.
  - Ein Dokumentationsmodell (s. o.) kann verwendet werden, um Informationen zu dem Metamodell und den Metamodell-Elementen anzuzeigen. Dies fördert das Verständnis und erleichtert die Verwendung des Metamodells. Eine Möglichkeit die Informationen zu präsentieren sind Tooltips. Tooltips haben aber den Nachteil, dass sie nicht ständig sichtbar sind.
  - Anzeige von Aktionen, die für Modell-Elemente ausgeführt werden können. Die Aktionen könnten über ein Kontextmenü aufgerufen werden. Besser wäre es aber, wenn diese direkt sichtbar sind.
  - Das Aufgabenmodell kann zusammen mit der Auswahl im Diagramm verwendet werden, um die für die Auswahl verfügbaren Aktionen zu ermitteln. Eine Aktion kann einem Schritt entsprechen.
  - Hervorhebung von nicht gesetzten Attributen.
- Werkzeuge, die Modelle verwalten.
- Assistenten, die den Entwicklern erleichtern, sich einen Überblick über die Modelle eines Softwaresystems zu schaffen und die Zusammenhänge zu verstehen. Die Entwickler sollen kein Expertenwissen über jede Domäne benötigen. Dies wurde in Hailpern und Tarr (2006) als Problem erkannt (siehe Kapitel 2.1.4).
- Verwendung der Dokumente, aus denen die funktionalen Anforderungen hervorgehen, um bei der Modellierung die Umsetzung der Anforderungen überprüfen zu können. Dafür müssen die Anforderungsdokumente ein formales Modell sein. Die Elemente eines Modells könnten mit den zugehörigen Anforderungen in Verbindung gebracht werden. Dadurch wäre eine Rückverfolgbarkeit der Anforderungen möglich. Wenn die modellgetriebene Softwareentwicklung konsequent durchgezogen wird, werden alle funktionalen Anforderungen in einem Modell umgesetzt. Deshalb könnten alle funktionalen Anforderungen überwacht werden.

Ein Assistenz-System kann auch mehrere der genannten Techniken verwenden, um den unterschiedlichen Bedürfnissen aller Benutzer gerecht zu werden.

4.4. Musterarchitektur für ein Assistenzsystem

Dieser Abschnitt stellt eine mögliche Musterarchitektur für ein System vor, das Assistenz bei der modellgetriebenen Softwareentwicklung anbietet. Das System soll die Benutzer wie zuvor dargestellt unterstützen.

Für die Architektur sind drei Anforderungen von besonderer Bedeutung:

[A1] Die Architektur darf die Art, wie Assistenz angeboten wird, nicht einschränken. Die Unterstützung soll auch in anderer Form als Wizards möglich sein (siehe Abschnitt 4.3.8).

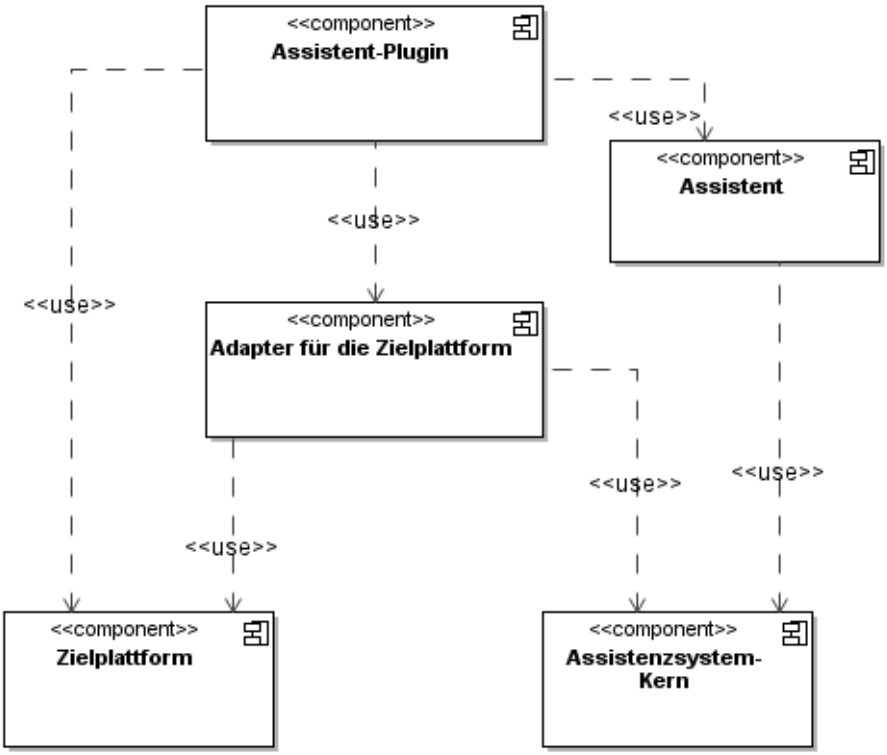
[A2] Die Architektur darf die vom Assistenzsystem unterstützten Entwicklungsumgebungen nicht auf eine bestimmte beschränken. Das System soll für jede beliebige Entwicklungsumgebung angepasst werden können, sofern es die Entwicklungsumgebung technisch zulässt. Es soll damit erreicht werden, dass die Entwickler bei der Wahl der Werkzeuge möglichst keine Einschränkungen haben.

[A3] Die von den Benutzern erstellten Assistenten müssen mit geringem Aufwand erstellt und in die Entwicklungsumgebung integriert werden können (siehe Abschnitt 4.3).

Die hier vorgestellte Musterarchitektur sowie die fachliche und technische Architektur des Prototyps erfüllen diese Anforderungen.

4.4.1. Statische Sicht

Abbildung 4.2 gibt einen Überblick über die Komponenten der Muster-Architektur. Die Komponenten werden im Folgenden erläutert.



Komponentendiagramm 4.2.: Die Komponenten der Musterarchitektur

Zielplattform

Die Zielplattform ist eine Entwicklungsumgebung für die modellgetriebene Softwareentwicklung, in der Assistenten Unterstützung anbieten sollen. Die Zielplattform ist keine Komponente des Assistenz-Systems, sondern vielmehr ein Nachbarsystem. Ein Assistent soll möglichst unabhängig von der verwendeten Entwicklungsumgebung sein, damit er mit geringem Aufwand für jede Entwicklungsumgebung angepasst werden kann.

Assistenzsystem-Kern

Die Anwendungslogik, die allen Assistenten gemein ist, ist im Assistenzsystem-Kern gekapselt. Diese Komponente darf keine Abhängigkeiten zu anderen Komponenten haben, damit sie für jede Zielplattform verwendbar ist. Der Assistenzsystem-Kern beinhaltet den Mechanismus, der ermittelt welche Assistenz einem Benutzer angeboten werden kann. Außerdem besteht er aus Unterkomponenten für die verschiedenen Möglichkeiten, Assistenz anzubieten (z. B. Wizards).

Adapter für die Zielplattform

Mit einem Adapter (siehe [Gamma u. a. (2004)]) für eine Zielplattform werden die Schnittstellen des Assistenzsystem-Kerns für eine bestimmte Entwicklungsumgebung umgesetzt. Die Adapter implementieren die plattformspezifischen Details an zentraler Stelle und können in den Plugins für die Assistenten wiederverwendet werden. Die Adapter implementieren insbesondere die grafische Benutzungsoberfläche der Assistenten und die Schnittstellen für die Modelle. Dabei werden die Bibliotheken der Zielplattform verwendet. Für jede Zielplattform ist ein eigener Adapter notwendig, der wiederum aus mehreren Adaptern für die verschiedenen technischen Schnittstellen der Zielplattform zusammengesetzt ist. Die Adapter für die technischen Schnittstellen können in verschiedenen Zielplattformen wiederverwendet werden.

Diese Adapter ermöglichen es, die Anforderung [A2] zu erfüllen.

Assistent

Die Anwendungslogik eines bestimmten Assistenten ist in einer Komponente gekapselt und verwendet die Schnittstellen des Assistenzsystem-Kerns. Abhängigkeiten zu anderen Komponenten dürfen nicht bestehen, damit der Assistent in jeder beliebigen Entwicklungsumgebung verwendet werden kann. Ein solcher Assistent kann den Benutzer z. B. mit Wizards bei der Modellierung unterstützen.

Assistent Plugin

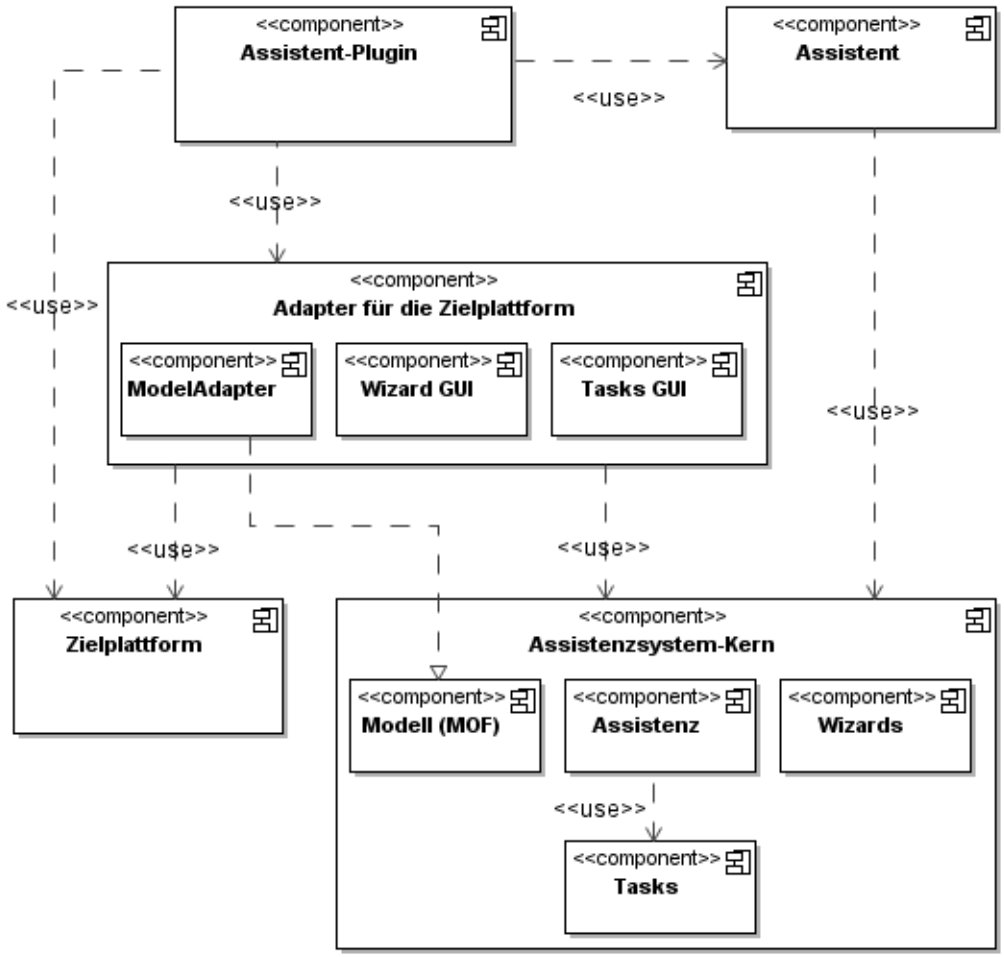
Ein Assistent wird mit einem Plugin in eine bestimmte Entwicklungsumgebung integriert. Das Plugin verwendet den Adapter für die Zielplattform, um den Assistenten für die Zielplattform verfügbar zu machen. Für jede Zielplattform ist ein eigenes Plugin nötig. Bei modellgetriebener Entwicklung können die Plugins aber ohne großen Aufwand automatisch generiert werden.

4.5. Fachliche Architektur des Prototyps

Der Prototyp verwendet die zuvor beschriebene Musterarchitektur, um exemplarisch einen Assistenten für die modellgetriebene Softwareentwicklung mit Wizards zu demonstrieren. Dieser Abschnitt beschreibt die fachliche Architektur des Prototyps. Die technische Umsetzung mit Eclipse und openArchitectureWare wird anschließend beschrieben.

4.5.1. Statische Sicht

Die Komponenten der fachlichen Architektur werden im Folgenden erläutert. Abbildung 4.3 gibt einen Überblick über die Komponenten.



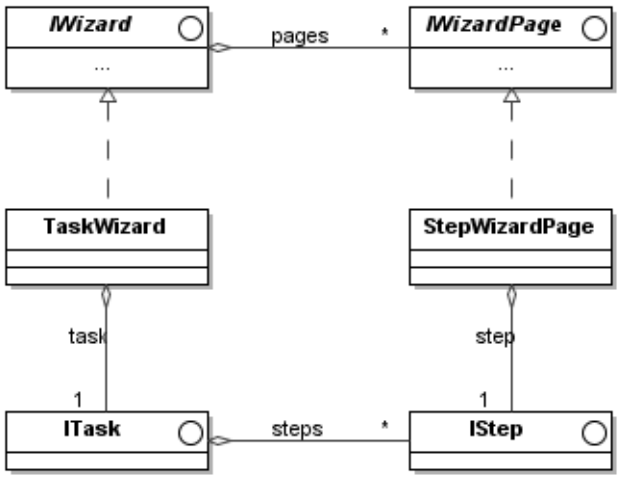
Komponentendiagramm 4.3.: Fachliche Architektur des Prototyps

Assistenzsystem-Kern

Der Kern des Assistenzsystems besteht aus den Komponenten Assistenz, Modell, Tasks und Wizards:

Modell-Komponente Die Modell Komponente stellt Schnittstellen für Modelle bereit. Die Schnittstellen der Modelle entsprechen der MOF-Spezifikation (MOF Spezifikation: siehe [OMG (2006)]).

Wizard-Komponente Die Wizard-Komponente implementiert die grundsätzliche Anwendungslogik eines Wizards und exportiert die Schnittstellen, mit denen ein Wizard verwendet werden kann. Diese Komponente wird von den Assistenten verwendet, welche die Benutzer mit Wizards unterstützen.



Klassendiagramm 4.4.: Zusammenspiel von Wizards und Aufgaben (Task)

Tasks-Komponente Die Tasks-Komponente ist die zentrale Komponente für die Unterstützung der Benutzer bei ihren Modellierungstätigkeiten. Die Tätigkeiten bei der Modellierung sind, wie in Abschnitt 4.3 beschrieben, in Aufgaben und Schritte aufgeteilt.

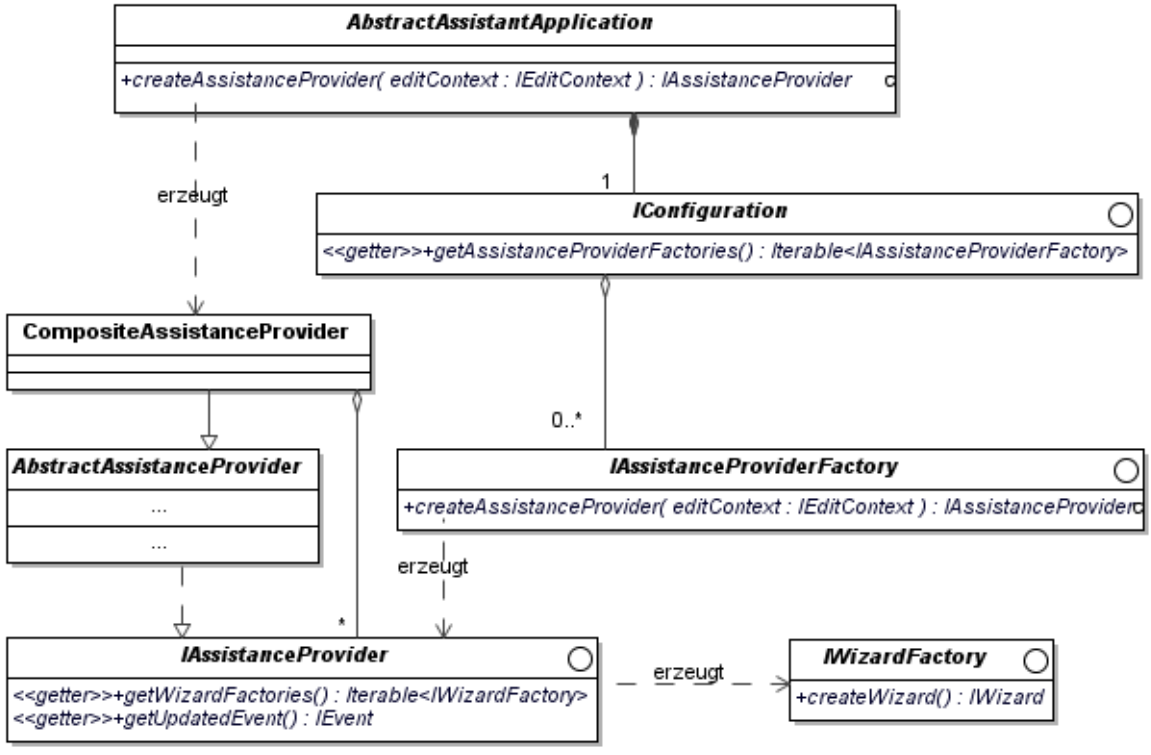
Die Tasks-Komponente definiert verschiedene Klassen von Schritten. Mögliche Schritt-Klassen sind z. B. Schritte, in denen Modellelemente ausgewählt werden, Attributwerte gesetzt werden oder Stereotypen angewandt werden. Ein konkreter Assistent verwendet diese Schritt-Klassen und setzt sie zu einer Aufgabe zusammen. Die Schritt-Klassen korrespondieren mit den Elementen des Aufgaben-Metamodells.

Die von der Tasks-Komponente definierten Schritt-Klassen sind die generischen Bausteine, aus denen eine Aufgabe zusammengesetzt wird. Sie decken möglichst viele Anwendungsfälle bei der Modellierung ab. Der Implementierungsaufwand für die Ersteller eines konkreten Wizards soll durch die Wiederverwendung dieser Schritt-Klassen minimiert werden. Die grundsätzlichen Anwendungsfälle, die bei der Anwendung eines Metamodells vorkommen,

sind bei Verwendung der UML das Setzen von Stereotypen und Tagged-Values sowie die Auswahl von Modell-Elementen. In der Tasks-Komponente sind für diese Anwendungsfälle entsprechende Klassen definiert.

Assistenz-Komponente Die Assistenz-Komponente implementiert den Mechanismus, der die in einem bestimmten Kontext mögliche Assistenz ermittelt. Bei diesem Prototyp beschränkt sich der Mechanismus auf die Ermittlung der verfügbaren Wizards. Abbildung 4.5 zeigt die wichtigsten Klassen und Schnittstellen, die dazu benötigt werden.

Für die Ermittlung der möglichen Assistenz ist ein Objekt mit der `IAssistanceProviderFactory`-Schnittstelle zuständig. Diese Schnittstelle erzeugt für einen bestimmten Kontext ein Objekt, welches die Schnittstelle `IAssistanceProvider` implementiert. Die `getWizards` Methode von `IAssistanceProvider` liefert eine Liste von Fabrik-Klassen (siehe [Gamma u. a. (2004)]), welche die in dem Kontext verfügbaren Wizards erzeugen. Diese beiden Schnittstellen werden von den Assistenten (Komponente «Assistent») implementiert.



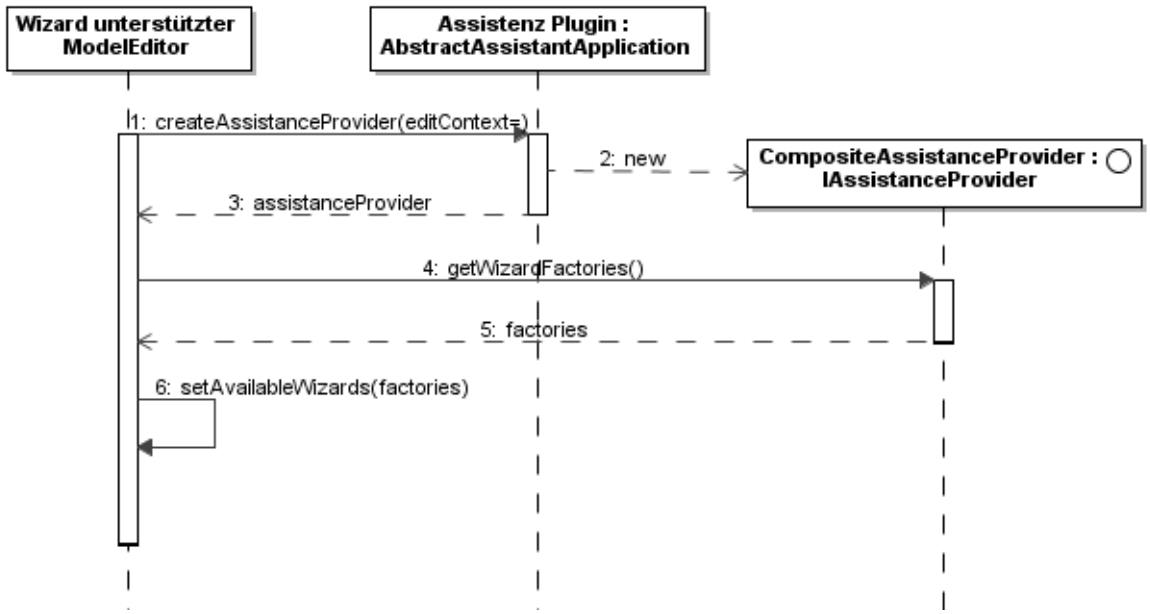
Klassendiagramm 4.5.: Assistenz Komponente

Adapter für die Zielplattform

Wie von der Musterarchitektur vorgesehen, werden die Schnittstellen des Assistenzsystem-Kerns mit einem für Adapter für die Zielplattform an eine Zielplattform angepasst. Der Adapter ist aus mehreren Komponenten zusammengesetzt. Diese Komponenten sind Adapter für die Modelle und die grafische Benutzungsoberfläche von Wizards. Die Modell-Objekte und die Benutzungsoberfläche werden zur Laufzeit mit, von den Adapter-Komponenten bereitgestellten, Fabrikklassen erzeugt.

4.5.2. Dynamische Sicht

Die einem Benutzer angebotene Assistenz wird zur Laufzeit von der Assistenz-Komponente ermittelt. Wenn der Benutzer ein Dokument öffnet, ermittelt der Editor die für den Kontext des Dokuments verfügbaren Assistenten (z. B. in Form von Wizards) (siehe Sequenzdiagramm Abb. 4.6).



Sequenzdiagramm 4.6.: Ermittlung der im Kontext verfügbaren Assistenz

4.6. Technische Architektur des Prototyps (statische Sicht)

Abbildung 4.7 zeigt die statische Sicht auf die Komponenten der technischen Architektur des Prototyps zur Unterstützung der Entwicklung von Persistenzmodellen.

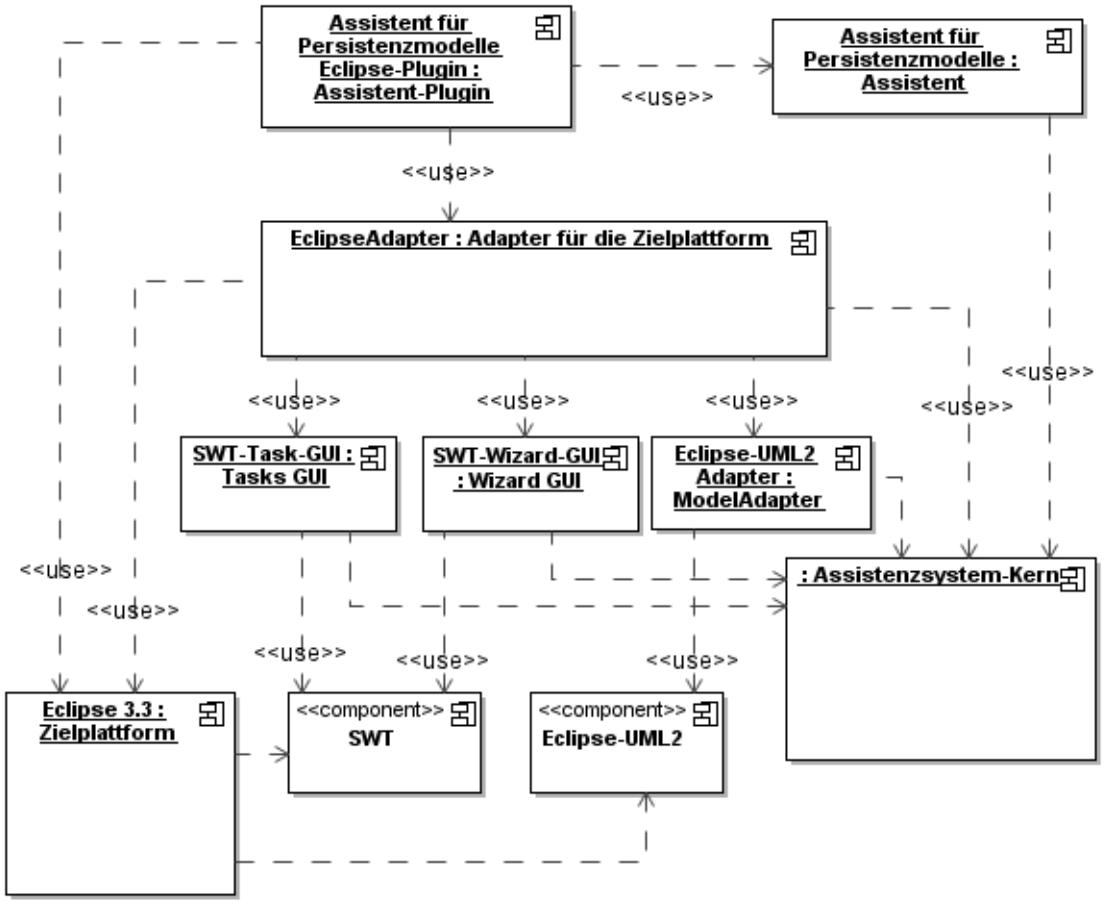


Abbildung 4.7.: Technische Architektur des Prototyps eines Assistenten, der die Entwicklung von Persistenzmodellen unterstützt.

Die konkrete Zielplattform des Prototyps ist die Eclipse 3.3 Entwicklungsumgebung zusammen mit openArchitectureWare (oAW). Die Schnittstellen dieser Zielplattform sind, wie von der fachlichen Architektur vorgesehen, mit Adaptern an die technisch neutralen Schnittstellen des Assistenzsystemkerns angepasst. Die für die Implementierung verwendete Programmiersprache ist Java.

4.7. Der Generator für Assistenten

Um einen Assistenten für die vorgestellte Architektur zu entwickeln, empfiehlt sich eine modellgetriebene Entwicklung. Dafür gibt es zwei Gründe. Erstens kann so der Aufwand ein Plugin für eine bestimmte Entwicklungsumgebung zu entwickeln minimiert werden. Und zweitens muss mit Änderungen an einem Metamodell gerechnet werden, sodass auch Änderungen an einem Assistenten nötig sind (siehe Abschnitt 4.3.2). Bei einem modellgetriebenen Vorgehen sind auch solche Änderungen an einem Assistenten mit geringem Aufwand durchführbar.

4.7.1. Vorgehen bei der Erstellung von Assistenz-Plugins

Ein Assistent und das zugehörige Plugin für eine Entwicklungsumgebung können modellgetrieben entwickelt werden, wie in Abb. 4.8 veranschaulicht. Die Anwendungslogik eines Assistenten wird in einem formalen Modell mit den Mitteln eines speziellen Metamodells beschrieben. Anschließend wird der Assistent in einem weiteren Modell einem Plugin zugeordnet. Danach kann ein Plugin für eine Entwicklungsumgebung generiert werden.

Die für diesen Prototyp verwendeten Metamodelle werden im Anhang erläutert (Abschnitt C.1, ab Seite 84).

4.7.2. Komponenten des Generators

Die Komponenten, welche die Assistenten generieren, können in plattformunabhängige und plattformabhängige Teile aufgeteilt werden (siehe Abb. 4.9).

Die Generierung der Anwendungslogik eines Assistenten erfolgt mit einer plattformunabhängigen Generator-Komponente (Generator-Kern). Diese Komponente ist plattformunabhängig, da auch die Anwendungslogik der generierten Assistenten plattformunabhängig ist.

Ein Plugin, mit dem ein Assistent in eine Entwicklungsumgebung integriert wird, ist hingegen plattformabhängig und muss deshalb für jede Zielplattform speziell generiert werden. Diese Generierung übernimmt ein Generator für eine bestimmte Zielplattform.

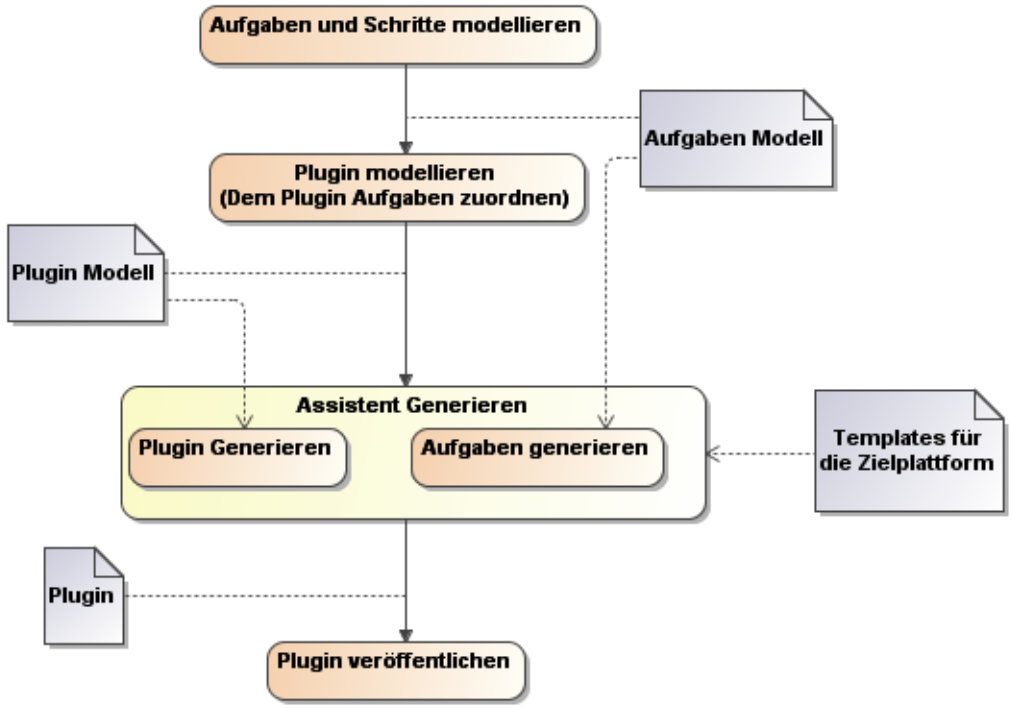


Abbildung 4.8.: Ablauf der Erstellung eines Assistenten

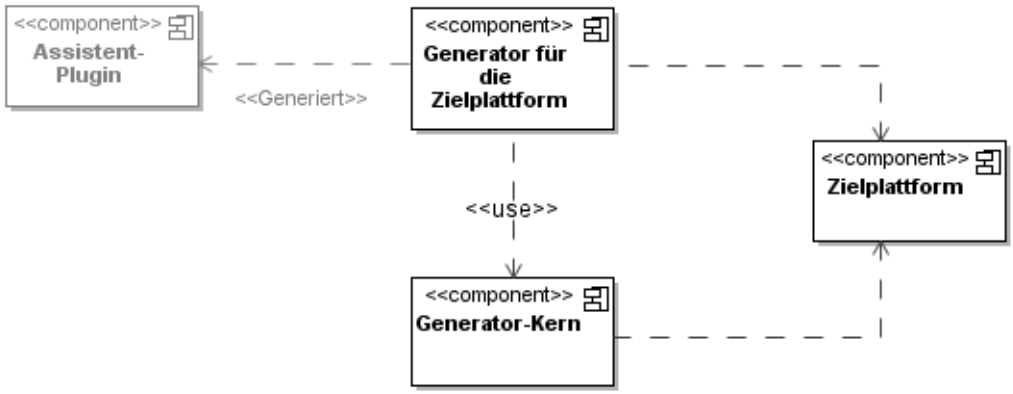


Abbildung 4.9.: Architektur des Generators

4.8. Nutzung des Metawizards

Mit einem Metawizard kann ein Wizard erstellt werden (siehe Abschnitt 4.3.4). Das Ergebnis des Wizards ist ein Modell von Aufgaben und den zugehörigen Schritten. Im letzten Schritt wird daraus ein neuer Wizard als Endprodukt generiert. Der Wizard kann für die selbe Aufgabe wiederholt durchgeführt werden. Dann wird der Wizard bei der Generierung aktualisiert. Der Metawizard des Prototyps hat folgende Schritte:

1. **Start:**
   Eine kurze Einführung in den Wizard.
2. **Aufgabe auswählen:**
   In diesem Schritt wird eine neue Aufgabe erstellt oder eine bestehende Aufgabe ausgewählt. Die Aufgabe wird im weiteren Verlauf des Wizards angepasst.
3. **Aufgabe dokumentieren:**
   Dokumentiert die Aufgabe. Dazu gehören der in der Benutzungsoberfläche angezeigte Titel der Aufgabe und eine Beschreibung der Aufgabe. Die Beschreibung soll den Zweck der Aufgabe den Benutzern deutlich machen.
4. **Schritte auswählen:**
   Erstellt die Schritte der zuvor in «Ausgabe auswählen» ausgewählten Aufgabe.
5. **Schritte dokumentieren:**
   Dokumentiert die Schritte einer Aufgabe. Dazu gehören der in der Benutzungsoberfläche angezeigte Titel des Schritts, eine Beschreibung des Schritts und eine Anweisung, die der Benutzer erhält, wenn der einen Schritt durchführt.
6. **Schritte konfigurieren:**
   In diesem Schritt wird festgelegt, was in einem bestimmten Schritt gemacht werden kann.
7. **Abschluss:**
   Schließt den Metawizard ab. Wenn dieser Schritt erreicht wird, ist das Modell fertig. An dieser Stelle wird aus dem soeben erstellten Modell der Aufgabe ein Wizard generiert. In diesem abschließenden Schritt sind aber noch weitere Modelltransformationen denkbar. Es könnten weitere Assistenten generiert werden, die ebenfalls das Aufgabenmodell verwenden.

Die Abbildungen zu den Schritten des Metawizards befinden sich im Anhang (Abschnitt C.3, ab Seite 95).

4.9. Wizard für Persistenz-Modelle

Den Einsatz von Wizards als Assistenten bei der modellgetriebenen Softwareentwicklung demonstriert der Prototyp mit einem Wizard für die Erstellung eines Persistenzmodells. Das Ergebnis des Wizards ist ein Persistenzmodell. Aus dem Persistenzmodell werden im letzten Schritt des Wizard der Programmcode für die Entitätsklassen und eine Konfigurationsdatei für Hibernate generiert. Mit der Konfigurationsdatei ist die Middleware Hibernate in der Lage, das Objekt-relationale Mapping zwischen den Entitäten des Persistenzmodells und den Tupeln in der relationalen Datenbank durchzuführen.

Der Wizard hat folgende Schritte:

1. **Start:**
   Eine kurze Einführung in den Wizard.
2. **Entität auswählen:**
   In diesem Schritt wird die im weiteren Verlauf des Wizards bearbeitete Entität ausgewählt. Es kann eine neue Entität dem Modell hinzugefügt werden.
3. **Assoziationen und Generalisierung modellieren:**
   In diesem Schritt erhalten die Benutzer einen Hinweis, die Assoziations- und Genereralisierungsbeziehungen wie gewohnt mit den Mitteln der UML zu modellieren. Es wird darauf hingewiesen, dass im Persistenzmodell keine darüberhinaus gehenden Informationen benötigt werden. Der Wizard enthält diesen Hinweis, weil sich die Testpersonen Usability-Test nicht sicher waren, ob sie das Modell mit zusätzlichen Informationen über die Beziehungen zwischen den Entitäten anreichern müssen oder nicht.
4. **Tabelle anpassen:**
   Dieser Schritt passt die Tabelle der Datenbank an, in der die ausgewählte Entität gespeichert wird. Bei diesem Prototyp sind die Einstellungen auf die Angabe des Tabellennamens beschränkt.
5. **Indexe erstellen:**
   In diesem Schritt werden die zu indexierenden Attribute der Entität ausgewählt.
6. **Primärschlüssel auswählen:**
   In diesem Schritt werden die Attribute der Entität ausgewählt, die zum Primärschlüssel gehören.
7. **Abschluss:**
   Schließt den Metawizard ab. Als abschließende Aktion können aus dem erstellten Persistenzmodell Code und eine Konfigurationsdatei für Hibernate generiert werden.

Mit dem Wizard können noch nicht alle Details des Persistenzmodells modelliert werden. Der Prototyp reicht aber aus, um das Prinzip des Wizard zu demonstrieren und die prinzipielle Umsetzbarkeit des Konzepts zu überprüfen. Die zusätzlich benötigten Modellierungsschritte würden nach dem gleichen Prinzip ablaufen, wie die im Prototyp vorhandenen Schritte.

Die Abbildungen zu den Schritten des Wizards befinden sich im Anhang (Abschnitt C.2, ab Seite 90).

5. Usability-Test des Prototyps

Im zweiten Usability-Test wurde die Benutzbarkeit des zuvor vorgestellten Prototyps untersucht. Der erste Abschnitt erläutert das Testkonzept des Usability-Test. Danach werden die Ergebnisse präsentiert und bewertet.

5.1. Testkonzept

5.1.1. Testziel

Dieser Test untersucht primär, ob ein Wizard prinzipiell bei MDSD einsetzbar ist. Es soll untersucht werden, ob die vom Prototyp angestrebten Verbesserungen (siehe 4.3.7) die im ersten Test beobachteten Probleme tatsächlich vermeiden kann. Außerdem sollen mögliche neue Probleme aufgedeckt werden. Das Finden von Schwächen der Benutzungsoberfläche des Prototyps spielt bei diesem Test eine untergeordnete Rolle.

5.1.2. Testumgebung

Die Testläufe fanden wie beim ersten Test im Usability-Labor der Hochschule für Angewandte Wissenschaften Hamburg [URL:HAW-Uselab] statt. Siehe dazu Kapitel 3.2.2.

5.1.3. Testobjekt

Das Testobjekt ist diesmal der in die Eclipse-IDE (Version 3.3) integrierte Prototyp. Speziell wird der Wizard zum Erstellen von Persistenzmodellen getestet.

5.1.4. Aufgabenstellung

Wie beim ersten Test ging es bei dieser Aufgabe um die Erstellung eines Persistenzmodells für einen fiktiven Onlineshop. Als Aufgabe wurde der erste Teil des ersten Usability-Tests übernommen. Der zweite Teil der Aufgabe wurde weggelassen, weil hier das Testobjekt keine Rolle spielt und keine Erkenntnisse für das Testziel zu erwarten wären.

Die genaue Aufgabenstellung befindet sich im Anhang A.2 (Seite 76).

5.1.5. Testpersonen und Testablauf

Die Testpersonen waren auch diesmal drei Studenten der Hochschule für Angewandte Wissenschaften Hamburg. Zwei Teilnehmer befanden sich im sechsten Semester des Bachelor-Studiengangs «Angewandte Informatik». Die dritte Testperson war ein Student des Master-Studiengangs «Informatik». Eine Testperson hatte bereits am ersten Usability-Test teilgenommen.

Im Anschluss an die Tests fand jeweils eine kurze Nachbesprechung mit der Testperson und dem Testleiter statt.

5.2. Testergebnisse

Viele der beim ersten Test aufgetretenen Probleme konnte der Prototyp verhindern. Bei diesem Test sind aber einige neue Probleme aufgetreten. Im Folgenden sind zuerst die beobachteten Probleme aufgelistet. Danach werden die Ergebnisse der beiden Tests verglichen, um Verbesserungen zu ermitteln. Die Ergebnisse werden anschließend in Abschnitt 5.3 diskutiert.

5.2.1. Beobachtete Probleme

1. **Benutzung des Wizards nicht wie vorgesehen**
   Keine Testperson benutzte den Wizard wie vorgesehen. Das Benutzungsmodell des Wizards ging davon aus, dass die Benutzer den Wizard für jede Entität einmal ausführen. Die Schritte des Wizards waren so konzipiert, dass in einem Durchlauf des Wizards alle Einstellungen für eine Entität vorgenommen werden können. Eine wiederholte Durchführung war nur für Änderungen bzw. Korrekturen vorgesehen. Stattdessen führten die Testpersonen den Wizard für jede einzelne Einstellung durch. Die Testpersonen durchliefen die Seiten des Wizards nur zu Beginn der Tests vollständig.

Dabei versuchten sie aber immer, nur eine bestimmte Einstellung vorzunehmen. Die anderen Einstellungen nahmen die Testpersonen zwar wahr, übersprangen sie aber. Sobald die Testpersonen mit dem Wizard vertraut waren, sprangen sie nach jeder einzelnen Einstellung zum Anfang des Wizards zurück bzw. starteten ihn neu. Sie suchten gezielt nach einer Wizardseite mit einer bestimmten Einstellung. Die Testpersonen führten die vom Wizard repräsentierte Aufgabe also nicht Schritt für Schritt durch. Der wesentliche Grund dafür ist, dass die Testpersonen den gleichen Schritt nacheinander für mehrere Entitäten ausführen wollten. Infolgedessen beklagten die Testpersonen die langen Wege im Wizard.

2. **Anweisungen nicht lesen sondern Ausprobieren**
   Alle Testpersonen haben häufig die textuellen Anweisungen des Wizards nicht oder nicht vollständig gelesen. Deshalb konnten sie einige Schritte nicht auf Anhieb erfolgreich bearbeiten. Falls die Testpersonen nicht selbst auf die Lösung kamen, gab ihnen der Testleiter einen Hinweis auf die überlesenen Anweisungen. Daraufhin lasen die Testpersonen die Anweisungen und verstanden meistens, was sie tun mussten.

   Die Testpersonen gaben verschiedene Gründe dafür an, die Anweisungen nicht oder nicht vollständig zu lesen. Zwei Testpersonen gaben an, die in einem Wizard möglichen Einstellungen häufig bzw. generell erst einmal auszuprobieren und die Anweisungen dabei überwiegend nicht zu lesen. Erst wenn sie nicht mehr weiterkommen, lesen sie die Anweisungen. Zu lange Texte nannten zwei Testpersonen als weiteren Grund. Bereits Anweisungen ab zwei Sätzen haben die Testpersonen nicht beachtet, weil sie es zu mühsam fanden, diese durchzulesen.

3. **Anweisungen nicht korrekt verstanden**
   Sofern die Testpersonen die Anweisungen vollständig gelesen haben, haben sie sie meistens korrekt verstanden. Dennoch sind einige Anweisungen missverständlich und nicht deutlich genug.

4. **Hinweis fälschlicherweise als Fehlermeldung gehalten**
   Ein Schritt des Wizards bestand nur aus dem Hinweis, dass die Beziehungen der Klassen im UML-Diagram bearbeitet werden können, aber nicht für das Persistenzmodell angepasst werden müssen. Diesen Hinweis hielt eine Testperson für eine Fehlermeldung. Sie ging davon aus, in einem vorherigen Schritt etwas falsch gemacht zu haben und deshalb in diesem Schritt nichts machen zu können.

5. **Fachliche Konzepte nicht vollständig klar**
   Allen Testpersonen war bekannt, was Indexe und Primärschlüssel sind. Eine Testperson nahm aber fälschlicherweise an, dass ein Primärschlüssel gleichzeitig auch ein Index sein muss. Die Testperson markierte ein Attribut als Index und als Primärschlüssel, obwohl der Index nicht gefordert war.

6. **Probleme durch Mehrfachauswahl**
   Alle Testpersonen hatten große Probleme mit der Wizardseite zum Auswählen einer Entität. Auf dieser Wizardseite wird das im weiteren Verlauf des Wizards bearbeitete Modell-Element ausgewählt. Das Problem dieser Seite ist, dass zwei verschiedene Dinge ausgewählt werden können. Die Elemente eines Modells können als Entität markiert werden (mittels einer Checkbox) und die zu bearbeitende Entität kann ausgewählt werden (mittels eines Radiobuttons). Der Unterschied dieser Auswahlmöglichkeiten war nicht klar.

7. **Grafische Benutzungsoberfläche des Prototyps nicht optimal**
   Die noch nicht ausgereifte grafische Benutzungsoberfläche des Prototyps führte zu einigen Problemen:

   a) Im Wizard kann immer nur ein Modell-Element ausgewählt und bearbeitet werden. Eine Testperson äußerte den Wunsch, mehrere Entitäten gleichzeitig auswählen und bearbeiten zu können.

   b) Die verfügbaren Wizards listet der Prototyp unstrukturiert auf. Den Testpersonen war deshalb nicht klar, dass für sie nur ein Wizard von Bedeutung ist. Dass die anderen Wizards für eine ganz andere Domäne verwendet werden, wussten sie nicht.

   c) Eine Testperson war unsicher, ob die vorgenommenen Einstellungen auch tatsächlich umgesetzt wurden. Die Testperson hatte eine Rückmeldung erwartet.

   d) Der Prototyp verwendet eine Baumstruktur, um das UML-Modell zu darzustellen. Dies trägt nicht zum Verständnis bei.

   e) Die Art und Weise, wie der Prototyp die Fehlermeldungen anzeigt, ist nicht optimal. Eine Testperson empfand es als störend, wenn Hinweise auf nicht getätigte Einstellungen wie Fehlermeldungen angezeigt wurden, bevor Sie die Möglichkeit hatte, die Einstellungen vorzunehmen. Die Fehlermeldungen sollten außerdem näher am Ort der Ursache angezeigt werden. Dies behinderte die Testpersonen aber nur unwesentlich.

5.2.2. Vergleich mit dem ersten Test

Viele der beim ersten Usability-Test beobachteten Probleme sind bei diesem Test nicht aufgetreten. Tabelle 5.1 stellt gegenüber, welche Ursachen für Fehlhandlungen der Wizard ausräumen soll (siehe 4.3.7) und welche Fehlhandlungen beobachtet wurden.

Ursache für Fehlhandlung	**Fehlhandlung ausgeräumt**	**Anmerkung**
Vorgehen nach dem Trial and Error Prinzip	(✔)	Die Testpersonen versuchten nicht mehr durch Ausprobieren herauszufinden, wie etwas modelliert wird. Allerdings versuchten die Testpersonen zum Ziel zu kommen, ohne die Anweisungen vollständig zu lesen (s. o., Punkt 2). Die Testpersonen erkundeten explorativ die Benutzungsoberfläche. Dies ist eine andere Art von Ausprobieren.
Der Benutzer ist mit den Details des verwendeten Metamodells überfordert	✔	Es gab keine Situation, in der die Konzepte des Metamodells zu komplex präsentiert wurden. Verständnisprobleme gab es nur, wenn die Erläuterungen nicht verständlich waren.
Falsche Vermutungen des Benutzers	✔	Es wurden keine Entscheidungen die auf Vermutungen über das Metamodell basieren beobachtet.
Nichteinhaltung von Constraints / Vergessen die benötigten Attribute zu setzen	✔	Alle Testpersonen haben keine im Persistenzmodell benötigten Details vergessen. So bemerkten sie beispielsweise sofort nach dem Anlegen eines Indexes, dass ein Index einen Namen haben muss. Beim ersten Test wurde dies erst bei der Generierung von Code bemerkt. Wie erwartet konnte der Wizard aber nicht verhindern zu vergessen, funktionale Anforderungen umzusetzen (siehe 4.3.7).
Verwechslung von Metamodell-Elementen	✔	Wurde nicht beobachtet.
Benutzer verliert den Überblick über die Modellierungsmöglichkeiten	✔	Den Testpersonen war klar, was sie mit dem Wizard machen können.
Hohe Anforderungen an die UML-Kenntnisse des Benutzers	✔	Die Testpersonen benötigten keine UML-Kenntnisse, um den Wizard zu bedienen.
Benötigte Informationen sind nur indirekt verfügbar.	✔	Alle benötigten Informationen waren in der grafischen Benutzungsoberfläche enthalten.

Tabelle 5.1.: Gegenüberstellung der Ursachen für Fehlhandlung beim ersten Test und der vom Prototyp angestrebten Verbesserungen

Neben der Vermeidung von Fehlhandlungen konnten weitere Verbesserungen festgestellt werden:

1. **Keine Unterbrechung der Arbeit**
   Bei dem ersten Test mussten die Testpersonen sehr häufig ihre Arbeit unterbrechen, um in der Dokumentation nachzulesen. Bei diesem Test gab es nur Unterbrechungen, wenn die Testpersonen die Aufgabenstellung lasen. Der Arbeitsablauf war dadurch flüssiger.

2. **Effizienter**
   Alle aufgezeichneten Metriken haben bessere Werte, als beim ersten Test. Die Metriken sind im Anhang detailliert aufgelistet (Anhang B, Seite 79).

   Insbesondere war die Bearbeitungszeit für die Aufgaben im Schnitt nur noch halb so lang. Die längste Bearbeitungszeit lag noch eine Minute unter der schnellsten Zeit beim ersten Test. Die Testperson, die die längste Bearbeitungszeit benötigte, machte während des Tests die mit Abstand meisten Anmerkungen. Ohne die zahlreichen Anmerkungen wäre der Unterschied zwischen der kürzesten Bearbeitungszeit im ersten Test und der längsten Bearbeitungszeit im zweiten Test noch größer ausgefallen.

   Auch die Metriken der Eingabegeräte waren besser als bei dem ersten Test. Es wurden im Schnitt 42% weniger Mausklicks und 60% weniger Tastaturanschläge benötigt. Die zurückgelegte Mausstrecke sank um knapp ein Viertel. In den Metriken ist mit eingerechnet, dass beim Test des Prototyps der zweite Aufgabenteil wegfiel.

5.3. Diskussion der Testergebnisse

Das Ergebnis des Usability-Tests ist insgesamt positiv. Allerdings hat der Test auch Probleme aufgezeigt. Diese müssen gelöst werden, bevor ein Wizard bei der Modellierung eingesetzt werden kann.

Viele der beobachteten Probleme lagen daran, dass ein Prototyp mit noch unausgereifter Benutzungsoberfläche getestet wurde. Dies gilt für die Punkte 3, 4, und 7 in Abschnitt 5.2.1. Diese Probleme sind aber relativ einfach lösbar. Die grundsätzliche Verwendbarkeit von Wizards gefährden sie nicht.

Allerdings bedarf es besonderer Aufmerksamkeit darauf, dass die Benutzung des Wizards nicht wie geplant verlief (Punkt 1). Dies gefährdet die Verwendbarkeit von Wizards in der Praxis. Die Ursache ist, dass die Schritte des Wizards nicht den Arbeitsablauf der Testpersonen repräsentiert haben. Die Testpersonen konnten ihre Arbeit nicht flexibel genug gestalten, sondern bekamen einen nicht angemessenen Ablauf aufgezwungen. Sie wurden dadurch

bei der Arbeit behindert. Damit ist der Wizard in dieser Form nicht optimal für einen Expertenarbeitsplatz geeignet. Der Wizard muss deshalb so verbessert werden, dass die Arbeitsabläufe flexibel sind und allen Benutzern gerecht werden. Das Schwierige daran ist, allen Benutzern gerecht zu werden. Außerdem besteht zwischen einer flexiblen Arbeitsgestaltung der Entwickler und der Vorgabe eines groben Ablaufs durch einen Wizard ein Widerspruch. Wenn es nicht gelingt, diesen Widerspruch zu umgehen, sind Wizards nicht das optimale Mittel für die Unterstützung der Modellierung.

Der Test hat einige Verbesserungsmöglichkeiten für den Wizard aufgezeigt, die den Entwicklern eine flexiblere Gestaltung der Arbeitsabläufe ermöglichen könnten:

- Flexiblere Navigation durch die Seiten des Wizards.
- Kürzere Wege zu den gewünschten Einstellungen. Nur die vom Benutzer tatsächlich benötigten Wizardseiten sollten durchlaufen werden.
- Enge Verzahnung mit der grafischen Darstellung des Modells (z. B. UML-Diagramm). Die von einem Wizard zu bearbeitenden Elemente könnte der Benutzer über die Auswahl in der grafischen Darstellung des Modells bestimmen.
- Feinere Granularität der Aufgaben. Das bedeutet: Mit einem einzelnen Wizard kann weniger gemacht werden, aber es gibt mehr verschiedene Wizards.
- Umkehren der Auswahl: erst wird ausgewählt, was gemacht werden soll und dann für welche Modell-Elemente. Bisher wurde zuerst das zu bearbeitende Modell-Element ausgewählt. Damit würde auch eine feinere Granularität der Aufgaben einhergehen.

Die Beobachtung, dass die Benutzer nicht alle Anweisungen des Wizards vollständig lesen, kann vermutlich bei Wizards allgemein gemacht werden. Zumindest gaben die Testpersonen an, dies regelmäßig bei anderen Wizards zu machen. Die Verwendbarkeit von Wizards stellt dies deshalb nicht in Frage. Man muss es nur bei der Entwicklung eines Wizards bedenken. Aus der genannten Beobachtung kann man eine für die Entwicklung von Wizards wichtige Erkenntnis ziehen. Die Aufmerksamkeit des Benutzers ist zuerst auf die sichtbaren Einstellungsmöglichkeiten gerichtet. Die einleitende Anweisung einer Wizardseite wird kaum beachtet. Erst bei Problemen liest der Benutzer die Anweisung genauer durch und liest die zusätzlichen Hinweise. Deshalb müssen die Formulierung und Länge der Anweisungen und Hinweise sowie der Aufbau einer Wizardseite diesem Ablauf gerecht werden. Die einleitende Anweisung muss kurz und prägnant sein. Details enthalten nur die Hinweise zu den einzelnen Optionen. Es darf nicht nötig sein, erst eine detaillierte Anweisung zu lesen. Beim im Usability-Test getesteten Wizard war dies der Fall.

Im ersten Usability-Test wurde beobachtet, dass die Testpersonen versuchten, durch Ausprobieren zu einer Lösung zu kommen. Auch bei diesem Test haben die Testpersonen versucht, durch Ausprobieren zu einer Lösung zu kommen. Die Art des Ausprobierens ist hier aber

eine andere (siehe Tabelle 5.1). Beim ersten Test haben die Testpersonen Probleme gehabt, die Konzepte des verwendeten Metamodells zu verstehen und anzuwenden. Die benötigten Informationen konnten sie nicht finden. Deshalb hatten sie als einzigen Ausweg gesehen, die Modellierungsmöglichkeiten auszuprobieren. Beim Test des Prototyps haben die Testpersonen hingegen den Wizard explorativ erkundet. Sie haben sich dadurch einen Überblick über die möglichen Einstellungen gemacht. Das Ausprobieren lag hier also nicht an der Hilflosigkeit der Testpersonen. Das explorative Vorgehen ist keine Folge von Mängeln des Wizards, sondern ist als normal zu beurteilen.

Trotz der neuen Probleme ist das Ergebnis des Tests insgesamt positiv. Alle angestrebten Verbesserungen konnten umgesetzt werden. Die Ursachen für die Fehlhandlungen beim ersten Test konnte der Wizard verhindern (siehe Tabelle 5.1). Beim Test des Prototyps war die Zahl der Probleme insgesamt geringer, und die Probleme waren vor allem weniger gravierend. Die Erstellung eines Modells ist unter den Gegebenheiten des ersten Tests ohne Expertenwissen über das Metamodell kaum möglich. Der Prototyp hingegen fordert gar keine Vorkenntnisse über das Metamodell. Die Wahl der Metamodell-Elemente ist kein Problem, da dies dem Benutzer vom Wizard abgenommen wird. Aus diesen Gründen kann mit einem Wizard ohne Unterbrechungen gearbeitet werden (siehe Abschnitt 5.2.2). Das Arbeiten mit einem Wizard ist deutlich Effizienter, wie die Metriken belegen (siehe Abschnitt 5.2.2).

5.3.1. Konsequenzen für die Weiterentwicklung des Prototyps

Bei einer Weiterentwicklung des Prototyps sollten insbesondere die folgenden Verbesserungsmöglichkeiten umgesetzt werden:

- Aufteilung der Markierung von Klassen als Entität und die Auswahl einer Entität in zwei Schritte.
- Trennung der Aufgabe «Entität Erstellen» in mehrere Aufgaben feinerer Granulatrität (z. B. «Entitäten auswählen», «Primärschlüssel anlegen», «Indexe anlegen» etc.).
- Umformulierung der Anweisungen. Die Anweisungen müssen kürzer sein.
- Zusätzlich anbieten, die zu bearbeitenden Modell-Elemente über das Diagramm auszuwählen.

Anschließend muss in einem Usability-Test untersucht werden, ob nach den Maßnahmen tatsächlich Verbesserungen zu beobachten sind und neue Probleme aufgetreten sind. Zusätzlich zu diesen speziellen Maßnahmen sollten die allgemeinen Empfehlungen des nächsten Abschnitts beachtet werden. Die zuvor genannten Verbesserungsmöglichkeiten sollten aber zuerst umgesetzt werden.

5.3.2. Handlungsempfehlungen für die Entwicklung von Assistenten und Wizards allgemein

Aus den Beobachtungen des Usability-Test können Konsequenzen für die Entwicklung von Assistenten und Wizards allgemein gezogen werden:

- Bei sehr flexiblen Abläufen müssen Alternativen zu Wizards gesucht werden. Wizards sind zu starr, um alle Tätigkeiten unterstützen zu können.
- Ein Wizard muss so flexibel wie möglich zu bedienen sein.
- Bei der Entwicklung eines Assistenten müssen die Arbeitsabläufe der Benutzer detailliert untersucht werden.
- Texte müssen kurz und prägnant formuliert werden. Anweisungen sollten nicht länger als ein Satz sein.
- Anweisungen sollten sparsam eingesetzt werden. Möglichst nur eine Anweisung pro Wizardseite.
- Die zusätzlichen und detaillierten Hinweise müssen für den Benutzer als ergänzende Informationen erkennbar sein. Die Hinweise dürfen nicht im Vordergrund stehen.
- Mehrere Arbeitsschritte dürfen nicht zu einer Wizardseite zusammengefasst werden (z. B. keine Mehrfachauswahl).

Dies sind Empfehlungen, die möglicherweise in einigen Fällen nicht alle umsetzbar sind. Man sollte sich dann aber über die möglichen negativen Auswirkungen bewusst sein und Maßnahmen treffen, um die Auswirkungen frühzeitig zu erkennen und darauf reagieren zu können.

5.3.3. Fazit

Die Vorteile des Wizards überwiegen die beobachteten Probleme. Mit dem Wizard war es möglich, ein korrektes Modell zu erstellen. Außerdem war die Bearbeitungszeit signifikant geringer als beim ersten Test. Der Prototyp ist deshalb eine Verbesserung gegenüber einem herkömmlichen Modellierungswerkzeug. Die Frage, ob ein Wizard prinzipiell bei der modellgetriebenen Softwareentwicklung einsetzbar ist, kann bejaht werden.

5.4. Erkenntnisse aus den Usability-Tests für Usability-Tests

Zum Schluss seien hier noch die wichtigsten Erkenntnisse über die Durchführung von Usability-Tests selbst genannt. Diese Erkenntnisse sollte man bei der Planung von Usability-Tests beachten.

Bei dem ersten Usability-Test wurde bei einer Testperson ein steigendes Gefühl von Unwohlsein beobachtet, weil sie von Anfang an große Probleme mit der Lösung der Aufgaben hatte. Die Aufgabenstellung sollte deshalb zuerst eine einfach lösbare Teilaufgabe vorsehen. Das Erfolgserlebnis hilft der Testperson, ihre Anspannung zu lösen. Dieser psychologische Faktor sollte unbedingt beachtet werden, wenn bei den folgenden Aufgaben mit großen Problemen gerechnet wird.

Eine weitere Erkenntnis ist, dass es schwierig ist, freiwillige Testpersonen für anspruchsvolle Aufgaben zu finden. Nicht nur die Menge der infrage kommenden Personen ist dann kleiner. Auch die Bereitschaft zur Teilnahme am Test ist dann sehr gering. Das gilt umso mehr, wenn keine finanziellen Anreize geboten werden können.

6. Zusammenfassung und Ausblick

Diese Arbeit schlägt die Verwendung von Assistenten für die modellgetriebene Softwareentwicklung (MDSD) vor. Die Assistenten sollen die Entwickler bei der Modellierung unterstützen. Ein erster Usability-Test einer MDSD-Entwicklungsumgebung ohne Assistenten hat den Bedarf dafür eindeutig gezeigt. Keines der untersuchten Produkte bietet den Entwicklern solche Assistenten an.

Besonders anspruchsvoll ist für einen Entwickler die Verwendung eines Metamodells, über das er keine Expertenkenntnisse hat. Expertenkenntnisse sind aber nötig, um mit dem Metamodell ein korrektes Modell zu erstellen. Dabei ist es schwierig, die passenden Metamodellelemente zu finden und keine Details zu vergessen. Infolgedessen werden von den Entwicklern häufig Fehlhandlungen gemacht. Die dadurch im Modell vorhandenen Fehlerzustände werden meistens erst spät gefunden.

Diese Probleme können Assistenten verhindern. Der in dieser Arbeit vorgestellte Prototyp unterstützt die Entwickler bei der Entwicklung eines Modells. Er ermöglicht den Entwicklern, auf einfache Weise ein Metamodell zu verwenden. Vorkenntnisse über das Metamodell sind nicht nötig. Viele Fehlhandlungen der Entwickler kann der Assistent von vornherein vermeiden. Bei diesem Prototyp wird der Assistent den Entwicklern in Form von Wizards zur Verfügung gestellt. Die Wizards geben einen groben Ablauf vor und verschaffen dem Entwickler einen Überblick über die Modellierungsmöglichkeiten.

Ein Usability-Test des Prototyps hat bestätigt, dass ein Assistent die Modellierung nachhaltig unterstützten kann. Die Modellierung ist für die Entwickler leichter, und sie machen weniger Fehlhandlungen. Für ein Softwareprojekt hat das große Vorteile. Die Produktivität eines Entwicklers ist höher. Weniger Fehlerzustände bedeuten weniger Kosten (Personal, Zeit und Geld) für die Behebung von Fehlerzuständen. Dadurch ist das Risiko für ein Scheitern des Projekts geringer.

Der Prototyp kann damit den Entwicklern die modellgetriebene Softwareentwicklung erleichtern. Dennoch sind beim Usability-Test des Prototyps einige Probleme zum Vorschein gekommen, die Verbesserungen des Assistenten vor einem produktiven Einsatz erfordern. Der Usability-Test hat vor allem die Grenzen von Wizards aufgezeigt. Wizards sind zur Unterstützung von sehr flexiblen Abläufen weniger geeignet, da sie einen zu starren Ablauf vorgeben.

Es muss deshalb versucht werden, die Flexibilität der Wizards zu erhöhen, um den Arbeitsabläufen aller Benutzer gerecht zu werden. Außerdem muss nach Alternativen zu Wizards gesucht werden, die es ermöglichen die Schritte bei der Modellierung in allen sinnvollen Reihenfolgen durchzuführen, den Benutzer bzw. Entwickler aber dennoch führen können.

Erstrebenswert für die Zukunft ist eine Unterstützung für die Entwickler in allen Bereichen der modellgetriebenen Softwareentwicklung. Erfreulich wäre eine Unterstützung auf einem so hohen Niveau, wie es zurzeit bei der Programmierung in einer integrierten Entwicklungsumgebung (IDE) möglich ist.

Die Modellierung ist nicht der einzige Anwendungsfall der modellgetriebenen Softwareentwicklung, der von einem Assistenten unterstützt werden kann. Es bleibt zu untersuchen, wie die weiteren Anwendungsfälle unterstützt werden können. Die Assistenten müssen dabei nicht unbedingt einen Wizard verwenden. Ein Wizard ist nicht immer das richtige Mittel für einen Assistenten. Einige Möglichkeiten wurden in dieser Arbeit genannt, müssen aber noch im Detail auf Machbarkeit untersucht werden.

Um den gesamten Entwicklungsprozess durch ein Assistenzsystem unterstützen zu können, könnte man den Entwicklungsprozess selbst mit Modellen beschreiben. Aus den Modellen und dem aktuellen Stand eines Projekts könnte ein Assistenz-System ableiten, wie die am Projekt beteiligten Personen angemessen unterstützt werden können. Zwischen Entwicklungsprozess und Produkt wäre dadurch ein fließender Übergang. Es sind allerdings standardisierte Metamodelle notwendig, um das realisieren zu können.

Anhang

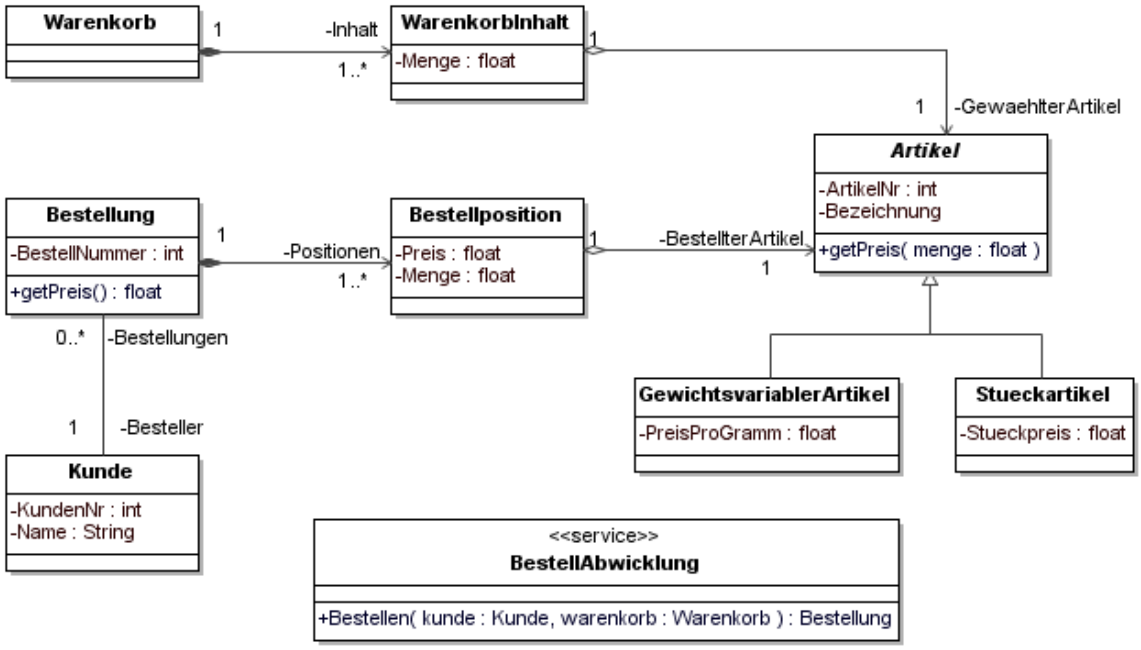
A. Aufgabenstellung der Usability-Tests von openArchitectureWare

A.1. Aufgabenstellung für den ersten Usability-Test

Das Szenario

Ihr Entwicklungsteam entwickelt gerade für einen Händler einen Onlineshop. Der Händler verkauft Artikel, die entweder nach Stückzahl oder Gewicht abgerechnet werden. Die Anforderungen sind bereits aufgenommen und die Geschäftsabläufe analysiert worden. Sie sollen nun das System unter Anwendung der modellgetriebenen Softwareentwicklung realisieren.

Folgendes UML-**Klassendiagramm** ist bei der Analyse der Geschäftsabläufe entstanden:



Aus dem Pflichtenheft gehen unter Anderem folgende **Anforderungen** hervor:

[REQ_4711] Die Geschäftsobjekte müssen dauerhaft in nichtflüchtigem Speicher gespeichert werden. Diese Geschäftsobjekte sind:

1. alle Artikel
2. alle Kunden
3. alle ausgeführten Bestellungen
4. alle Bestellposition einer Bestellung
5. die Warenkörbe aller aktiven Sitzungen und ihr Inhalt

Infrastruktur

Folgende **Werkzeuge** stehen zur Verfügung:

- Entwicklungsumgebung: Eclipse 3.3
- MDA-Werkzeug: openArchitectureWare (oAW) 4.2 (als Plugin in Eclipse integriert)
- UML-Werkzeug: MagicDraw 15 Community Edition

Als Middleware für die Persistenz[8] wird Hibernate verwendet. Die Generierung der Persistenzaspekte wird von einem Cartridge[9] für oAW übernommen.

Zusätzliche Hilfsmittel

Die Dokumentation des Hibernate Cartridges für oAW ist als PDF und Ausdruck vorhanden.

[8] Persistenz = Objekte dauerhaft in nichtflüchtigem Speicher speichern
[9] Cartridge (engl. Steckmodul) = Generatorkomponente für eine spezifische Domäne

Aufgaben

Öffnen Sie das bereits angelegte Eclipse-Projekt. Führen Sie nun folgende Aufgaben aus:

Teil 1

Erweitern Sie das Projekt, bis die Anforderung REQ_4711 erfüllt ist. Diese Anforderung ist erfüllt, wenn:

1. die Generalisierungs-Beziehungen (Vererbung) im Persistenzmodell korrekt abgebildet werden
2. die Assoziations-Beziehungen im Persistenzmodell korrekt abgebildet werden
3. aus dem Persistenzmodell hervorgeht, dass:
    a) die in REQ_4711 genannten Geschäftsobjekte persistent gemacht werden können
    b) das Attribut “ArtikelNr” der Primärschlüssel von “Artikel” ist
    c) das Attribut “KundenNr” der Primärschlüssel von “Kunde” ist
    d) das Attribut “BestellNr” der Primärschlüssel von “Bestellung” ist
    e) die übrigen Geschäftsobjekte einen künstlichen Primärschlüssel haben
    f) das Attribut “Name” von “Kunde” indexiert wird
    g) die Tabelle der Bestellpositionen den Namen “Bestellpos” hat
4. alle Klassen erfolgreich generiert wurden
5. keine Fehlermeldungen (Error aber nicht Warning) mehr vorhanden sind

Teil 2

Sehen sie im Fenster “Recipes” nach, ob hier Fehlermeldungen vorliegen. Versuchen Sie die Fehler zu beheben.

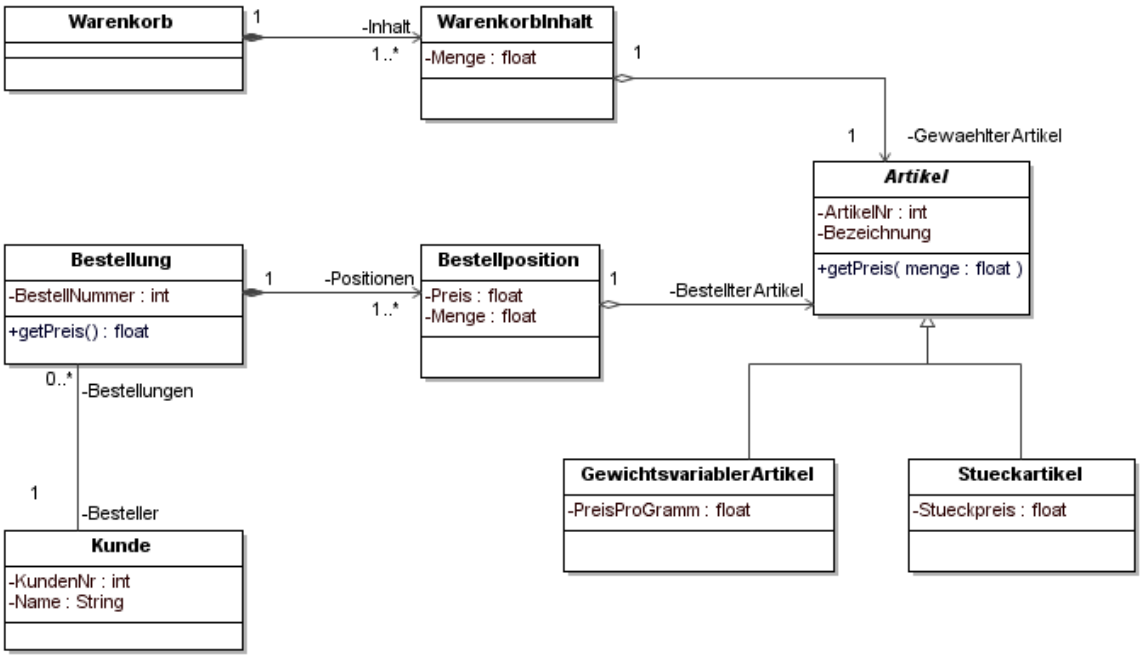
Implementieren Sie die Methode “Bestellen” der Service Komponente “Bestellabwicklung”. Dies wird als erfolgreich betrachtet, wenn die Methode ein neues Bestellungs-Objekt zurückgibt (return new ...).

A.2. Aufgabenstellung für den Usability-Test des Prototyps

Das Szenario

Ihr Entwicklungsteam entwickelt gerade für einen Händler einen Onlineshop. Der Händler verkauft Artikel, die entweder nach Stückzahl oder Gewicht abgerechnet werden. Die Anforderungen sind bereits aufgenommen und die Geschäftsabläufe analysiert worden. Sie sollen nun das System unter Anwendung der modellgetriebenen Softwareentwicklung realisieren.

Folgendes UML-**Klassendiagramm** ist bei der Analyse der Geschäftsabläufe entstanden:



Aus dem Pflichtenheft gehen unter Anderem folgende **Anforderungen** hervor:

[REQ_4711] Die Geschäftsobjekte müssen dauerhaft in nichtflüchtigem Speicher gespeichert werden. Diese Geschäftsobjekte sind:

1. alle Artikel
2. alle Kunden
3. alle ausgeführten Bestellungen
4. alle Bestellpositionen einer Bestellung
5. die Warenkörbe aller aktiven Sitzungen und ihr Inhalt

Infrastruktur

Folgende **Werkzeuge** stehen zur Verfügung:

- Entwicklungsumgebung: Eclipse 3.3
- Assistent zum Erstellen der Entitäten eines Persistenzmodells

Als Middleware für die Persistenz[10] wird Hibernate verwendet. Die Generierung der Persistenzaspekte wird von einem Cartridge[11] für oAW übernommen.

[10] Persistenz = Objekte dauerhaft in nichtflüchtigem Speicher speichern
[11] Cartridge (engl. Steckmodul) = Generatorkomponente für eine spezifische Domäne

Aufgaben

Öffnen Sie das bereits angelegte Eclipse-Projekt. Führen Sie nun folgende Aufgaben aus:

Erweitern Sie das Model, bis die Anforderung REQ_4711 erfüllt ist. Diese Anforderung ist erfüllt, wenn:

1. die Generalisierungs-Beziehungen (Vererbung) im Persistenzmodell korrekt abgebildet werden
2. die Assoziations-Beziehungen im Persistenzmodell korrekt abgebildet werden
3. aus dem Persistenzmodell hervorgeht, dass:
   a) die in REQ_4711 genannten Geschäftsobjekte persistent gemacht werden können
   b) das Attribut “ArtikelNr” der Primärschlüssel von “Artikel” ist
   c) das Attribut “KundenNr” der Primärschlüssel von “Kunde” ist
   d) das Attribut “BestellNr” der Primärschlüssel von “Bestellung” ist
   e) die übrigen Geschäftsobjekte einen künstlichen Primärschlüssel haben
   f) das Attribut “Name” von “Kunde” indexiert wird
   g) die Tabelle der Bestellpositionen den Namen “Bestellpos” hat
4. alle Klassen erfolgreich generiert wurden

B. Bei den Usability-Tests ermittelte Metriken

Die folgenden Metriken wurden bei den beiden Usability-Tests ermittelt. In den Metriken ist mit eingerechnet, dass beim Test des Prototyps der zweite Aufgabenteil wegfiel.

Tabelle B.1.: Metrik Bearbeitungszeit (in Minuten)

	Testlauf 1	**Testlauf 2**	**Testlauf 3**	**Arithmetisches Mittel**
1. Test (mit Magic Draw)	28	34	18,5	26,8
2. Test (mit Prototyp)	14,5	17,5	9	13,7
Veränderung				-49%

Tabelle B.2.: Metrik Anzahl der Mausklicks (linke Maustaste)

	Testlauf 1	**Testlauf 2**	**Testlauf 3**	**Arithmetisches Mittel**
1. Test (mit Magic Draw)	167	211	107	162
2. Test (mit Prototyp)	143	115	71	110
Veränderung				-32%

Tabelle B.3.: Metrik Anzahl der Mausklicks (rechte Maustaste)

	Testlauf 1	**Testlauf 2**	**Testlauf 3**	**Arithmetisches Mittel**
1. Test (mit Magic Draw)	29	29	21	26
2. Test (mit Prototyp)	0	0	0	0
Veränderung				-100%

Tabelle B.4.: Metrik Summe der Mausklicks (rechte und linke Maustaste)

	Testlauf 1	**Testlauf 2**	**Testlauf 3**	**Arithmetisches Mittel**
1. Test (mit Magic Draw)	196	240	128	188
2. Test (mit Prototyp)	143	115	71	110
Veränderung				-42%

Tabelle B.5.: Metrik Mausstrecke (in Meter)

	Testlauf 1	**Testlauf 2**	**Testlauf 3**	**Arithmetisches Mittel**
1. Test (mit Magic Draw)	28,1m	28,7m	14,4m	23,8m
2. Test (mit Prototyp)	23,0	22,3m	8,8m	18,0m
Veränderung				-24%

Tabelle B.6.: Metrik Tastaturanschläge

	Testlauf 1	**Testlauf 2**	**Testlauf 3**	**Arithmetisches Mittel**
1. Test (mit Magic Draw)	52	65	41	53
2. Test (mit Prototyp)	20	26	17	21
Veränderung				-60%

B.1. Diagramme mit dem Verlauf der erfassten Daten

Die folgenden Diagramme zeigen den Verlauf der Tastaturanschläge, Mausstrecke und Mausklicks mit der linken sowie rechten Maustaste. In den Diagrammen ist nicht mit eingerechnet, dass beim Test des Prototyps der zweite Aufgabenteil wegfiel.

Abbildung B.1.: Test 1 - Testlauf 1

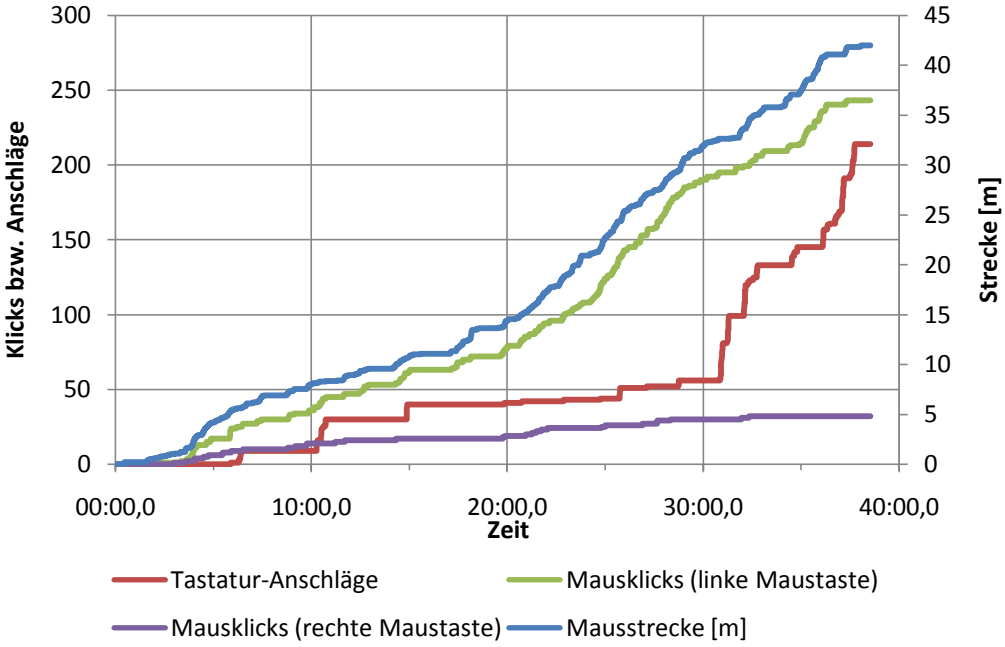


Abbildung B.2.: Test 1 - Testlauf 2

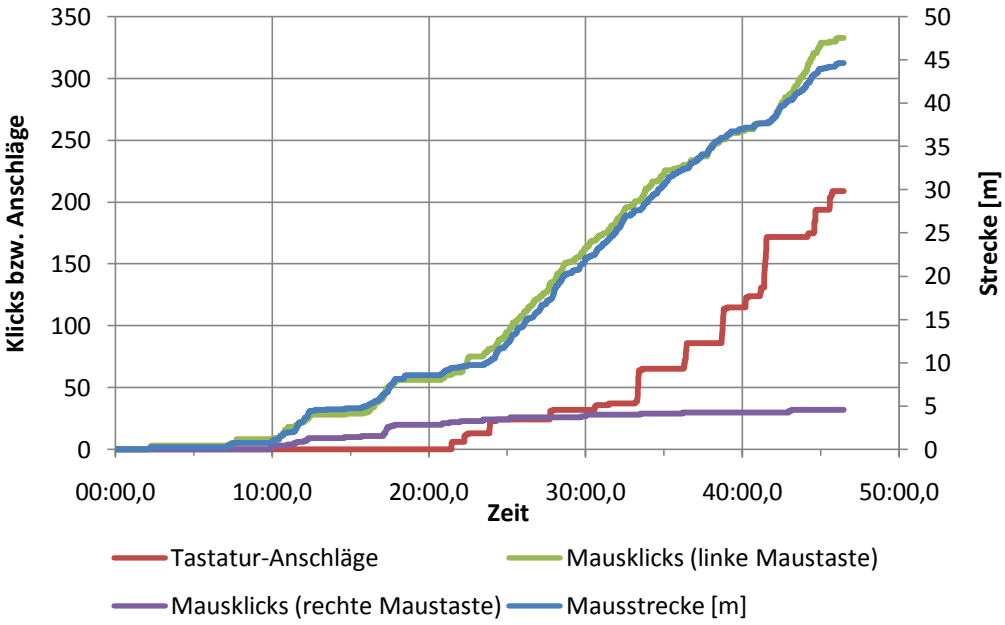


Abbildung B.3.: Test 1 - Testlauf 3

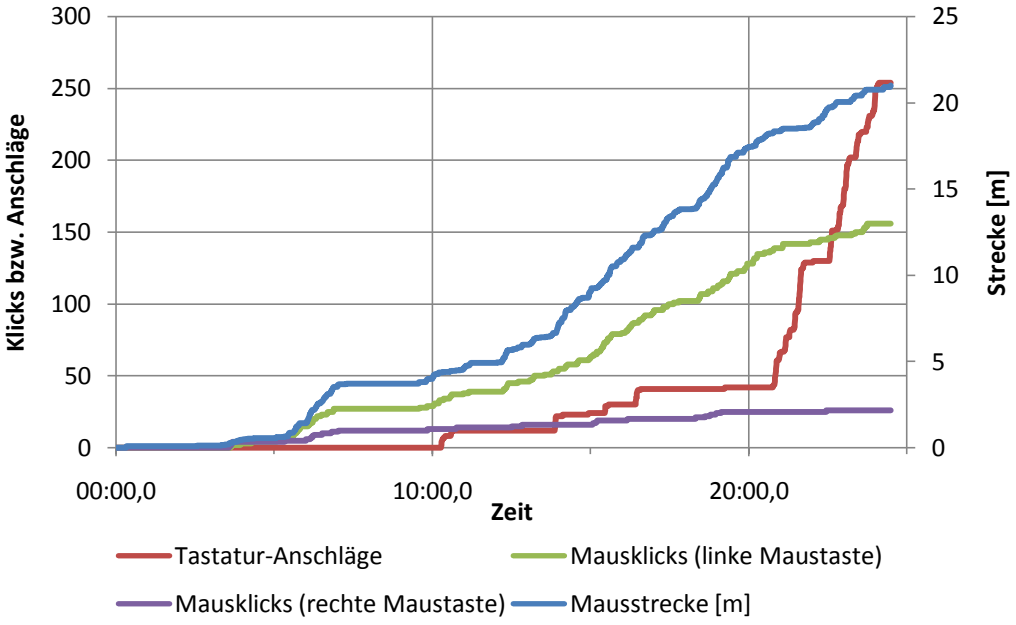


Abbildung B.4.: Test 2 - Testlauf 1

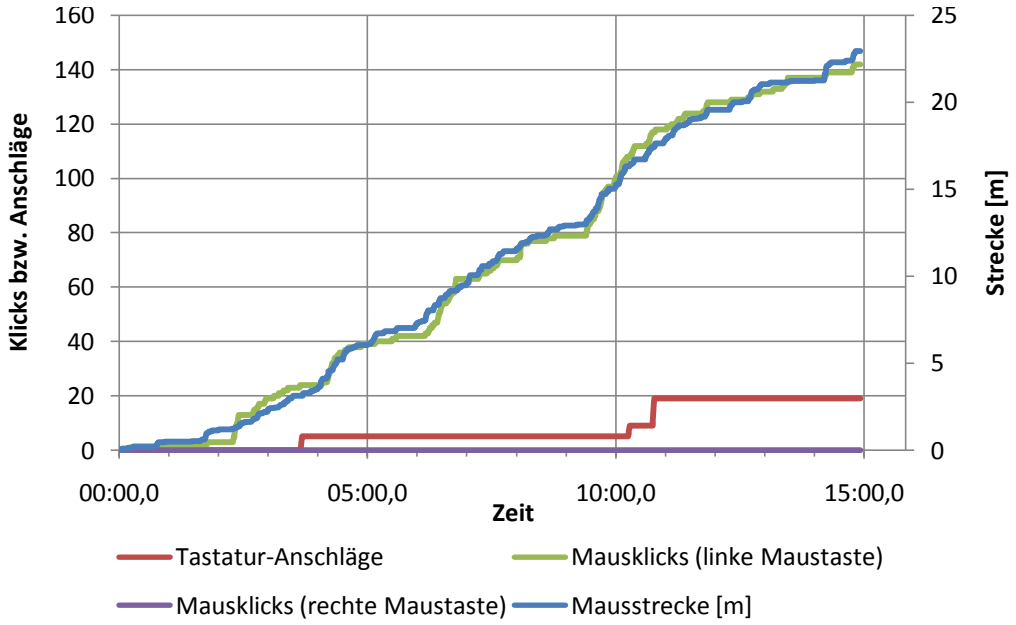


Abbildung B.5.: Test 2 - Testlauf 2

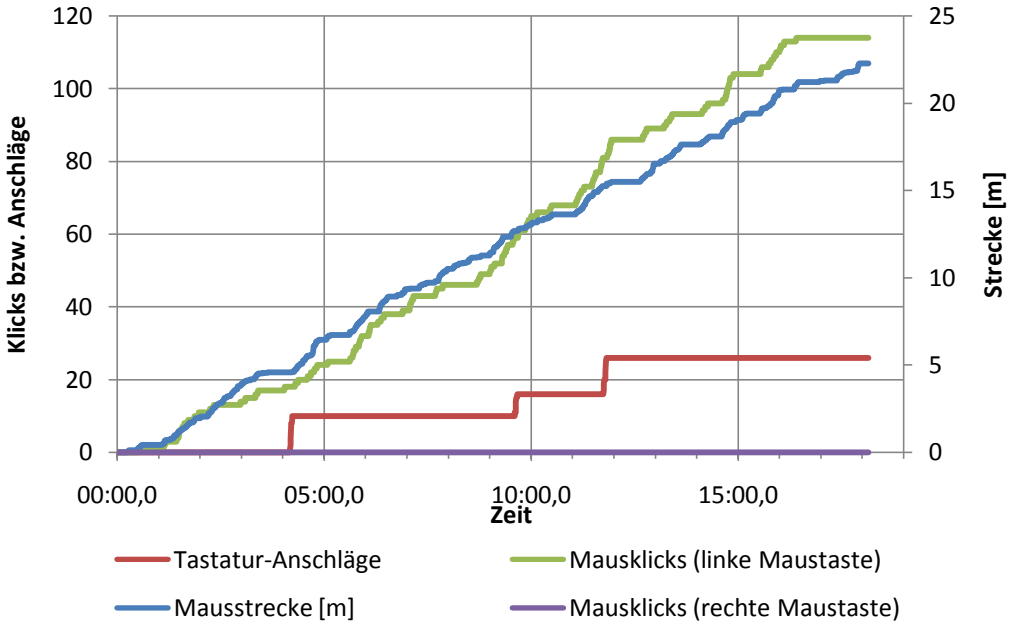
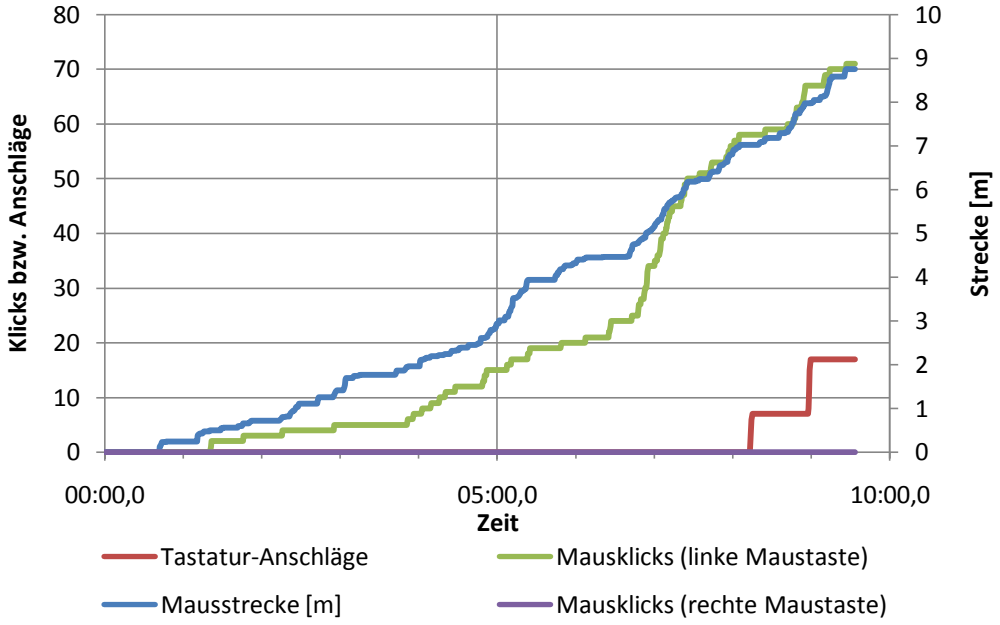


Abbildung B.6.: Test 2 - Testlauf 3



C. Dokumentation des Prototyps

C.1. Metamodelle

C.1.1. Das Aufgaben Metamodell

Die Aufgabe, die ein Wizard repräsentiert, wird mit dem im folgenden erläuterten Metamodell beschrieben. Die Abbildungen C.1, C.2 und C.3 geben einen Überblick über das Metamodell.

Task Ein Task ist eine Aufgabe, die bei der Modellierung durchgeführt wird. Um eine Aufgabe erfolgreich abzuschließen, müssen ein oder mehrere Schritte durchgeführt werden. Welche Schritte zu einer Aufgabe gehören und die Reihenfolge, in der sie ausgeführt werden müssen, wird mit dem Attribut `Steps` festgelegt. `Steps` ist eine Liste von `AbstractStep`s.

AbstractStep Die abstrakte Oberklasse aller Schritte. Ein Schritt wird durchgeführt, um eine Aufgabe (`Task`) zu erledigen. Die unterschiedlichen Klassen von Schritten werden später erläutert.

Output Abstrakte Klasse für die Schritte, in denen ein oder mehrere Modell-Elemente ausgewählt werden. Diese Auswahl ist die Ausgabe des Schritts und kann als Eingabe in einem anderen Schritt verwendet werden (s. u. bei `InputDependency`).

SingleElementOutput Ein Schritt, in dem genau ein Modell-Element ausgewählt wird.

CollectionOutput Ein Schritt, in dem mehrere Modell-Elemente ausgewählt werden.

InputDependency Ein Schritt, der ein oder mehrere Modell-Elemente als Eingabe benötigt. Die Eingabe ist die in einem zuvor durchgeführten Schritt getätigte Auswahl (siehe `Output`).

SingleElementInputDependency Ein Schritt, der genau ein Modell-Element als Eingabe benötigt. Die Eingabe ist vom Typ `SingleElementOutput` und wird mit dem Attribut `Input` gesetzt.

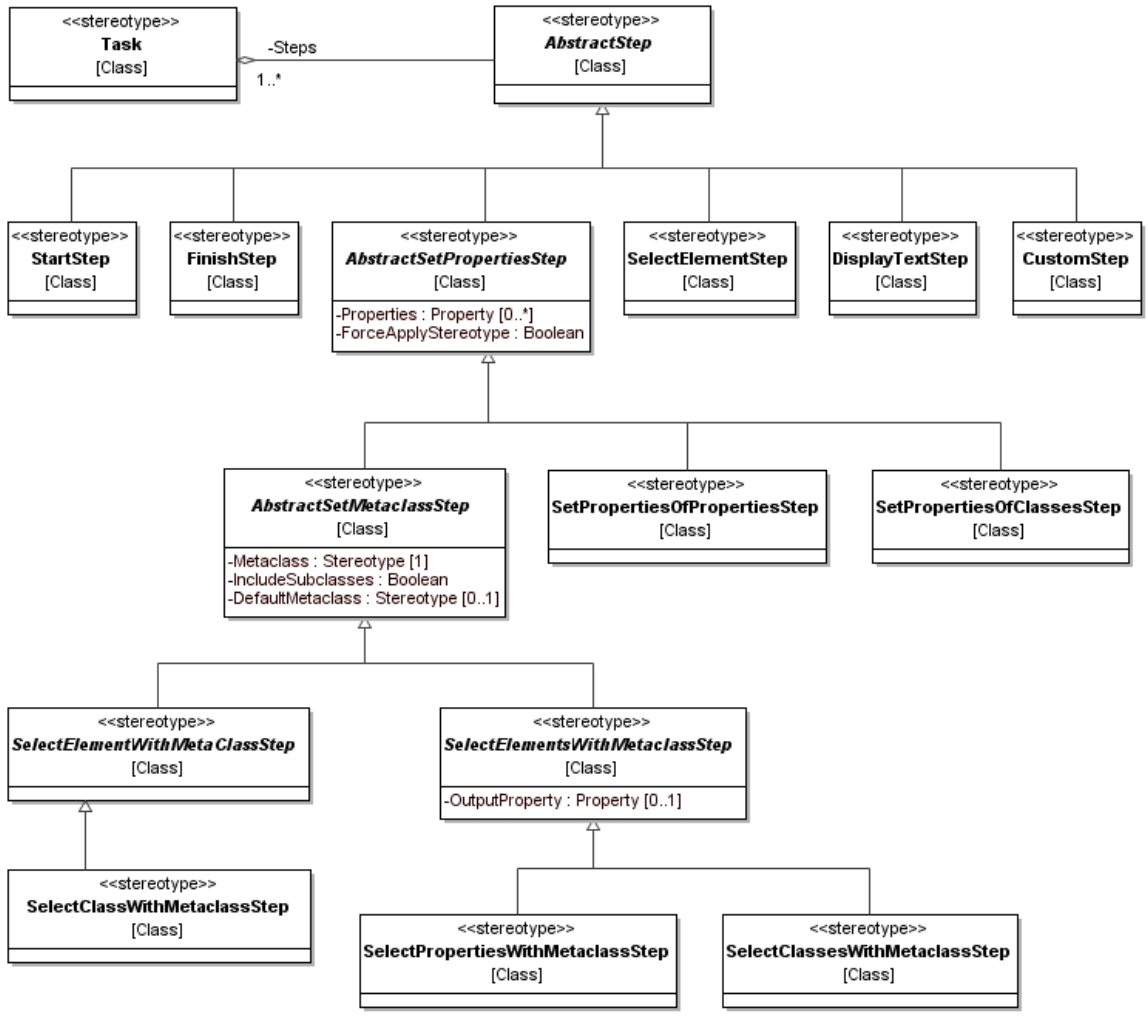


Abbildung C.1.: Das Aufgaben Metamodell

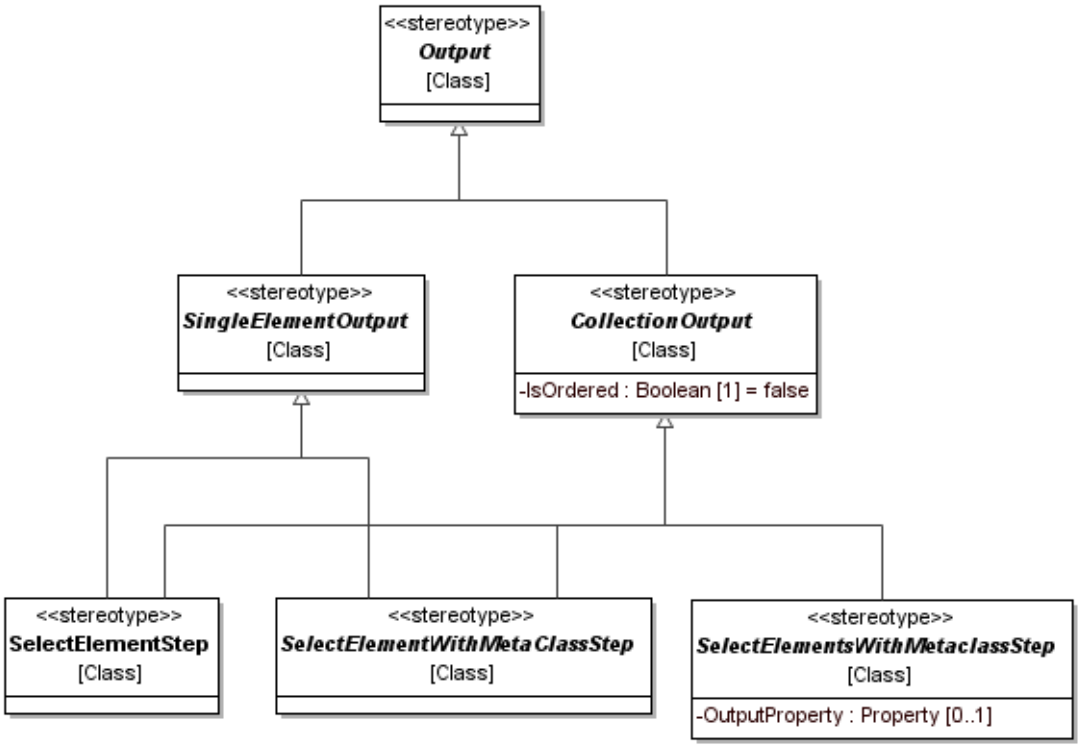


Abbildung C.2.: Das Aufgaben Metamodell - Output

CollectionInputDependency Ein Schritt, der eine Liste von Modell-Elementen als Eingabe benötigt. Die Eingabe ist vom Typ `CollectionOutput` und wird mit dem Attribut `Input` gesetzt. Wenn die Eingabe nicht gesetzt ist, wird das gesamte Modell als Eingabe verwendet. Dies muss mindestens in einem Schritt einer Aufgabe der Fall sein.

Schritte

Die im Folgenden erläuterten Schritt-Metaklassen spezialisieren `AbstractStep`. Bei der Modellierung mit UML und der Verwendung von Metamodellen (in Form von Profilen), ist eine der wesentlichen Tätigkeiten das Versehen von Modell-Elementen mit Stereotypen und das setzen von Tagged-Values. Diese Tätigkeiten werden in Schritten, welche die Metaklasse `AbstractSetMetaclassStep` haben, vorgenommen und sind der zentrale Punkt für die Unterstützung des Benutzers durch einen Wizard. Der Wizard soll den Benutzer in diesen Schritten so unterstützen, dass er das Metamodell ohne Probleme verwenden kann und keine Fehler bei der Modellierung macht.

Abbildung C.3.: Das Aufgaben Metamodell - Input

StartStep Der Anfang einer Aufgabe. Dieser Schritt verschafft einem Benutzer bzw. Entwickler einen Überblick über die folgenden Schritte.

FinishStep Ein Schritt, mit dem eine Aufgabe abgeschlossen wird. In diesem Schritt kann eine Modelltransformation durchgeführt werden.

CustomStep Ein benutzerdefinierter Schritt, der komplett selbst implementiert werden muss. Es ist allerdings vorgesehen, dass die anderen Step-Klassen alle Anwendungsfälle abdecken.

DisplayTextStep Ein Schritt, in welchem der Benutzer einen Hinweis oder Erläuterungen erhält.

SelectElementStep Ein Schritt, in dem ein Modell-Element ausgewählt wird. Das gewählte Modell-Element wird im weiteren Verlauf der Aufgabe verwendet. Die Eigenschaften des ausgewählten Elements werden in diesem Schritt nicht geändert.

AbstractSetPropertiesStep Eine abstrakte Oberklasse der Schritte, in denen Properties bzw. Attribute eines Modellelements angepasst werden können.

Welche Properties in dem Schritt gesetzt werden können, wird mit dem Attribut `Properties` angegeben.

Das Attribut ForceApplyStereotype vom Typ Boolean gibt an, ob die Properties auch dann gesetzt werden können, wenn das Stereotyp, welches die Property definiert, noch nicht auf das Modell-Element angewandt wurde. Wenn das Stereotyp noch nicht angewandt wurde, wird er auf das Modellelement angewandt, sobald der Wert einer seiner Properties gesetzt wird.

AbstractSetMetaclassStep Ein Schritt, in dem ein oder mehrere Modell-Elemente mit einer Metaklasse versehen werden. Bei der UML wird dies durch das Setzen von Stereotypen gemacht. `AbstractSetMetaclassStep` ist eine Unterklasse von `AbstractSetPropertiesStep`.

Die Metaklasse wird mit dem Attribut `Metaclass` angegeben. Die Modell-Elemente, welche die in `Metaclass` angegebene Metaklasse noch nicht haben, können in diesem Schritt mit dieser Metaklasse versehen werden.

Das Attribut `IncludeSubclasses` gibt an, ob auch die Unterklassen der in `Metaclass` angegebenen Metaklasse verwendet werden können. Wenn `IncludeSubclasses` den Wert «true» hat, kann mit dem Attribut `DefaultMetaclass` die standardmäßig verwendete Metaklasse angegeben werden.

SelectElementWithMetaclassStep Ein Schritt, in dem ein einzelnes Element ausgewählt wird, dass eine bestimmte Metaklasse hat. `SelectElementWithMetaclassStep` ist eine Unterklasse von `AbstractSetMetaclassStep`.

Das ausgewählte Modell-Element kann als Eingabe in einem der folgenden Schritte dienen.

SelectClassWithMetaclassStep Ein Schritt, in dem genau eine Klasse ausgewählt wird. `SelectClassWithMetaclassStep` ist eine Unterklasse von `SelectElementWithMetaclassStep`.

SelectElementsWithMetaclassStep Ein Schritt, in dem mehrere Elemente ausgewählt werden, die eine bestimmte Metaklasse haben. `SelectElementsWithMetaclassStep` ist eine Unterklasse von `AbstractSetMetaclassStep`.

Die ausgewählten Modell-Elemente können als Eingabe in einem der folgenden Schritte dienen.

SelectClassesWithMetaclassStep Ein Schritt, in dem mehrere Klassen ausgewählt werden. `SelectClassesWithMetaclassStep` ist eine Unterklasse von `SelectElementsWithMetaclassStep`.

SelectPropertiesWithMetaclassStep Ein Schritt, in dem mehrere Attribute einer Klasse ausgewählt werden. `SelectPropertiesWithMetaclassStep` ist eine Unterklasse von `SelectElementsWithMetaclassStep`.

SetPropertiesOfClassesStep Ein Schritt, in dem ein oder mehrere Attribute einer Klasse gesetzt werden. Bei der UML wird dies durch das Setzen von Tagged-Values gemacht. `SetPropertiesOfClassesStep` ist eine Unterklasse von `AbstractSetPropertiesStep`.

SetPropertiesOfPropertiesStep Ein Schritt, in dem ein oder mehrere Attribute (bzw. Properties) eines Attributs (bzw. einer Property) gesetzt werden. Bei der UML wird dies durch das Setzen von Tagged-Values gemacht. `SetPropertiesOfPropertiesStep` ist eine Unterklasse von `AbstractSetPropertiesStep`.

C.1.2. Metamodell zum Dokumentieren der Aufgaben und Schritte

Die Dokumentation von Aufgaben und Schritten ist eine Sub-Domäne der Modellierung von Aufgaben und Schritten (siehe Kapitel 4.3.3). Sie wird mit einem eigenen Metamodell beschrieben: mit dem Aufgaben-Metamodell. Mit dem Aufgaben-Metamodell werden Aufgaben und Schritte eines Aufgaben-Modells dokumentiert.

Die Aufgaben und Schritte werden dokumentiert, indem sie mit dem Stereotyp `TaskDocumentation` bzw. `StepDocumentation` versehen werden und die Attribute mit Tagged-Values gesetzt werden.

TaskDocumentation Mit diesem Stereotyp werden Aufgaben (siehe `Task`) versehen, um sie zu dokumentieren.

Mit dem Attribut `Description` wird der Zweck der Aufgabe beschrieben. Die Beschreibung soll dem Benutzer deutlich machen, wann und wozu er sie ausführen kann.

Mit dem Attribut `Title` wird der Aufgabe ein benutzerfreundlicher und aussagekräftiger Name gegeben.

StepDocumentation Mit diesem Stereotyp werden Schritte (siehe `AbstractStep`) versehen, um sie zu dokumentieren.

Das Attribut `Description` enthält eine Beschreibung des Schritts. Diese soll dem Benutzer deutlich machen, was in dem Schritt geschieht und wozu er durchgeführt wird.

Mit dem Attribut `Instruction` wird eine Anweisung an den Benutzer festgelegt. Diese Anweisung soll dem Benutzer zeigen, was er in dem Schritt machen soll.

Mit dem Attribut `Title` wird dem Schritt ein benutzerfreundlicher und aussagekräftiger Name gegeben.

C.1.3. Metamodell zum Dokumentieren von Metamodell-Elementen

Mit dem Dokumentations-Metamodell werden die Elemente eines Metamodells dokumentiert (siehe Kapitel 4.3.3).

Ein Modell-Element wird dokumentiert, indem das Element mit einem Stereotyp aus dem Dokumentations-Metamodell versehen wird und die Attribute mit Tagged-Values gesetzt werden.

ModelElementDocumentation Mit diesem Stereotyp werden die Elemente (Klassen, Attribute etc.) eines Metamodells versehen, damit ein Wizard die Elemente des Metamodells dem Benutzer verständlich präsentiert kann.

Dieses Stereotyp und seine Attribute sind die Dokumentation eines Modell-Elements. Die Dokumentation wird in einem Wizard verwendet, um dem Benutzer des Wizards das Modell-Element zu erklären und ihm seine Entscheidungen zu erleichtern.

Mit dem Attribut `Description` wird ein Modell-Element beschrieben und erklärt.

Das Attribut `DisplayName` ist optional und wird dann gesetzt, wenn der Name eines Modell-Elements nicht benutzerfreundlich oder aussagekräftig genug ist. Der Name eines Elements ist nicht benutzerfreundlich, wenn seine Schreibweise nicht der Rechtschreibung entspricht, abgekürzt ist oder Sonderzeichen enthält (z. B. «idx_Name» statt «Index Name»).

C.2. Wizard für Persistenzmodelle

Die folgenden Abbildungen zeigen die Erstellung einer Entität («Kunde») mit einem Wizard für Persistenzmodelle. In Abbildung C.8 ist eine Fehlermeldung zu sehen, die darauf aufmerksam macht, dass ein benötigtes Attribut (in diesem Fall der Name eines Indexes) nicht gesetzt wurde.

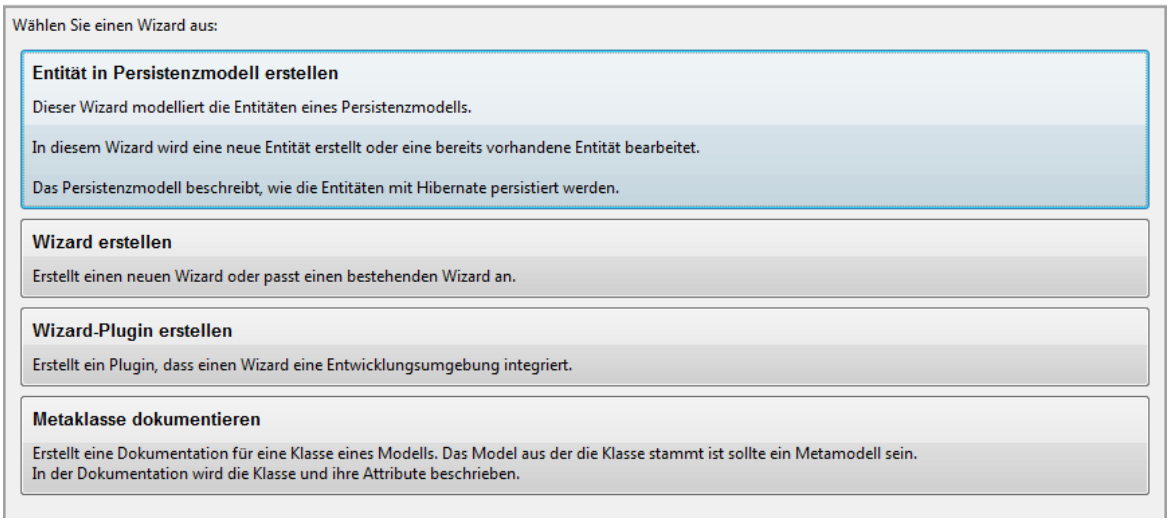


Abbildung C.4.: Modellierung eines Persistenzmodells mit einem Wizard - Auswahl des Wizards

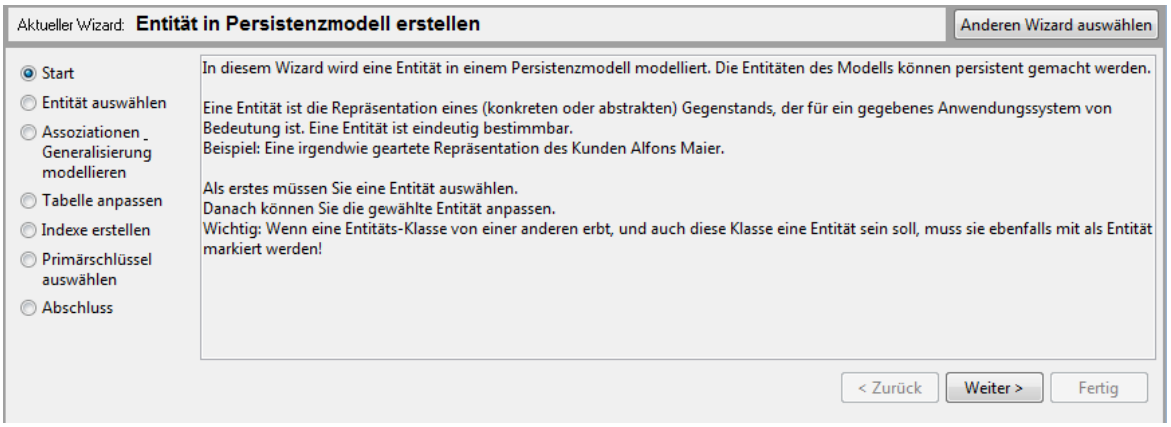


Abbildung C.5.: Modellierung eines Persistenzmodells mit einem Wizard - Erster Schritt

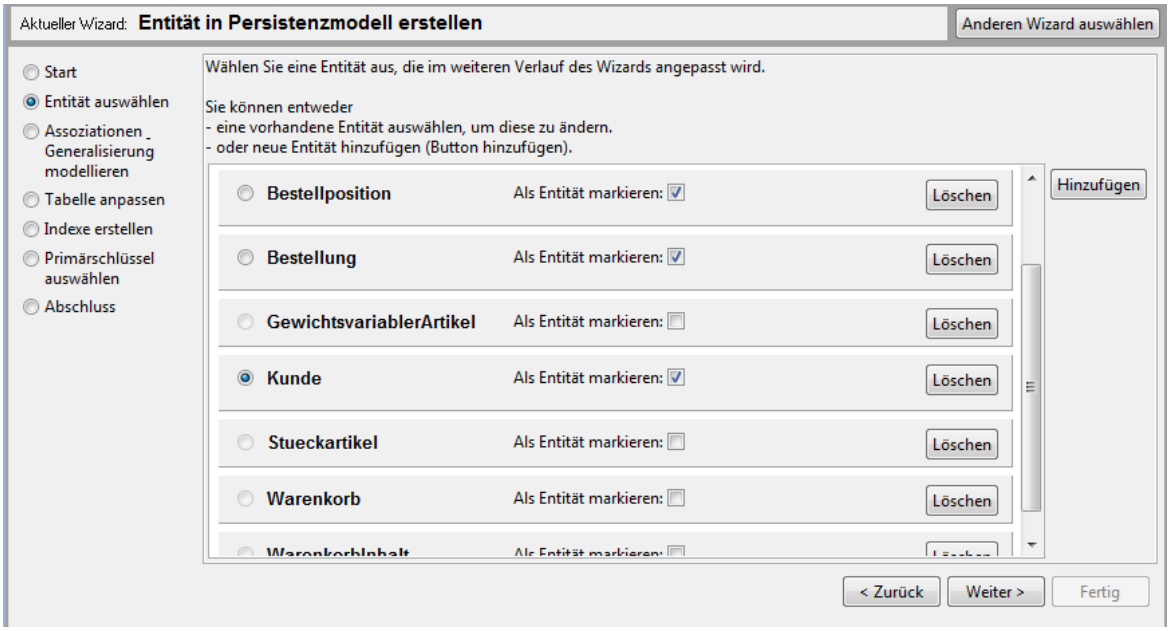


Abbildung C.6.: Modellierung eines Persistenzmodells mit einem Wizard - Entität auswählen

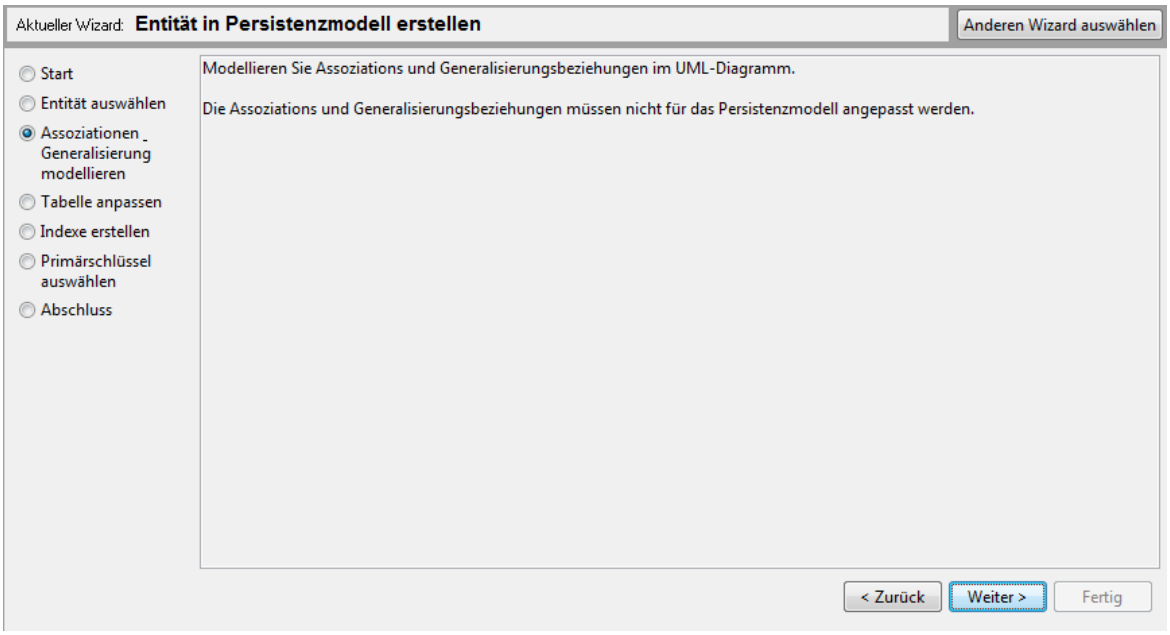


Abbildung C.7.: Modellierung eines Persistenzmodells mit einem Wizard - Hinweis zur Modellierung von Assoziationen

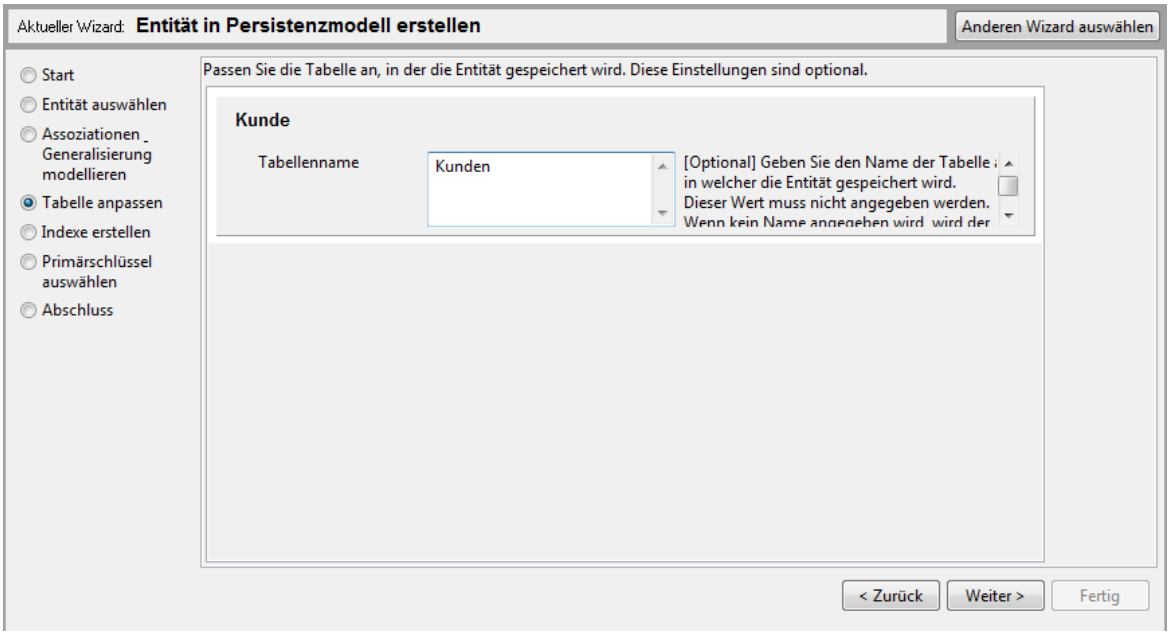


Abbildung C.8.: Modellierung eines Persistenzmodells mit einem Wizard - Tabelle anpassen

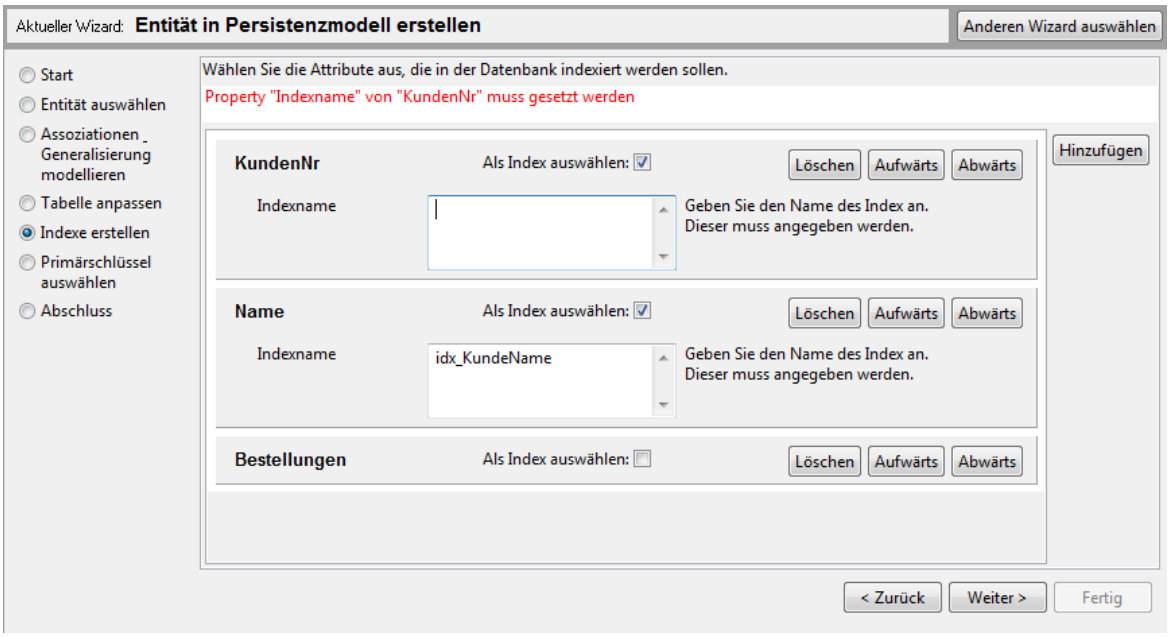


Abbildung C.9.: Modellierung eines Persistenzmodells mit einem Wizard - Indexe

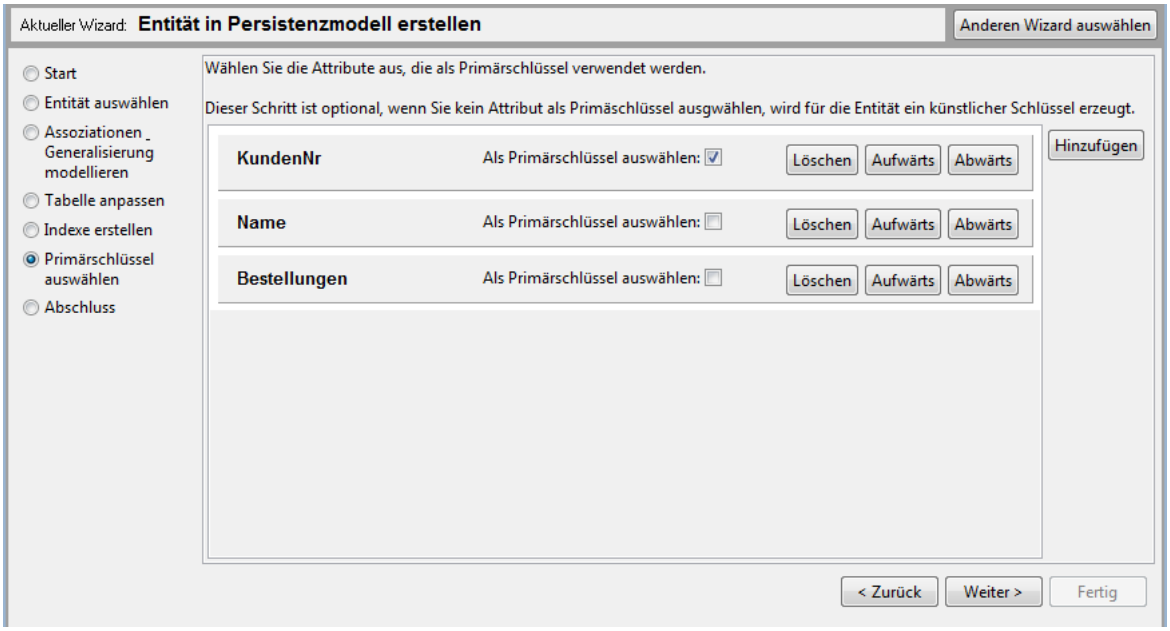


Abbildung C.10.: Modellierung eines Persistenzmodells mit einem Wizard - Primärschlüssel

Abbildung C.11.: Modellierung eines Persistenzmodells mit einem Wizard - Abschluss

C.3. Metawizard

Die folgenden Abbildungen zeigen die Erstellung des Wizards für Persistenzmodelle mit dem Metawizard.

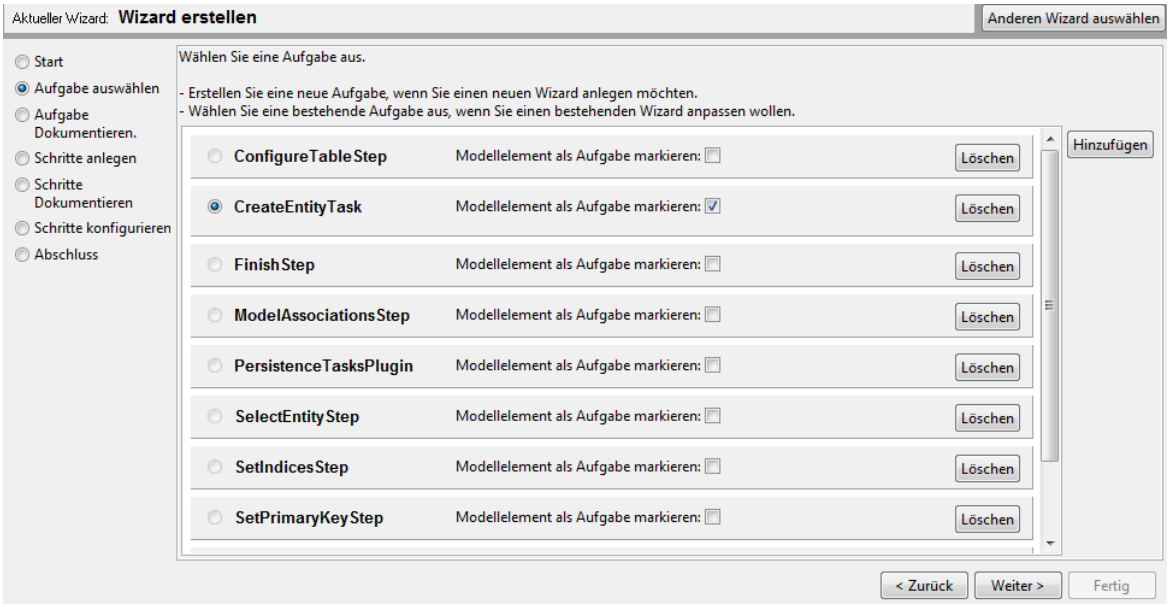


Abbildung C.12.: Metawizard - Aufgabe auswählen

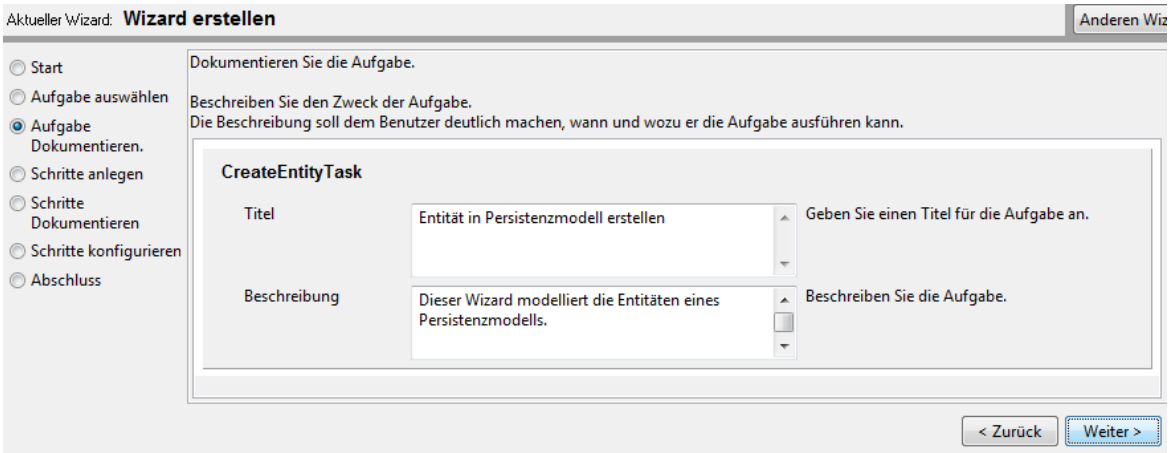


Abbildung C.13.: Metawizard - Aufgabe dokumentieren

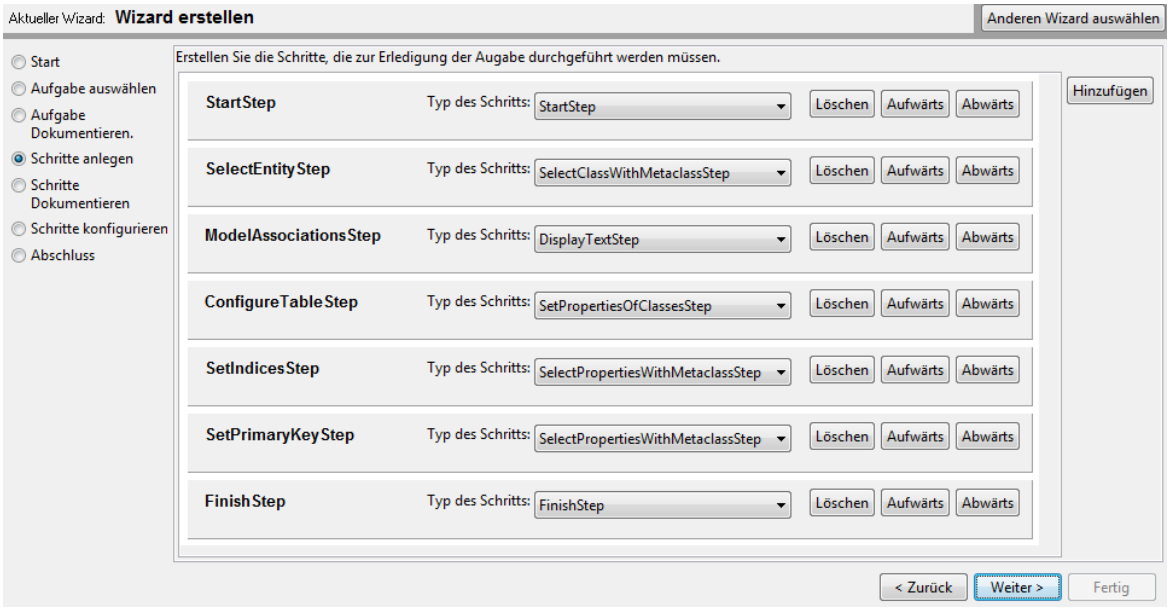


Abbildung C.14.: Metawizard - Schritte erstellen

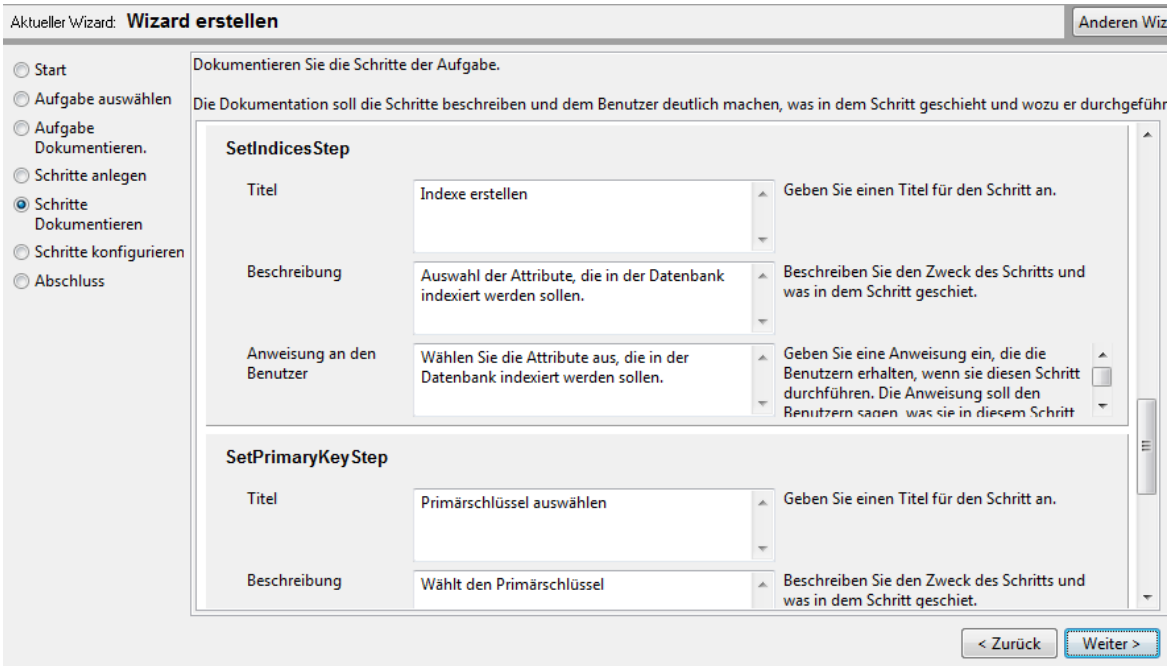


Abbildung C.15.: Metawizard - Schritte dokumentieren

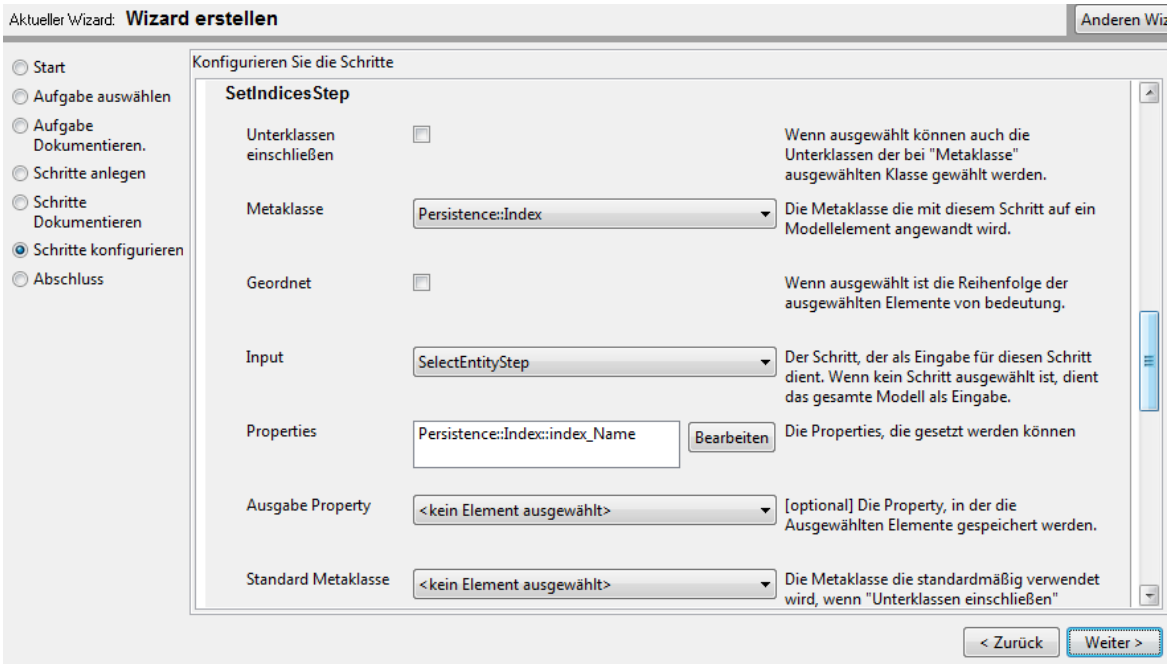


Abbildung C.16.: Metawizard - Schritte konfigurieren

D. Inhalt der beiliegenden DVD

Die beiliegende DVD beinhaltet folgende Dateien:

\ Wurzelverzeichnis der DVD

- **Bachelorarbeit.pdf**
  Dieses Dokument
- **inhalt.txt**
  Das Inhaltsverzeichnis der DVD

Eclipse3.3_mit_oAW4.2\ Eclipse 3.3 mit allen Plugins, die vom Prototyp benötigt werden

Quellen\ Die Quellen dieser Bachelorarbeit, sofern sie als PDF verfügbar sind.

- **index.html**
  Übersicht über die Quellen. Verlinkt die PDF-Dokumente.

Screenshots\ Bildschirmfotos vom Prototyp

Source\ Die Quelltexte des Prototyps.

- **source.zip**
  ZIP-Archiv mit den Quelltexten. Die Projekte können in Eclipse als Projekt importiert werden (Import -> Existing Projects into Workspace).
- **example-project.zip**
  Beispiel-Projekt
- **how-to-install.txt**
  Installationsanleitung
- **how-to-run.txt**
  Anleitung zum ausführen des Prototyps

UsabilityTests\ Filme und Metriken der Usability-Tests

UsabilityTests**Filme_Test1**\ Dieser Ordner enthält die beim ersten Usability-Test aufgezeichneten Filme. Getestet wurde die Modellierung eines Persistenzmodells mit Eclipse, openArchitectureWare und Magic Draw.

UsabilityTests**Filme_Test2**\ Dieser Ordner enthält die beim zweiten Usability-Test aufgezeichneten Filme. Hier wurde die Modellierung eines Persistenzmodells mit dem in Eclipse integrierten Prototyp getestet.

UsabilityTests**Metriken**\ Dieser Ordner enthält bei Usability-Tests aufgezeichneten Maus- und Tastaturdaten sowie die daraus ermittelten Metriken.

- **Hibernate_Manual.pdf**
  Die Dokumentation des Hibernate-Cartridges, die den Testpersonen beim ersten Usability-Test zur verfügung gestellt wurde.

Abbildungsverzeichnis

2.1. Zusammenhang der Begriffe bei der Modellierung (aus [Stahl u. a. (2007)]) . 6
2.2. Dreistufige Hierarchie von Modellen (nach [Stahl u. a. (2007)]) 9
2.3. Der schalenartige Aufbau der MDA (Quelle: [URL:MDA]) 10
2.4. Datenfluss bei Assistenzsystemen (nach Rech u. a. (2006)) 18

3.1. Android Projekt-Wizard: Konfiguration der verwendeten Komponenten 24
3.2. objectiF Entwicklungsumgebung 26
3.3. objectiF “Neues System Anlegen” Wizard: Seite2 27
3.4. openArchitectureWare mit Eclipse 28

4.1. Rollen bei der Modellverwendung 39
4.2. Die Komponenten der Musterarchitektur 48
4.3. Fachliche Architektur des Prototyps 50
4.4. Zusammenspiel von Wizards und Aufgaben (Task) 51
4.5. Assistenz Komponente 52
4.6. Ermittlung der im Kontext verfügbaren Assistenz 53
4.7. Technische Architektur des Prototyps eines Assistenten, der die Entwicklung von Persistenzmodellen unterstützt. 54
4.8. Ablauf der Erstellung eines Assistenten 56
4.9. Architektur des Generators 56

B.1. Test 1 - Testlauf 1 81
B.2. Test 1 - Testlauf 2 81
B.3. Test 1 - Testlauf 3 82
B.4. Test 2 - Testlauf 1 82
B.5. Test 2 - Testlauf 2 83
B.6. Test 2 - Testlauf 3 83

C.1. Das Aufgaben Metamodell 85
C.2. Das Aufgaben Metamodell - Output 86
C.3. Das Aufgaben Metamodell - Input 87
C.4. Modellierung eines Persistenzmodells mit einem Wizard - Auswahl des Wizards 91
C.5. Modellierung eines Persistenzmodells mit einem Wizard - Erster Schritt . . . 91

C.6. Modellierung eines Persistenzmodells mit einem Wizard - Entität auswählen . 92
C.7. Modellierung eines Persistenzmodells mit einem Wizard - Hinweis zur Modellierung von Assoziationen 92
C.8. Modellierung eines Persistenzmodells mit einem Wizard - Tabelle anpassen . 93
C.9. Modellierung eines Persistenzmodells mit einem Wizard - Indexe 93
C.10. Modellierung eines Persistenzmodells mit einem Wizard - Primärschlüssel . . 94
C.11. Modellierung eines Persistenzmodells mit einem Wizard - Abschluss 94
C.12. Metawizard - Aufgabe auswählen 95
C.13. Metawizard - Aufgabe dokumentieren 95
C.14. Metawizard - Schritte erstellen 96
C.15. Metawizard - Schritte dokumentieren 96
C.16. Metawizard - Schritte konfigurieren 97

Tabellenverzeichnis

3.1. Werkzeugübersicht 30

5.1. Gegenüberstellung der Ursachen für Fehlhandlung beim ersten Test und der vom Prototyp angestrebten Verbesserungen 64

B.1. Metrik Bearbeitungszeit (in Minuten) 79
B.2. Metrik Anzahl der Mausklicks (linke Maustaste) 79
B.3. Metrik Anzahl der Mausklicks (rechte Maustaste) 79
B.4. Metrik Summe der Mausklicks (rechte und linke Maustaste) 80
B.5. Metrik Mausstrecke (in Meter) 80
B.6. Metrik Tastaturanschläge 80

Glossar

Cartridge

Ein Cartridge (engl. Steckmodul) ist bei der modellgetriebenen Softwareentwicklung eine Generator-Komponente für eine spezifische Domäne.

Fehlerwirkung

Die Wirkung eines Fehlerzustands in einer Software, die zur Laufzeit nach außen in Erscheinung tritt. Eine Fehlerwirkung wird als Abweichung eines Programms vom spezifizierten verhalten wahrgenommen (Sollwert $\neq$ Istwert). [Spillner und Linz (2005)]

Fehlerzustand

Ein Defekt in einer Software (z. B. eine falsch programmierte oder vergessene Anweisung), der die Ursache für eine Fehlerwirkung ist. Ein Fehlerzustand ist die Folge einer Fehlhandlung. [Spillner und Linz (2005)]

Fehlhandlung

Eine Handlung eines Entwicklers, die zu einem Fehlerzustand in einer Software führt [Spillner und Linz (2005)].

Metaklasse

Eine Metaklasse ist ein Element eines Metamodells mit dem Elemente eines Modells klassifiziert werden.

Software-Lebenszyklus

Der Ablauf der Entstehung und die Fortentwicklung eines Software-Systems. Schließt alle Maßnahmen und Tätigkeiten ein, die während dieser Periode erforderlich sind. Ein Teil des Software-Lebenszyklus ist der Entwicklungsprozess. [URL:Informatikbegriffsnetz]

Usability-Test

Mit einem Usability-Test wird die Benutzbarkeit eines Software-Systems geprüft. Während eines Usability-Tests werden potentielle Benutzer bei der Benutzung des Systems beobachtet. Anhand der Beobachtungen wird bewertet, ob das System die Anforderungen an die Benutzbarkeit und Ergonomie erfüllt.

Quellenverzeichnis

Literatur[12]

[Ames 2001] AMES, Andrea L.: Just what they need, just when they need it: an introduction to embedded assistance. In: *SIGDOC '01: Proceedings of the 19th annual international conference on Computer documentation*. New York, NY, USA : ACM, 2001, S. 111–115. – URL http://doi.acm.org/10.1145/501516.501539. – ISBN 1-58113-295-6

[Balzert 1998] BALZERT, Helmut: *Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*. Spektrum Akademischer Verlag, 1998 (Lehrbücher der Informatik). – ISBN 3-8274-0065-1

[Bauer und King 2006] BAUER, Christian ; KING, Gavin: *Java persistence with Hibernate*. Manning, Greenwich, 2006. – ISBN 1-932394-88-5

[Bräutigam und Schneider 2003] BRÄUTIGAM, Lothar ; SCHNEIDER, Wolfgang: Projektleitfaden Software-Ergonomie. (2003). – URL http://www.hessen-it.de/data/download/broschueren/software-ergonomie.pdf. – http://www.ergonomie-leitfaden.de/. ISBN 3-936598-43-6

[Burton u. a. 1999] BURTON, Mary ; WICKHAM, Daina P. ; PHELPS, Lori ; SPAIN, Kelly ; CREWS, Janna ; RICH, Nicki: Secondary navigation in software wizards. In: *CHI '99: CHI '99 extended abstracts on Human factors in computing systems*. New York, NY, USA : ACM, 1999, S. 294–295. – URL http://doi.acm.org/10.1145/632716.632896. – ISBN 1-58113-158-5

[Dijkstra 1972] DIJKSTRA, Edsger W.: The humble programmer. In: *Commun. ACM* 15 (1972), Nr. 10, S. 859–866. – URL http://doi.acm.org/10.1145/355604.361591. – ISSN 0001-0782

[Dryer 1997] DRYER, D. C.: Wizards, guides, and beyond: rational and empirical methods for selecting optimal intelligent user interface agents. In: *IUI '97: Proceedings of the 2nd international conference on Intelligent user interfaces*. New York, NY, USA : ACM, 1997,

[12] Bei Quellen aus dem Internet wurde die URL und der Inhalt am 20.06.2008 überprüft

S. 265–268. – URL http://doi.acm.org/10.1145/238218.238347. – ISBN 0-89791-839-8

[Gamma u. a. 2004] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Entwurfsmuster*. Addison-Weslay Verlag, 2004. – ISBN 978-3-8273-2199-2

[Gruhn u. a. 2006] GRUHN, Volker ; PIEPER, Daniel ; RÖTTGERS, Carsten: *MDA – Effektives Software-Engineering mit UML und Eclipse*. Springer-Verlag, Berlin Heidelberg, 2006 (Xpert.press). – URL http://www.springerlink.com/content/w05681/. – ISBN 978-3-540-28746-9

[Hailpern und Tarr 2006] HAILPERN, Brent ; TARR, Peri: Model-driven development: the good, the bad, and the ugly. In: *IBM Syst. J.* 45 (2006), Nr. 3, S. 451–461. – URL http://www.research.ibm.com/journal/sj/453/hailpern.pdf. – ISSN 0018-8670

[Neumann 2005] NEUMANN, Carola: *User Centered Design - Benutzerzentrierte Softwareentwicklung*, Hochschule für Angewandte Wissenschaften Hamburg, Studienarbeit, 2005

[OMG 2006] OBJECT MANAGEMENT GROUP: *Meta Object Facility (MOF) Core Specification*. Version 2.0. : , 01 2006

[Raskin 2000] RASKIN, Jef: *The humane interface: new directions for designing interactive systems*. New York, NY, USA : ACM Press/Addison-Wesley Publishing Co., 2000. – ISBN 0-201-37937-6

[Rech u. a. 2006] RECH, Jörg ; RAS, Eric ; DECKER, Björn: Intelligente Assistenz in der Softwareentwicklung 2006. Zusammenfassung der Ergebnisse / Fraunhofer IESE. URL http://publica.fraunhofer.de/starweb/servlet.starweb?path=pub0.web&search=N-50981, 2006 (Reportnr.: 045.06/D). – Forschungsbericht

[Siedersleben 2004] SIEDERSLEBEN, Johannes: *Moderne Software-Architektur – Umsichtig planen, robust bauen mit Quasar*. 1. Auflage, korrigierter Nachdruck 2006. dpunkt.verlag, Heidelberg, 2004. – ISBN 3-89864-292-5

[Spillner und Linz 2005] SPILLNER, Andreas ; LINZ, Tiulo: *Basiswissen Softwaretest*. 3., überarbeitete und aktualisierte Auflage, korrigierter Nachdruck 2007. dpunkt.verlag GmbH, Heidelberg, 2005. – ISBN 3-89864-358-1

[Stahl u. a. 2007] STAHL, Thomas ; VÖLTER, Markus ; EFFTINGE, Sven ; HAASE, Arno: *Modellgetriebene Softwareentwicklung – Techniken, Engineering, Management*. 2. Auflage. dpunkt.verlag, Stuttgart, 2007. – ISBN 978-3-89864-448-8

[Starke 2005] STARKE, Gernot: *Effektive Software Architekturen*. 2. Auflage. Carl Hanser Verlag München Wien, 2005. – ISBN 3-446-22846-2

[Thomas 2004] THOMAS, Dave: MDA: Revenge of the Modelers or UML Utopia? In: *IEEE SOFTWARE* Volume 21 (2004), S. 22–24. – URL http://www.martinfowler.com/ieeeSoftware/mda-thomas.pdf

[Wanner und Siegl 2007] WANNER, Gerhard ; SIEGL, Stefan: Modellgetriebene Softwareentwicklung auf Basis von Open-Source-Werkzeugen – reif für die Praxis? In: *Informatik Spektrum* Band 30, Heft 5 (2007), S. 340–352

Normen

[ISO 9126] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION (ISO): *ISO 9126*. – Zitiert nach [Balzert (1998)]

[ISO 9241] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION (ISO): *ISO 9241*. – Zitiert nach [Bräutigam und Schneider (2003)]

Webseiten[13]

[URL:AndroMDA] ANDROMDA.ORG: *AndroMDA*. – URL http://www.andromda.org

[URL:Eclipse-oAW] ECLIPSE FOUNDATION: *openArchitectureWare (oAW)*. – URL http://www.eclipse.org/gmt/oaw/

[URL:FornaxPlatform] KAMANN, Thorsten: *The Fornax-Platform*. – URL http://www.fornax-platform.org/

[URL:HAW-Uselab] HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG: *Usability Labor der Informatik*. – URL http://users.informatik.haw-hamburg.de/~use-lab/

[URL:Hibernate] HIBERNATE.ORG: *Hibernate*. – URL http://www.hibernate.org

[URL:Informatikbegriffsnetz] GESELLSCHAFT FÜR INFORMATIK E.V. (GI): *Informatik-Begriffsnetz*. – URL http://www.informatikbegriffsnetz.de/

[URL:MDA] OBJECT MANAGEMENT GROUP: *Model Driven Architecture (MDA)*. – URL http://www.omg.org/mda

[13] Die Gültigkeit der URLs wurde am 20.06.2008 überprüft

[URL:oAW] OPENARCHITECTUREWARE.ORG: *openArchitectureWare (oAW)*. – URL http://www.openarchitectureware.org

[URL:objectiF] MICROTOOL GMBH: *objectiF*. – URL http://www.microtool.de/objectiF

[URL:OMG] OBJECT MANAGEMENT GROUP: *The Object Management Group (OMG)*. – URL http://www.omg.org

[URL:UML] OBJECT MANAGEMENT GROUP: *Unified Modeling Language*. – URL http://www.uml.org

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) bzw.§24(4) ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 20. Juni 2008 Hauke Wittern