FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

# Entwurf und Implementierung eines GUI basierten RCP Frameworks zur Spezifikation und Modellierung von Testfällen zur Optimierung der White-Box-Komplexität bei hoher Software-Volatilität

Diplomarbeit

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Software Engineering and Internet Computing**

eingereicht von

**Markus Zoffi**
Matrikelnummer 0725182

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Thomas Grechenig
Mitwirkung: Roland Breiteneder

Wien, 20. Juli 2014

(Unterschrift Verfasser/In) (Unterschrift Betreuung)

# Entwurf und Implementierung eines GUI basierten RCP Frameworks zur Spezifikation und Modellierung von Testfällen zur Optimierung der White-Box-Komplexität bei hoher Software-Volatilität

Diplomarbeit

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Software Engineering and Internet Computing**

eingereicht von

**Markus Zoffi**
Matrikelnummer 0725182

ausgeführt am
Institut für Rechnergestützte Automation Forschungsgruppe Industrial Software der Fakultät für Informatik der Technischen Universität Wien

**Betreuung:** Thomas Grechenig
**Mitwirkung:** Roland Breiteneder

Wien, 20. Juli 2014

# Eidesstattliche Erklärung

Markus Zoffi
Herzgasse 60/7/9, 1100 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

I hereby declare that I am the sole author of this thesis, that I have completely indicated all sources and help used, and that all parts of this work – including tables, maps and figures – if taken from other works or from the internet, whether copied literally or by sense, have been labelled including a citation of the source.

| | |
|---|---|
| (Ort, Datum) | (Unterschrift Verfasser/In) |

# Kurzfassung

Das Testen von Software sowie ihre Qualitätssicherung sind feste Bestandteile des Software-Entwicklungsprozesses und tragen maßgeblich zum Erfolg eines Projekts bei. Aufgrund der Volatilität von Anforderungen, Komponenten, Applikationen und besonders von Benutzeroberflächen kann sich das Testen derselben als aufwändig und zeitintensiv herausstellen. Vor allem im Zusammenhang mit Modifikationen der Benutzeroberfläche kann die Konsistenz von Testfällen oder Testskripts wesentlich beeinträchtigt werden, da diese einen hohen Grad an Abhängigkeit von den Elementen der Benutzeroberfläche besitzen können. Durch eine Modellrepräsentation der Benutzeroberfläche und einen modellbasierten Testansatz könnte diese Problematik vermindert werden.

Im Rahmen dieser Arbeit wird ein Ansatz für modellbasiertes Testen von Benutzeroberflächen entworfen und prototypisch implementiert. Die realisierte Applikation vereinfacht die Spezifikation von Testfällen und unterstützt den Tester bei der funktionalen Testfallerstellung, ohne dabei Programmierkenntnisse vorauszusetzen. Durch Verwendung des entworfenen Prototyps ist es dem Tester möglich, Testfälle anhand der Modellrepäsentation einer Benutzeroberfläche zu spezifizieren und diese anschließend als konkreten Testfall zu exportieren und in einen bestehenden Testprozess zu integrieren.

Zu Beginn der Arbeit wird eine Einführung in die Grundlagen der relevanten Themenbereiche gegeben. Hier werden grundlegende Definitionen zum besseren Verständnis der Arbeit beschrieben sowie gängige Praktiken erläutert. Die nachfolgenden Kapitel der Arbeit umfassen theoretische und praktische Vorgehensweisen. Planungsprozesse wie eine Anforderungsanalyse und ein Architekturkonzept sowie eine heuristische Evaluierung des Designs der geplanten Benutzeroberfläche spiegeln den theoretischen Aspekt dieser Arbeit wider. Im darauffolgenden praktischen Teil wird bei konkrete Implementierung dargestellt und detailliert beschrieben.

Der entwickelte Prototyp ermöglicht dem Benutzer das Einlesen eines Modells, das den Funktionsumfang einer Webseite darstellt. Anschließend können darauf basierend Abläufe für Testfälle spezifiziert und wahlweise als Testskript für die Frameworks JUnit oder TestNG exportiert werden. Zusammenfassend kann gezeigt, dass der im Rahmen dieser Arbeit entwickelte Prototyp durch die Benutzung eines modellbasierten Testansatzes die Spezifikation und Generierung von Testfällen erleichtert und dadurch die Robustheit und Konsistenz von Testfällen erhöht wird. Abschließend wird ein Ausblick auf die weitere Vorgehensweise gegeben, wobei sowohl technische Aspekte, als auch eine praxisrelevante Evaluierung durch Benutzer näher betrachtet werden.

## Schlüsselwörter

Softwaretesten, GUI-Testen, modellbasiertes Testen, GUI, Quellcodegenerierung, Eclipse RCP

# Abstract

The procedure of software testing and quality management is an essential part of today's software development process and contributes to the success of a project. Due to the volatility of components, requirements and in particular user interfaces, testing these elements becomes more complex and time-consuming. Especially modifications of the user interface may affect the consistency of test cases and test scripts, as they are highly dependent on the user interface's elements. To overcome this problem, a model-based approach, where a model represents an element of the user interface, could be helpful.

In the context of this thesis a model based approach for user interface testing is planned and implemented. The application eases the specification of test cases and supports the user during the creation of functional tests without the requirement of any programming skills. The usage of the prototype enables the definition of test cases in relation to a model referring to the user interface. Furthermore, the user is able to export the defined test cases and integrate them into an established test process.

The initial chapters of this thesis provide a short overview on the required fundamentals. The succeeding chapters are split into a theoretical and a practical part. The theoretical part consists of the planning and analysis of the concept, requirements and the architecture as well as a heuristic evaluation of sketches of the user interface. The practical part provides explanations and representations about the implementation of the project.

By using the prototype developed in this thesis with a provided model representation of a website as input for the application, a typical user can specify concrete procedures for test cases and generate these procedures as test cases for either the JUnit or TestNG test framework. It can be concluded that the prototype simplifies the specification and generation of test cases by applying a model based approach that increases the robustness and consistency of test cases. Finally some technical improvements as well as a practice-orientated evaluation conducted by potential users are recommended as future tasks.

## Keywords

Software Testing, GUI Testing, Model-based Testing, GUI, Source Code Generation, Eclipse RCP

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Quellcodeverzeichnis

# Glossar

**Adapter** Übersetzt abstrakte Testfälle in konkrete Funktionsaufrufe des zu testenden Systems [56]. 14, 15

**Anforderungen** beschreiben zu erfüllende Eigenschaften der Software. Dabei werden Anforderungen häufig in:

- funktionale Anforderungen: definieren Funktionen, die die Software beinhalten muss. Funktionale Anforderungen werden bereits in der Planung nach ihrer Relevanz kategorisiert. Hier wird zwischen **MUST-HAVE** und **NICE-TO-HAVE** Funktionen unterschieden.
- nicht-funkionale Anforderungen: definieren Eigenschaften der Software.

unterteilt . 33, 37, 52, 53, 67

**Bibliotheken** stellen eine Sammlung von Hilfsprogrammen oder Funktionen dar, die thematisch gruppiert sind und im Rahmen anderer Programme verwendet werden. Bibliotheken sind keine eigenständigen Programme, sondern dienen nur zur Erweiterung und erfüllen das Prinzip der Datenkapselung. 61, 65, 71

**Branding** auch Customization, bezeichnet die individuelle Anpassung des Aussehens der Software. Besondere Bilder, Icons oder Texte werden somit in das Programm integriert. 38, 68

**Datenkapselung** bezeichnet das Verbergen von Daten vor dem Zugriff von außen. Ein direkter Zugriff auf interne Datenstrukturen soll vermieden werden, stattdessen sollen definierte Schnittstellen den Zugriff ermöglichen. 36, 67

**Debuggen** ist das testweise Ausführen eines Programms mit der Intention Fehler zu finden. Mittels eines Debuggers ist es möglich, Programmcode bis zu einer gewissen Stelle auszuführen, um explizit nach semantischen Fehlern zu suchen. 44

**Document Object Model** definiert eine Konvention für die Baumdarstellten von Objekten in HTML und XML. 71

**Drag&Drop** beschreibt eine Interaktionsform mit einer Benutzeroberfläche. Dabei werden Elemente aus einem Bereich mittels des Mauszeigers selektiert und mit gedrückter Maustaste in einen anderen Bereich gezogen. 34, 47, 49, 56, 79

**Dummy** ein Platzhalter einer im Zuge des Testens benötigten, jedoch nicht relevanten Variable. Beispiel: eine Liste muss ein Element enthalten, das Element selbst ist jedoch irrelevant [5]. 6

**Erweiterungspunkt** ist ein spezifizierter Punkt, an dem andere Plug-ins hinzugefügt werden können [49]. Die Funktionalität anderer Plug-ins wird über Erweiterungspunkte hinzugefügt . 28, 37

**Graphische Benutzeroberfläche** engl. graphical user interface (GUI).
bezeichnet die Benutzeroberfläche eines Programms. Der Benutzer kann mittels graphischen Symbolen mit dem Programm interagieren. 41

**Integrierte Entwicklungsumgebung** engl. integrated development environment (IDE).
ist ein Programm zur Unterstützung von Programmieren im Rahmen der Softwareentwicklung und soll dem Programmierer häufig wiederkehrende Aufgaben abnehmen sowie einen schnellen Zugriff auf wichtige Funktionen bieten. 43

**Internationalisierung** , auch mit *i18n* abgekürzt, bezeichnet die Möglichkeit ein Programm in mehreren Sprachen anzubieten.. 67

**Jetty** ist ein Webserver und Servlet-container. Jetty ist ein der Eclipse-Foundation zugehöriges Open-Source Projekt.. 65

**Kopplung** beschreibt den Grad der Abhängigkeit zwischen mehreren Komponenten. Eine lose Kopplung bedeutet demnach eine geringe Abhängigkeit und ist im Rahmen der Softwareentwicklung erstrebenswert. 41, 61

**Lokalisierung** , auch *L10n* abgekürzt, bezeichnet einen Teil der Struktur des Programms. Dieser soll so aufgebaut sein, dass beispielsweise Internationalisierung ohne Veränderungen im Quellcode umgesetzt werden kann.. 67

**Look&Feel** bezeichnet das standardisierte Erscheinungsbild einer Benutzeroberfläche [**duden**]. Natives Look&Feel bezeichnet das Aussehen der Benutzeroberfläche gemäß des zugrundeliegenden Betriebssystems. 26, 28, 35, 68

**Mock** ersetzt die reale Implementierung eines Objekts im Zuge des Modultests, um isoliertes Testen zu ermöglichen [5]. 6

**Separation-of-Concerns** beschreibt eine Trennung der Verantwortlichkeiten innerhalb eines Softwaresystems. Die einzelnen Module des Systems sollen soweit voneinander separiert sein, dass sie voneinander unabhängig sind. 36, 38, 67

**Stub** ersetzt die noch nicht umgesetzte Implementierung einer Komponente im Zuge von Komponenten- oder Integrationstests [51]. 6

**Testabdeckung** bezeichnet die Menge der während eines Tests durchlaufenen Elemente im Quellcode im Verhältnis zur Gesamtanzahl an Elementen [8]. Beispielsweise kann ein Testfall gefordert werden, der eine Schleife oder Bedingung einfach oder mehrfach durchläuft. 8, 9, 13

**Transformationstool** benutzt verschiedene Templates und Mappings, um den abstrakten Testfall in ein ausführbares Testskript umzuwandeln [56]. 15

**UML** bezeichnet die Unified Modeling Language (UML). Eine graphische Modellierungssprache, die dazu genutzt wird, Teilbereiche von Softwaresystemen zu Spezifizieren, Konstruieren, Visualisieren und Dokumentieren. Das Resultat der Modellierung wird in Form von Modellen und Diagrammen repräsentiert [29]. 47

# Abkürzungen

**CSS** Cascading Style Sheets. 72

**DOM** Document Object Model. 71

**EMF** Eclipse Modeling Framework. 28

**GEF** Graphical Editing Framework. 2, 26, 28, 29, 57

**GMF** Graphical Modeling Framework. 28

**GUI** Graphische Benutzeroberfläche. 32, 42–46, 66, 72

**HTML** Hypertext Markup Language. 17, 65

**IBM** International Business Machines Corporation. 26

**IDE** Integrierte Entwicklungsumgebung. 35, 43–47

**ISO** International Organization of Standardization. 9, 18

**ISTQB** International Software Testing Qualifications Board. 5, 9, 18

**JSON** JavaScript Object Notation. 76

**MVC** Model-View-Controller. 29, 57

**NASA** National Aeronautics and Space Administration. 26

**OSGi** Open Services Gateway initiative. 27, 28

**RCP** Rich Client Platform. 2, 26–28, 30, 36, 37, 43, 52, 68, 69, 72

**SWT** Standard Widget Toolkit. 28, 56

**UCD** User Centered Design. 21, 22

**UI** User Interface. 28

**UML** Unified Modeling Language. 14, 28, 34, 47

**XML** Extensible Markup Language. 16, 17, 64, 65, 71

# 1 Einleitung

## 1.1 Problemstellung und Motivation

Die Tätigkeit des Testens nimmt in Software-Projekten einen immer größeren Stellenwert ein und ist fester Bestandteil der Softwarequalitätssicherung [21]. Unterschiedliche Testtypen stellen dabei verschiedene Anforderungen an den Tester. Es ist wichtig, dass bestehende Tests effizient verwaltet, adaptiert sowie automatisiert ausgeführt werden können. Bestehende Frameworks, Tools und Bibliotheken zum Testen von Benutzeroberflächen unterscheiden sich stark in ihren Realisierungsansätzen. Hier existieren skriptbasierte Ansätze, welche vor allem Programmierkenntnisse erfordern, bis hin zu Ansätzen, die Interaktionen des Benutzers aufzeichnen und anschließend automatisiert wiedergeben.

Besonders bei aufgezeichneten Testskripts ergibt sich die Problematik, dass sich ihre Wartung und Adaption als sehr zeitaufwändig und schwierig herausstellen kann [37]. Veränderungen am zugrundeliegenden System können in der Anpassung der Testfälle bis hin zur kompletten Neuerstellung dieser resultieren [1]. Um diesem Problem entgegenzuwirken ist modellbasierters Testen als Ansatz zur (teil-) automatisierten Erstellung von Testfällen Gegenstand von aktuellen wissenschaftlichen Untersuchungen [1, 53]. Modellbasiertes Testen verwendet als Grundlage für die automatisierte Erstellung der Testfälle detaillierte Modellbeschreibungen und Spezifikationen des zu testenden Systems. Die Modelle schaffen zudem einen gewissen Grad an Unabhängigkeit zwischen den Testfällen und der Benutzeroberfläche, da deren Modifikation nicht zwingend die für die Testfälle relevante Struktur des Modells beeinflusst.

In der Praxis werden modellbasierte Verfahren für die automatisierte Erstellung von Tests für Benutzeroberflächen aufgrund fehlenden Expertenwissens und eines Mangels an verfügbaren Werkzeugen kaum eingesetzt [53]. Ein derartiger Ansatz würde jedoch den Komfort und die Effizienz der automatisierten Testfallerstellung mit der Flexibilität und Adaptivität der Modelle hinsichtlich Veränderungen des Grundsystems kombinieren. Erfolgt die Spezifikation von Testfällen dabei über eine graphische Benutzeroberfläche, ergeben sich weitere Vorteile für den Benutzer. Expertenwissen bezüglich einer oder mehrerer Programmiersprachen sowie Kenntnisse über die inneren Strukturen der zu testenden Applikation sind für den Benutzer nicht erforderlich. Die Bereitstellung einer Benutzeroberfläche bietet zwar eine benutzerfreundliche und intuitive Arbeitsweise, allerdings sollte der Benutzer mit Grundlagen des Softwaretestens vertraut sein sowie grundlegende Kenntnisse über Testframeworks und deren spezifische Konstrukte besitzen, damit eine effiziente Arbeitsweise und Unterstützung für den Prozess der Softwarequalitätssicherung erreicht wird.

Ausgangssituation für die Untersuchung dieses fehlenden Ansatzes sind Java-Klassen, die als Modelle der Benutzeroberflächen von Webapplikationen fungieren und für die Untersuchung näher betrachtet werden. Derartige Modelle können entweder manuell oder von gängigen Webtestingframeworks (wie z.B. [45]) erzeugt werden. Der vorgestellte Ansatz soll jedoch nicht auf bestimmte Modelle beschränkt werden, sondern durch einen generischen Ansatz sämtliche Java-Klassen als Eingabeformat akzeptieren und weiterverarbeiten. Eine prototypische Implementierung für die modellbasierte Erstellung von Testfällen für Benutzeroberflächen soll eine Untersuchung der Praxistauglichkeit eines solchen Ansatzes ermöglichen.

## 1.2 Zielsetzung

Das Ziel dieser Arbeit besteht im Entwurf und der prototypischen Implementierung einer Applikation, welche einen modellbasierten Ansatz zur Erstellung von Tests von Benutzeroberflächen verfolgt. Diese Applikation soll die Spezifikation von Testfällen vereinfachen und auch Testbeauftragten ohne Programmierkenntnisse die Erstellung von Testfällen ermöglichen. Die Modellrepräsentationen der Benutzeroberflächen werden dabei durch kompilierte Java-Klassen dargestellt und sollen von der hier umgesetzten Applikation weiterverarbeitet werden.

Das Modell soll für den Benutzer aufbereitet werden, sodass dieser über eine einfach gestaltete und intuitiv zu bedienende Benutzeroberfläche konkrete Abläufe für das zu testende System definieren kann. Abschließend sollen die spezifizierten Abläufe als ausführbare Testfälle wahlweise für das Testframework JUnit oder TestNG exportiert und in einen bestehenden Testprozess integriert werden können.

Zur Erreichung dieses Ziels werden sowohl theoretische als auch praktische Methoden angewandt. Der theoretische Teil umfasst dabei die Planung und den Entwurf des Projekts, beginnend mit einer Anforderungsanalyse, einem Architekturkonzept sowie Skizzen der geplanten Benutzeroberflächen und einer heuristischen Evaluierung derselben.

Der praktische Teil umfasst die Umsetzung der Applikation basierend auf den im theoretischen Teil angefertigten Entwürfen. Die Implementierung soll als Eclipse Rich Client Plattform (RCP)-Applikation erfolgen und stellt den Schwerpunkt dieser Arbeit dar.

Als Resultat dieser Arbeit wird ein ausführbarer Prototyp erwartet, der den beschriebenen modellbasierten Ansatz zur Teilautomatisierung der Testfallerstellung unterstützt und für weitere Evaluierungen bezüglich der Praxistauglichkeit eines derartigen Ansatzes herangezogen werden kann.

## 1.3 Aufbau der Arbeit

Der Anfang dieser Arbeit behandelt sowohl Grundlagen als auch Definitionen der zugrundeliegenden Themenbereiche und dient als Basis für die nachfolgenden Kapitel. Kapitel 2 umfasst die Grundlagen des Themenbereichs um das Testen von Software. Hierbei werden zum einen die Begriffe Testen und Testarten definiert und zum anderen die Grundsätze des Testens erläutert. Weiters wird auf die Notwendigkeit des Testens eingegangen und gängige Realisierungsansätze werden vorgestellt. Kapitel 3 bietet einen Einblick in das Gebiet der Usability und des Usability-Engineerings. Dieses Kapitel fasst die Relevanz und Bedeutung von Usability für Softwareprojekte zusammen und zeigt geläufige Heuristken und Richtlinien sowie Evaluierungsmethoden. Kapitel 4 stellt die Technologie der Eclipse Rich Client Plattform (RCP) sowie das Graphical Editing Framework (GEF) vor, da diese Technologien die Grundlage der Implementierung des in dieser Arbeit behandelten Projekts darstellen.

In Kapitel 5 wird näher auf die Relevanz und Motivation des Projekts eingegangen. Dieses Kapitel definiert den Begriff der Softwarevolatilität und leitet mit einem Überblick über die Projektbeschreibung und Anforderungsanalyse die Planungsphase des Projekts ein. Weiters wird im Zuge der Planung in Kapitel 6 ein Entwurf der Systemarchitektur abgebildet und eine detaillierte Beschreibung sowie eine heuristische Evaluierung von Skizzen der geplanten Benutzeroberfläche dargestellt.

Kapitel 7 beschreibt die Implementierungsphase dieser Arbeit. Hier wird die Umsetzung der vorgestellten Applikation beschrieben sowie näher auf Komplikationen während der Entwicklung und Besonderheiten der verwendeten Technologien eingegangen.

Kapitel 8 fasst den Inhalt und die wesentlichen Erkenntnisse dieser Arbeit zusammen und stellt die weitere Vorgehensweise und zukünftige Erweiterungen des Projekts in Ausblick.

# 2 Grundlagen: Das Testen von Software

Im folgenden Kapitel werden die Grundlagen des Testens von Software näher erläutert. Es wird auf die Motivation und Relevanz von Software Tests eingegangen, sowie mittels Definitionen und Begriffserklärungen dargestellt, worum es sich bei Software Tests handelt, worum nicht und welche Ziele dabei verfolgt werden. Anschließend werden die wesentlichen Grundsätze des Testens erklärt und die grundlegende Vorgehensweise im Testprozess erläutert. Es folgt eine Kategorisierung der Testarten, sowie die Begriffserklärung und Relevanz der Testautomatisierung. Daraufhin folgt ein Ausblick über unterschiedliche Ansätze von Softwaretests sowie eine kurze Vorstellung gängiger Testframeworks.

## 2.1 Motivation und Relevanz

Wurde das Testen von Software bisher meist als lästige Pflicht oder Nice-To-Have Feature angesehen und automatisch mit hohen Kosten und Zeitaufwand in Verbindung gebracht, so hat die Vergangenheit klar gezeigt, dass Softwaretests unerlässlich sind [21]. Eines der bekanntesten Beispiele für mangelhafte Tests stellt die Explosion der Rakete Ariane5 [18] im Jahr 1996 dar. Allerdings ist es nicht notwendig, Softwaremängel so weit in der Vergangenheit zu suchen. Aktuellere Beispiele sind der Telekom-Ausfall im Jahr 2004 [52], oder der Skype-Ausfall 2010 [38].

Das Testen von Software ist heutzutage bereits fester Bestandteil und eine der wichtigsten Tätigkeiten des Entwicklungsprozesses und wird als Expertentätigkeit angesehen. Nur mit genauer und effektiver Planung ist es möglich, die Komplexität eines Systems umfassend zu testen [21]. Allerdings ist eine klare Definition der Tätigkeit des Testens nicht unbedingt trivial zu erfassen. So definieren Denert und Siedersleben in [11] Software Testen wie folgt:

> *„[Software Testen ist der] überprüfbare und jederzeit wiederholbare Nachweis der Korrektheit eines Softwarebausteines relativ zu vorher festgelegten Anforderungen“*

Eine weitere Definition liefert Myers [34]:

> *„[Software] Testing is the process of executing a program with the intent of finding errors.“*

Softwaretests zielen also darauf ab, Fehler zu finden und sind jederzeit beliebig oft wiederholbar. Sie gehören somit in den Bereich der Qualitätssicherung. Myers [34] erkennt hier bereits eine umgedrehte Denkweise. Er zeigt, dass ein Softwaretest nicht beweist, dass ein Programm fehlerfrei ist, bzw. sagt, dass ein Test nur dann als erfolgreich angesehen wird, wenn ein Fehler gefunden wird.

Praktisch gesehen ist es unmöglich, komplexe Softwaresysteme zu schaffen, die komplett fehlerfrei sind. Selbst wenn Tests keine Fehler mehr aufzeigen ist nicht garantiert, dass keine Fehler mehr auftreten können. Dies ist allerdings auch nicht erforderlich [21]. Die Software muss lediglich daraufgehend getestet sein, dass beim typischen Gebrauch und bei naheliegenden Fehlanwendungen keine Schäden entstehen können [8]. Wichtig ist demnach eine möglichst hohe funktionale und nicht-funktionale Abdeckung eines Tests [21].

Neben dem Auffinden von Fehlern ist auch die **Verifikation** und **Validierung** von Software Ziel des Testens. **Verifikation** testet ein Modell gegen das tatsächliche Produkt. Verifikation soll die Frage beantworten, ob das Produkt der Spezifikation entspricht. Bei der **Validierung** soll gewährleistet werden, dass das Produkt sowohl für seinen vorgesehenen Einsatzzweck, als auch den Benutzer geeignet ist. Es soll also garantiert werden, dass das Produkt für den vorgesehenen Zweck passt [8, 39].

## 2.2 Grundsätze

Trotz der Wichtigkeit von Softwaretests kann die Tätigkeit des Testens mitunter sehr zeitintensiv ausfallen. Um allgemeine Fallstricke zu vermeiden, hat das International Software Testing Qualifications Board (ISTQB) gewisse Grundsätze des Testens spezifiziert. Diese Grundsätze sind allgemeine Aussagen, deren sich ein Tester bzw. Test-Manager bewusst sein sollte, um sich nicht zu verstricken. Nachfolgend wird ein Überblick über die Grundsätze der ISTQB [20] gegeben:

Testen zeigt Fehler auf:
: Durch Tests sollen Fehler gefunden und die Wahrscheinlichkeit, dass unentdeckte Fehler auftreten, minimiert werden. Wenn keine Fehler gefunden werden, bedeutet dies nicht, dass keine mehr vorhanden sind.

Vollständiges Testen ist nicht möglich:
: Sämtliche Kombinationen von Eingaben, Vorbedingungen und Systemzuständen zu testen ist nicht umsetzbar und wäre auch nicht praktikabel. Stattdessen müssen gewisse Risiken eingegangen und der Testfokus auf gewisse Prioritäten gelegt werden.

Frühzeitig Testen:
: Testen ist keine abschließende Tätigkeit. Je früher im Entwicklungszyklus mit dem Testen begonnen wird, desto besser. Frühzeitig gefundene Fehler sind einfach und günstig zu beheben, ohne gleichzeitig Auswirkungen auf andere Teile des Systems fürchten zu müssen.

Häufungen beachten:
: Fehler sind nicht gleichmäßig verteilt. Werden in einem Modul einige Fehler gefunden, sind dort meist weitere zu erwarten.

Veränderung statt Wiederholung:
: Das wiederholte Ausführen von Testfällen wird keine neuen Defekte identifizieren. Stattdessen sollten Testfälle regelmässig verändert und erweitert werden, um unterschiedliche Programmteile anzusprechen.

Testen ist kontextabhängig:
: Unterschiedliche Systeme haben unterschiedliche Anforderungen und müssen auch unterschiedlich getestet werden.

Trugschluss: Fehlerlose Systeme sind auch brauchbar:
Defekte zu finden und zu beheben garantiert nicht, dass das Produkt den Anforderungen und Bedürfnissen des Benutzers entspricht. Andere nicht-funktionale Anforderungen, z.B. Usability, sind ebenso ausschlaggebend.

## 2.3 Teststufen

Das Entwickeln von Software bedarf aufgrund deren steigenden Umfangs und zunehmender Komplexität sowie hoher Qualitätsansprüche und knapp gesetzten zeitlichen Fristen einer immer strikteren Planung [21]. Eine wohldefinierte Abfolge der Software-Entwicklungsaktivitäten und eine darauf abgestimmte Menge an qualitätssichernden Maßnahmen ist von entscheidender Bedeutung für ein Projekt. Um bereits frühzeitig mit dem Testprozess beginnen zu können empfiehlt sich ein Vorgehen mit korrelierenden Phasen. Den durchlaufenen Entwicklungsphasen stehen dabei Prüfphasen gegenüber, welche sich nach [21] in vier Teststufen gliedern und die entstandenen Dokumente und Projektartefakte mit dafür geeigneten Techniken auf Fehler überprüfen.

In der nachfolgenden Auflistung werden die den Entwicklungstätigkeiten gegenüberstehenden prüfenden Testphasen gemäß [20, 39] näher erläutert.

- Komponententests
  Komponententests, oder auch *Modultests*, stellen die tiefgreifendste Testphase dar und werden auch als *low-level-Tests* bezeichnet [8]. Ziel von Komponententests, die häufig vom entsprechenden Entwickler selbst durchgeführt werden, ist die Überprüfung einzelner Komponenten hinsichtlich funktionaler und nicht-funktionaler Anforderungen. Ein wichtiges Kriterium hierfür ist Isolation. Zwar arbeiten im späteren Verlauf viele Komponenten zusammen, dennoch müssen sie isoliert getestet werden. Dies erlaubt es, gefundene Fehler eindeutig einer Komponente zuzuordnen [21].

  Um isoliertes Testen zu ermöglichen, steht dem Tester die Benutzung von so genannten Platzhaltern zur Verfügung. Platzhalter ermöglichen das isolierte Testen von abhängigen Komponenten durch deren Simulation und werden je nach ihrem Funktionsumfang als *Mock*, *Stub* oder *Dummy* bezeichnet. Zusätzlich zu Platzhaltern ist die Benutzung von Testtreibern üblich. Diese erleichtern die Ausführung und Automatisierung der Komponententests und sind für viele Programmiersprachen als Framework vorhanden (siehe 2.7) [8].

- Integrationstests
  Sind die einzelnen Komponenten isoliert getestet, muss auch die Zusammenarbeit zwischen den Modulen im Rahmen eines Integrationstests getestet werden. Spillner und Linz beschreiben [51], den Integrationstest als die Überprüfung der fehlerfreien Kommunikation und Kompatibilität von Komponenten, der auf den vorangegangenen Komponententests aufbaut. Umfassende Komponententests erhöhen die Kontrollierbarkeit der Integrationstests, allerdings können auch hier noch vereinzelt Fehler, die ein Modultest aufdecken sollte, gefunden werden. Anders ist es bei Nebenwirkungen von Modulfehlern. Diese werden besonders im Rahmen von Integrationstests aufgedeckt [39].

  Für die Integration und Tests existieren abhängig von der Entwicklungsreihenfolge der Komponenten unterschiedliche Integrationsstrategien [8]. Diese werden in *horizontale* und *vertikale Integration* unterteilt [21].

Unter horizontaler Integration versteht man nach [21] ein schichtweises Zusammenführen der Komponenten. Dies kann als *Top-Down Integration* erfolgen, wobei die oberste Schicht, also die Benutzerschnittstelle, zuerst integriert wird und die drunterliegenden Schichten mittels Platzhaltern simuliert werden. Die *Bottom-Up Integration* stellt das exakte Gegenteil zur *Top-Down Integration* dar. Eine weitere horizontale Integrationsstrategie ist die *Big-Bang Integration*, bei der das gesamte System gleichzeitig integriert wird und keinerlei Platzhalter Verwendung finden [21]. Eine vertikale Integration ist eine funktionsorientierte Strategie. Hier wird eine Teilmenge des fertigen Systems über alle Schichten gleichzeitig integriert,wodurch ein funktionales Teilsystem entsteht, das keinerlei Platzhalter benötigt und als früher Prototyp fungieren kann. Somit ist aus Sicht der Softwaretests sowie des Usability-Engineering (siehe Kapitel 3.3) diese Integrationsform der horizontalen vorzuziehen [21].

- Systemtests
  Nachdem die einzelnen Bestandteile des Systems sowohl separat, als auch in integrierter Form getestet worden sind, ist das Softwaresystem nun nahezu vollständig. Beim Systemtest liegt der Fokus auf Verifikation der Anforderungen und Überprüfung des fertigen Systems [8, 20]. Sneed [50] erkannte jedoch, dass ein System mehr als die Summe seiner Einzelteile ist. Er definiert ein System zusätzlich als Summe aller Beziehungen zwischen Einzelteilen, sowie weiters als Summe der durch die Integration entstandenen Effekte.

  Als Grundlage für Systemtests dienen somit die Spezifikation des Produkts, Anwendungsfälle, sowie Beschreibungen des Systems und Erfahrung der Tester [21]. Die inneren Strukturen, also die Komponenten und deren Schnittstellen, werden beim Systemtest nicht berücksichtigt. Es handelt sich also um einen Black-Box-Test (siehe 2.4).

  Besonders wichtig ist die Testumgebung. Diese soll der späteren Produktivumgebung möglichst genau entsprechen. Dadurch lassen sich umgebungsspezifische Fehler vermeiden sowie Kennzahlen bezüglich Performance und zeitlichen Bewertungen erschließen [8].

- Abnahmetests
  Anders als bei bisherigen Softwaretests zielt der Abnahmetest nicht mehr darauf ab, Fehler im System aufzudecken. Hier steht nicht mehr die Qualitätssicherung, sondern die Zufriedenheit und Akzeptanz des Benutzers im Vordergrund. Der Abnahmetest stellt eine Sonderform des Systemtests dar und wird daher auch als Akzeptanztest bezeichnet [21].

  Der Abnahmetest bezeichnet das Ende der Entwicklungsphase und den bevorstehenden Einsatz des Produkts. Hier werden im Rahmen der Abnahme, unter Beobachtung des Benutzers, in der dafür vorgesehenen Produktumgebung Tests und Demonstrationen durchgeführt, um das System gegen die ursprünglich definierten Anforderungen zu prüfen [8].

Die zuvor erwähnten Teststufen stellen dabei die gebräuchlichsten der korrelierenden Phasen dar [20]. Andere Variationen schließen Phasen zusammen oder reduzieren deren Umfang. Die genaue Anwendung und Umsetzung kann je nach Anforderungen und Prioritäten im Projekt verändert und anpasst werden.

## 2.4 Testarten

Eine treffende Definition von Testarten oder Testtypen liefert Cleff in [8]:

> *„Testarten bzw. Testtypen stellen eine Gruppierung der Einzeltests aufgrund unterschiedlicher Testziele und unterschiedlicher Einbindung im Entwicklungsprozess dar."*

Testarten beschreiben demnach verschiedene Techniken und deren Art und Weise zu testen. Sie können auf allen Teststufen zum Einsatz kommen und unterscheiden sich innerhalb der Stufen in ihren Schwerpunkten. Hauptsächlich wird zwischen Tests bezogen auf Funktionalität, Testen der nicht-funktionalen Eigenschaften, Testen der Architektur oder Testen im Zusammenhang mit Änderungen unterschieden [8].

Zusätzlich zu den von Cleff [8] genannten Unterscheidungen ist es notwendig, eine weitere Differenzierung der Testarten anzustreben. Abhängig davon, wer die Tests ausführt bzw. wie viel über das zu testende System bekannt ist, ist eine zusätzliche Unterteilung in Black-Box und White-Box Tests unerlässlich.

- Black-Box-Tests
  Beim Black-Box-Test werden Tests ohne Kenntnisse über die innere Funktionsweise des Systems entwickelt. Das System fungiert also als schwarze, undurchsichtige Box und reagiert nur auf Eingaben. Der Tester benötigt keinerlei Wissen über den Quellcode und kann nur mit den bekannten Schnittstellen des Systems arbeiten [13]. Die zu erstellenden Testfälle müssen aus der Spezifikation abgeleitet werden. Oftmals ist es notwendig, die Spezifikation dementsprechend zu formalisieren, wodurch sich der Nebeneffekt der Black-Box Tests ergibt. Schwachstellen und Unklarheiten innerhalb der Spezifikation werden hier erneut aufgedeckt [8, 19].

- White-Box-Tests
  Anders als beim Black-Box Test, liegt der Fokus beim White-Box (oder auch Glass-Box Test) auf dem Inneren der Box. Die innere Funktionsweise des zu testenden Teilsystems ist bekannt, Wissen um den Quellcode der Software ist zum Erstellen der Testfälle notwendig [19].

  Der Testfokus von White-Box Tests liegt beim Erreichen bzw. Bestimmen des Grades der Testabdeckung. Dieser kann je nach Elementtyp zwischen Anweisung, Pfad oder Bedingung unterschieden werden. Durch Testabdeckung kann in Entwicklungsumgebungen gemessen und farbig hervorgehoben werden, wie gut bzw. wie umfangreich bisherige Testfälle ausfallen und welche Codesegmente weitere Testfälle verlangen. Durch diese direkte Darstellung wird klar, in welchem Bereich weiterer Testbedarf besteht. Der Tester wird somit dazu angehalten, überlegt vorzugehen [8].

### 2.4.1 Funktionales Testen

Mittels funktionaler Tests werden das System und die spezifizierten Anforderungen auf Vollständigkeit und Korrektheit geprüft [8], es findet also eine funktionale Verifikation statt. Zusätzlich zu den bereits spezifizierten Anforderungen können einige implizite, nicht dokumentierte Anforderungen aufkommen. Diese ergeben sich gemäß [8] unter anderem aus dem aktuellen Stand der Technik oder Best Practices und sind bei funktionalen Tests ebenfalls zu beachten.

Wie bereits in den Grundsätzen des Testens in Abschnitt 2.2 erwähnt, ist vollständiges Testen nicht möglich. Daher ist es notwendig eine Testfallselektion vorzunehmen [21]. Die gewählte Anzahl der Testfälle soll also einen möglichst hohen Testabdeckungsgrad ermöglichen und gleichzeitig die höchste Wahrscheinlichkeit, Fehler zu finden, aufweisen [21].

Laut [20] werden funktionale Tests in zwei Ansätze unterteilt und können in sämtlichen Teststufen (siehe 2.3) zum Einsatz kommen. *Spezifikationsbasierende Tests* nutzen die definierten Anforderungen als Grundlage und überprüfen diese auf Korrektheit. Es handelt sich hierbei um Black-Box Tests [21]. *Strukturelle Tests* benötigen Kenntnis über der Systemaufbau und streben, als White-Box-Tests einen möglichst hohen Abdeckungsgrad an [21].

### 2.4.2 Nicht-Funktionales Testen

Wie das ISTQB [20] zeigt, beziehen sich nicht-funktionale Tests auf die nicht-funktionalen Anforderungen des Systems. Es wird also getestet und gemessen *wie* gut das System funktioniert. Getestet werden verschiedene Qualitätsmerkmale des Systems. Die International Organization of Standardization (ISO) hat hierfür eine Liste an Qualitätsmerkmalen [24] definiert, welche fünf Kategorien für nicht-funktionales Testen enthält und welche aus mehreren Subkategorien bestehen.

Diese Kategorien sind gemäß [24]:

**Zuverlässigkeit:** Die Software muss eine bestimmte Leistung erbringen und dieses Niveau aufrechterhalten können. Tests, die auf die Zuverlässigkeit abzielen, können weiters zwischen Fehlertoleranz, Wiederherstellbarkeit oder Konformität unterscheiden.

**Usability:** Beschreibt wie einfach es für einen Benutzer ist, mit der Software umzugehen. Diese Testart wird in Kapitel 3.4.2 genauer erläutert.

**Effizienz:** Hier werden vor allem zeitbezogene Tests im Bezug auf Performance, Auslastung, Antwortzeiten oder Ressourcenverteilung durchgeführt.

**Wartbarkeit:** Soll ermitteln wie gut ein System zu warten ist. Es wird gemessen, wie leicht es veränderbar, analysierbar und testbar ist.

**Portabilität:** Gibt an wie leicht das System in andere Umgebungen zu portieren ist und welche Änderungen durchzuführen sind.

Nicht-funktionale Anforderungen sind allerdings gemäß [20] nicht auf diese Kategorien limitiert, eine weiterer Punkt ist zum Beispiel das Testen bezüglich der Systemsicherheit [21]. Nicht-funktionale Tests können in sämtlichen Teststufen benutzt werden und unterscheiden sich abhängig von der getesteten Anforderung. So reicht für einen Effizienz-Test zum Beispiel ein Black-Box-Test aus, während für einen Test bezüglich der Wartbarkeit Einsicht in den Quellcode zwingend erforderlich ist [21].

### 2.4.3 Änderungsbezogenes Testen

Das ISTQB benennt in [20] eine weitere Art des Testens, nämlich änderungsbezogene Tests. Veränderungen oder Modifikationen am System, an einzelnen Komponenten oder der Umgebung können nicht nur die Funktionsweise oder die Struktur des Systems verändern, sondern durch Neben-

wirkungen auch dessen andere Teile beeinflussen. Änderungsspezifische Tests können alle Arten von Testfällen beinhalten und in sämtlichen Teststufen ausgeführt werden [20].

Hierzu wird nach [20] zwischen zwei Ausgangssituationen unterschieden:

Fehlernachtest
: Ein Fehlernachtest wird dann ausgeführt, wenn zuvor ein Fehler aufgefunden wurde und somit ein Testfall fehlgeschlagen ist. Ziel ist es, die jeweilige Aktion durch Verwendung derselben Eingabedaten oder gleicher Systemumgebung exakt nachzustellen. Es soll also mittels gleicher Voraussetzungen versucht werden die Aktion erneut fehlschlagen zu lassen. Ist der Defekt nicht reproduzierbar, bedeutet dies nur, dass dieser Teil der Software funktioniert. Neuerliche Nebenwirkungen in anderen Teilbereichen des Systems können auftreten [51].

Regressionstest
: Der Regressionstest testet auf ungeahnte Auswirkungen nach Änderungen im System. Hier werden Testfälle ausgeführt, die bereits beim vorhergehenden Durchlauf erfolgreich waren. Es wird gemäß [20] sichergestellt, dass sich das System nicht zurückentwickelt. Vollzogene Änderungen sollen keine Auswirkungen auf andere Systemteile haben und die Anforderungen müssen immer noch erfüllt sein [21, 51].

  Laut [21] empfiehlt es sich, bei jeder neuen Version eines Softwaresystems Regressionstests auszuführen. Die Anzahl an Regressionstests sollte dabei stetig mit dem System wachsen und sich mit dem System verändern. Hinzugefügte Funktionalität und nicht mehr verwendete oder gelöschte Features erfordern eine Anpassung der Testfälle. Mit zunehmendem Systemumfang ergibt sich demnach eine steigende Anzahl an Testfällen, die nach [20] als *Test-Suite* bezeichnet werden.

  Eine *Regressions Test Suite* ist eine Menge an Testfällen speziell für Regressionstests. Diese Testfälle decken im Kollektiv die meisten Funktionen des Systems ab, ohne diese im Detail zu testen und werden nach [21] automatisiert ausgeführt.

## 2.5 Testautomatisierung

Wie Jackson in [26] beschreibt, liegt gerade bei großen und komplexen Softwaresystemen der Testaufwand bei 30% - 40% des gesamten Realisierungsaufwands. Die Sicherstellung der Qualität ist allerdings in derartigen Projekten unverzichtbar. Während bei manuell ausgeführten Tests und besonders bei wachsenden und immer komplexeren Anwendungen der Aufwand für die Testaktivitäten zunehmend ansteigt, lassen sich durch überlegtes und gut organisiertes Testmanagement Aufwand und Kosten insbesondere durch Testautomatisierung verringern.

Als Testautomatisierung bezeichnet man das Bestreben, die Durchführung der ansonsten manuellen Testtätigkeiten weitgehend zu automatisieren.

Seidl, Baumgartner und Bucsics liefern in [44] eine treffende Definition:

> *„[Testautomatisierung ist] ein Werkzeug, das es dem professionellen Softwaretester erlaubt, seine kreativen Ideen umzusetzen und auch die großen Testmengen mit vernünftigem Aufwand und in gebotener Zeit zu bewältigen“*

Automatisierung ist vor allem für Tests, deren Exekution eine längere Laufzeit aufweist, geeignet, wobei das Spektrum von Testautomatisierung nahezu alle Tätigkeiten umfasst, die zur Überprüfung von Softwarequalität nötig sind. Oftmals wird Automatisierung nur auf die Erstellung der Testfälle bezogen, allerdings lassen sich auch, wie in [44] beschrieben, die Test-Datenerstellung, Skripterstellung, Durchführung, Auswertung, Dokumentation und Umgebungsherstellung automatisieren.

Wird der Einsatz von Testautomatisierung entschieden, sollte jedoch nicht einzig der wirtschaftliche Faktor der Einsparungen ausschlaggebend sein. Auf menschliche Tester kann keineswegs verzichtet werden. Die Automatisierung umfasst, gemäß [44], nur die manuell ausgeführten Tätigkeiten, nicht aber die intellektuelle, kreative oder intuitive Arbeit eines Softwaretesters, die ebenfalls für die Qualität des Testvorgangs erforderlich sind.

Weiters ist der Einsatz von Testautomatisierung projektabhängig. Kleine Projekte mit nur wenigen Testfällen oder Projekte mit kurzer Projektdauer werden nur wenig spürbaren Nutzen aus Testautomatisierung ziehen, da sich hierbei ein höherer Initialaufwand als bei manuellem Testen ergibt [21]. Ist das automatisierte Testen jedoch bereits in Verwendung ergibt sich, wie in [21] angegeben, ein wesentlich geringerer Aufwand pro Testfall. Auch der Wartungsaufwand steigt im automatisierten Testprozess etwas an, in [26] finden sich jedoch gute Gründe dafür. Jackson [26] bemerkt, dass der Blick auf den gesamten Lebenszyklus des Projekts gerichtet werden soll. Automatisierung erleichtert nachhaltig die Wiederverwendbarkeit und Wiederholbarkeit von Tests und stellt auf lange Sicht eine erhebliche Aufwand- und Kostenreduktion dar. Diese Aussage belegt eine von Dustin, Rashka und Paul in [13] durchgeführte Studie. Tabelle 2.1 zeigt den Umfang dieser Studie und den Prozentsatz der Verbesserungen im jeweiligen Testschritt.

| Testschritte | Aufwand manuelle Tests (in Stunden) | Aufwand automatisierte Tests (in Stunden) | Prozentsatz der Verbesserung |
|---|---|---|---|
| Testplan entwickeln | 32 | 40 | -25% |
| Testverfahren definieren | 262 | 117 | 55% |
| Testausführung | 466 | 23 | 95% |
| Analyse der Ergebnisse | 117 | 58 | 50% |
| Überwachung und Korrektur | 117 | 23 | 80% |
| Berichte erstellen | 96 | 16 | 83% |
| Gesamtdauer | 1090 | 277 | 75% |

**Tabelle 2.1:** Verbesserungen durch automatisiertes Testen nach einer Studie in [13].

Wie aus Tabelle 2.1 ersichtlich wird, ist für die Verwendung von Testautomatisierung ein erhöhter Initialaufwand erforderlich. Allerdings ist ebenso deutlich zu erkennen, dass in allen anderen Testtätigkeiten eine deutliche Reduktion des Aufwands vorliegt und besonders die Durchführung der Testfälle (Verbesserung um 95%) davon profitiert. Ebenfalls ist anhand der dargestellten Werte der allgemeine Nutzen und das Potential von Testautomatisierung ersichtlich.

Alle in 2.3 beschriebenen Teststufen können im Rahmen der Testautomatisierung umgesetzt werden. Seidl, Baumgartner und Bucsics unterteilen in [44] den Einsatz von automatisierten Tests nach der Häufigkeit deren Ausführung.

- Ständig automatisiert ausgeführte Tests sind laut [44] vor allem **Komponententests**, die üblicherweise nach einem neuen Build automatisiert ausgeführt werden. In diesem Kontext bekommen sie jedoch mehr den Charakter eines Regressionstests. Ziel ist es nicht, neue Funktionen zu testen, sondern sicherzustellen, dass die getätigten Änderungen keine unerwünschten Nebeneffekte ausgelöst haben.
- Häufig ausgeführte Tests, wie **kleinere Integrations-** oder **Systemtests**, sowie **umfassende Komponententests** können in regelmäßig kurzen Abständen, laut [44] meist mehrere Tage, automatisiert ausgeführt werden und liefern ebenfalls mit Test-Charakter Bestätigung über die ordnungsgemäße Funktionalität des Systems.
- Als **selten** ausgeführte Tests nennen Seidl, Baumgartner und Bucsics [44] **vollständige Integrations-** und **Systemtests**. Trotz ihrer Zeitintensivität sollten diese Testfälle zumindest einmal pro ausgelieferter Version durchgeführt werden.

Trotz des Bestreben mittels Testautomatisierung Fehler im System bzw. durch Änderung entstandene Fehler aufzudecken, ist es wichtig, die Testfälle sorgsam auszuwählen. Ein neu erstellter Testfall muss allerdings zumindest einmalig positiv ausgeführt worden sein, bevor er in das Testset der automatisierten Testfälle aufgenommen werden kann, andernfalls kann, wie in [44] beschrieben, nicht sichergestellt werden, dass der Testfall korrekt funktioniert. Ein fehlerhafter Testfall könnte dann den Eindruck eines Fehlers im Gesamtsystem vermitteln und voreilige Reaktionen der Entwickler hervorrufen. Abbildung 2.1 zeigt das übliche Vorgehen bei der Erstellung automatisierter Testfälle.

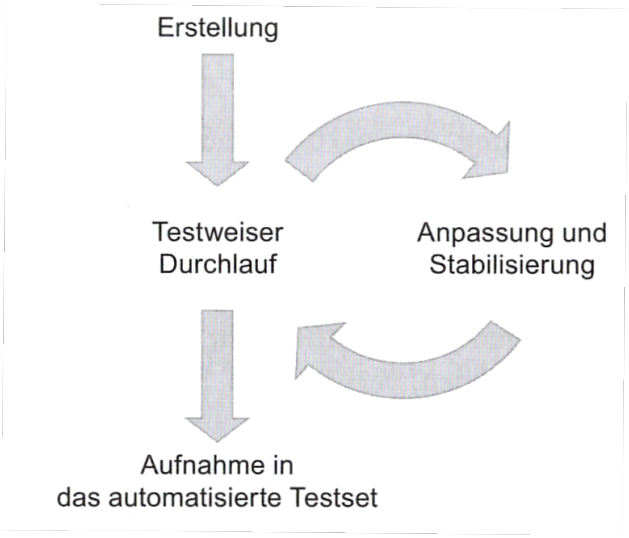



**Abbildung 2.1:** Vorgehen bei der Erstellung automatisierter Testfälle nach [44].

## 2.6 Realisierungsansätze

Abhängig vom Fokus des Projekts, Projektumfang, Projektdauer, beteiligtem Personal und dem aktuellen Projektfortschritt existieren unterschiedliche Herangehensweisen für die Realisierung der Testfälle. Ein Realisierungsansatz beschreibt dabei die einzelnen Schritte vom Entwurf und

der Erstellung der Testfälle, über die Ausführung dieser, bis hin zur Überprüfung der Ergebnisse. Utting und Legeard [56] sowie Graham u. a. [20] geben einen Überblick über unterschiedliche in der Praxis genutzte Ansätze. Nachfolgend werden einige, für diese Arbeit relevante Verfahren vorgestellt und näher erläutert.

### 2.6.1 Manuelles Testen

Beim manuellen Testen werden die Testfälle von Hand aus den Anforderungsdokumenten erarbeitet. Daraus resultiert ein für Menschen verständlicher Testplan, der anzeigt, welche Funktionen wie, in welcher Reihenfolge und wie oft getestet werden sollen. Anschließend werden diese Testfälle ebenfalls manuell durchgeführt. Es findet also eine direkte Interaktion zwischen Mensch und System statt. Die Ergebnisse werden in Form von Berichten festgehalten und das tatsächliche Resultat mit dem erwarteten verglichen [56].

Wie in [56] beschrieben, ist manuelles Testen allerdings sehr zeitintensiv und kann keine genauen Aussagen über die Abdeckung der getesteten Funktionen liefern.

### 2.6.2 Capture/Replay-Tests

Das Ziel von Capture/Replay-Tests ist laut [56] die automatisierte Wiederausführung von Testfällen. Hierzu findet, wie in [39] erläutert, in der Aufnahmephase eine direkte Interaktion mit der Benutzeroberfläche statt. Dabei werden die Aktionen des Testers aufgezeichnet und in der zweiten Phase, der sogenanntenWiederholungsphase, automatisch erneut durchlaufen und getestet. Nachträgliche Veränderungen an der Benutzeroberfläche sollten nach Möglichkeit vermieden werden. Es besteht die Gefahr, dass die Testskripts durch Veränderung funktionsunfähig gemacht werden und somit neu erstellt werden müssten.

### 2.6.3 Skriptbasiertes Testen

Automatisiertes Ausführen der Skripts ist der zentrale Vorteil des skriptbasierten Testens. Wie in [56] dargestellt, ist dieser Ansatz sehr verbreitet und vor allem für Regressionstests vorteilhaft. Ein Testkript ist ein programmierter Test, der das System sowohl kontrolliert als auch beobachtet. Wie bereits in Abschnitt 2.4.3 erwähnt, ist die Wartung und Anpassung von Testskripts unbedingt erforderlich. Bekannte Beispiele für skriptbasierte Tests stellen die Frameworks JUnit (siehe 2.7.1) und TestNG (siehe 2.7.2) dar.

### 2.6.4 Modellbasiertes Testen

Gemäß [42] und [56] ist das Hauptziel von Testansätzen, die mit Modellen arbeiten, die Erstellung der Testfälle zu (teil-) automatisieren. Dadurch ergibt sich eine Kombination mit der Testautomatisierung, in der modellgetriebene Tests ihre größte Wirkung entfalten. Durch die Verwendung von Modellen soll ein besseres Verständnis und mehr Transparenz bezüglich der Spezifikationen erreicht werden. Weiters nennen Roßner u. a. in [42] eine Kostenreduktion in der Testgenerierung und hohe Testabdeckung als Ergebnis von modellbasiertem Testen.

Dalal u. a. kommen in [9] zu dem Schluss, dass modellbasiertes Testen zwar nicht „die Wunderlösung“ repräsentiert, allerdings werden die Dauer und Kosten der Testerstellung reduziert und die Effizienz des Testens erhöht. Diese Methode eignet sich vor allem für Systeme, die sich oft

verändern. In diesem Fall reicht eine simple Anpassung der Modelle aus, um einen neuen Satz an Testfällen zu generieren. Modellbasiertes Testen lässt sich in allen Stufen des Testprozesses verwenden und bildet laut [42] eine sinnvolle Ergänzung zu anderen Methoden.

Modelle stellen in diesem Zusammenhang eine abstrakte Darstellung eines Teilbereichs des Systems dar. Pretschner [41] definiert die Notwendigkeit der Abstraktion, da sonst der Aufwand zur Validierung des Modells dem der Validierung des Systems entsprechen würde. Modelle können in unterschiedlichen Notationen geschrieben werden. Dies kann zum Beispiel in der Unified Modeling Language (UML) erfolgen, allerdings sind auch quellcodebasierte Modelle durchaus verbreitet [41].

Roßner u. a. unterscheiden in [42] zwischen folgenden verschiedenen Modellarten:
**Umgebungsmodell** - Beschreibt Struktur und Verhalten der Systemumgebung.
**Systemmodell** - Stellt den Aufbau, die Funktionalität und das Verhalten des Systems dar.
**Testmodell** - Beschreibt den Test eines Systems und bildet Testentscheidungen ab.

Weiters unterscheiden Roßner u. a. [42] zwischen drei grundsätzlichen Ausprägungen von modellgetriebenen Tests, die jeweils abhängig von den benutzen Modellarten sind:

**Modellorientiertes Testen** - Modelle dienen als Grundlage für das Testdesign, Generatoren müssen nicht zwingend eingesetzt werden. Die Testqualität wird bereits durch das Benutzen von Modellen gesteigert [42].

**Modellgetriebenes Testen** - Modelle werden solange verfeinert und spezifiziert, bis sie zur automatisierten Erstellung von Quellcode herangezogen werden können. Testfälle und andere Testartefakte werden automatisch mittels Generatoren aus den Modellen erzeugt. Ziel dieses Ansatzes ist es, eine Reduktion des Aufwands im Hinblick auf Erzeugung und Wartung von Testfällen zu erreichen, allerdings ist darauf zu achten, dass die Erstellung und Adaption der Modelle den Aufwand des manuellen Testens nicht übersteigt [42].

**Modellzentrisches Testen** - Im Gegensatz zum modellgetriebenen Testen dient ein Modell im modellzentrischen Ansatz nicht nur zur Erstellung der Testartefakte, sondern bildet den Mittelpunkt dieser Ausprägung und wird auch als Grundlage für weitere Arbeitsschritte, wie z.B. im Testmanagement verwendet. Sämtliche Informationen sollten - soweit sinnvoll - als Modell dargestellt werden und fließen in weitere Arbeitsschritte mit ein. Der Nutzen dieses Ansatzes liegt in reduzierten Abstraktionsschichten und einer transparenten, einheitlichen Darstellungsform für alle Testphasen, die eine einheitliche Arbeitsgrundlage für weitere Schritte bildet [42].

Abbildung 2.2 stellt die Vorgehensweise im modellgetriebenen Testen nach [56] dar. Diese besteht aus fünf Arbeitsschritten, welche im nachfolgenden Absatz näher beschrieben werden.

Den Anfang bildet die Erstellung eines abstrakten Modells des zu testenden Systems. Das Modell sollte sich auf die wesentlichen Aspekte des Systems fokussieren und wichtige Informationen über Anforderungen enthalten. Ist das Modell auf Korrektheit und Konsistenz geprüft, folgt der zweite Arbeitsschritt - die Ableitung abstrakter Testfälle von dem Modell. Als Resultat dieses Schrittes entsteht ein Set an abstrakten Testfällen, die Aktionsfolgen des Modells spezifizieren. Arbeitsschritt drei stellt die konkrete Erstellung von Testskripts dar. Hierzu werden die abstrakten Beschreibungen mittels Tools zu Testfällen transformiert oder mittels eines Adapters direkt als Anweisungen an das System übersetzt. Der Vorteil, die Erstellung der Testfälle in zwei Schichten

auszulagern, liegt laut [56] in der Sprachunabhängigkeit der abstrakten Testfallbeschreibungen. Dies ermöglicht durch den Austausch des Transformationstools oder Adapters eine Wiederverwendung der Testfallbeschreibungen. Die letzten beiden Schritte sind, äquivalent zu jedem anderen Testansatz, das Ausführen der Testfälle und Überprüfen der Resultate.

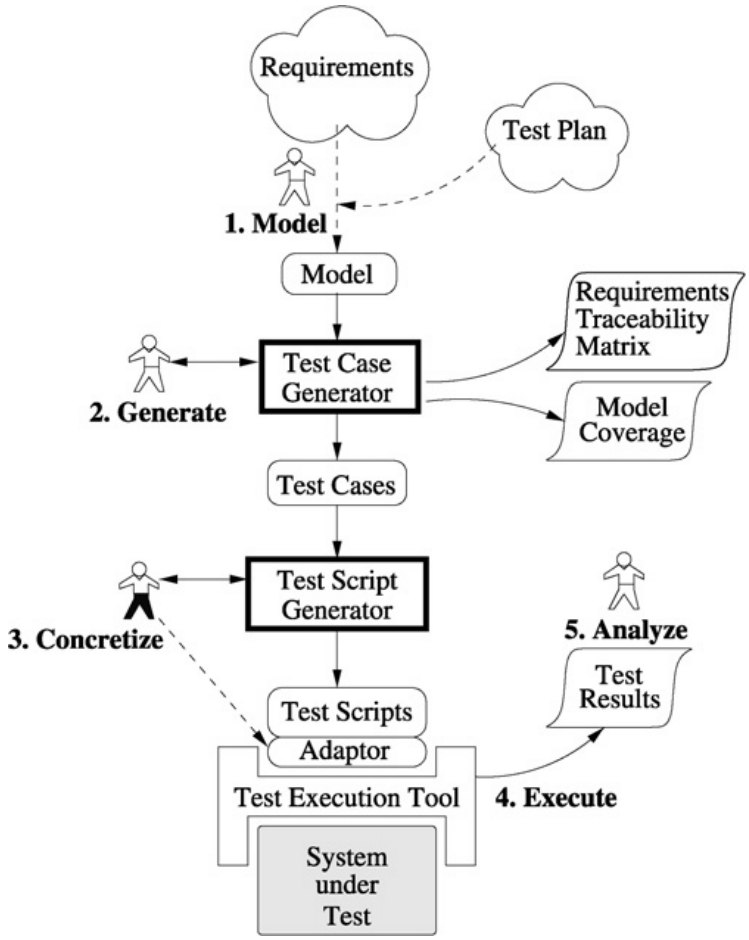



**Abbildung 2.2:** Vorgehensweise im modellbasierten Testen gemäß Utting und Legeard [56].

Im Rahmen dieser Arbeit wird ein Ansatz für modellgetriebenes Testen mit Systemmodellen verwendet. Dabei dienen die Modelle als Eingaben und beinhalten wesentliche Informationen über das zu testende System für die Erzeugung von Testfällen. Eine genauere Darstellung der benutzten Modelle findet sich in Abschnitt 6.1.3.

## 2.7 Frameworks

Testframeworks helfen dem Entwickler bei der Erstellung und Durchführung der Testfälle, sie sind mit den Entwicklungsumgebungen kombinierbar und ein wichtiger Bestandteil des Software-Testens. Diese Arbeit orientiert sich im späteren Verlauf an der Programmiersprache Java. Die bekanntesten Frameworks zur Automatisierung von Komponententests für Java-Programme werden anschließend kurz vorgestellt.

### 2.7.1 JUnit

JUnit[1] [28] ist ein Test-Framework für die Programmiersprache Java. Es soll den Entwickler beim Implementieren der Tests unterstützen und ist nach [22] eines der bekanntesten Frameworks für Java-Entwicklung. Ziel ist es, automatisierte Komponententests zu schreiben, in denen der Entwickler mittels Zusicherungen die *Soll-Ergebnisse* mit dem *Ist-Zustand* der Komponente vergleichen kann.

Baker u. a. erwähnen in [2], dass JUnit vor allem durch seine Einfachheit besticht. Die Testfälle sind verständlich, leicht zu warten und schnell auszuführen. JUnit Testfälle unterscheiden sich hauptsächlich durch die Benutzung von Annotationen von herkömmlichen Java-Methoden. Folgende Annotationen sind gebräuchlich [2, 22]:

**@BeforeClass** - Definiert jene Aktionen, die einmalig vor allen Tetsts auszuführen sind.
**@AfterClass** - Definiert jene Aktionen, die einmalig nach allen Tetsts auszuführen sind.
**@Before** - Definiert Aktionen, die vor jedem Test ausgeführt werden sollen.
**@After** - Definiert Aktionen, die nach jedem Test ausgeführt werden sollen.
**@Test** - Spezifiziert einen konkreten Testfall.

Eine JUnit Testklasse besteht demnach aus einer Vielzahl an Java-Methoden, die mit unterschiedlichen Annotationen gekennzeichnet sind.

Zusätzlich bietet JUnit die Möglichkeit, Testfälle zu Testsuiten zusammenzufassen. Künneth und Wolf beschreiben in [31] die Benutzung von Testsuiten, dabei handelt es sich um ein Set an definierten Testfällen, die sequentiell ausgeführt werden sollen.

### 2.7.2 TestNG

TestNG [6] ist ebenfalls ein Test-Framework für die Programmiersprache Java, das sich besonders für automatisierte Komponententests eignet und laut [22] eine mächtige Alternative zu JUnit darstellt. Obwohl TestNG auf den gleichen Konzepten wie JUnit basiert, beinhaltet es dennoch zusätzliche Funktionen. Es zeichnet sich dabei laut [5] und [22] vor allem durch folgende Merkmale aus:

- Tests können in einem XML-Dokument spezifiziert werden.
- Tests können mittels XML-Spezifikation oder über Annotationen zu Gruppen zusammengefasst werden.
- Es ist möglich, zwischen unterschiedlichen Gruppen Abhängigkeiten und Reihenfolgen zu definieren.
- Nebenläufigkeit wird unterstützt. Tests können in paralellen Threads ausgeführt werden.
- Testmethoden können über parametrisierte Annotationen gesteuert werden.
- Fehlgeschlagene Tests können separat erneut ausgeführt werden, es ist nicht notwendig, die gesamte Testsuite neu auszuführen.
- JUnit-Testfälle können problemlos ausgeführt werden.

[1] Diese Arbeit bezieht sich auf JUnit Version 4.x oder höher.

Annotationen in TestNG sind mit den in JUnit verwendeten vergleichbar. Der Unterschied besteht darin, dass TestNG ein paar zusätzliche Annotationen beinhaltet. Diese sind nach [22] zum Beispiel Anweisungen, die vor und nach dem Testen einer bestimmten Gruppe ausgeführt werden sollen.

Das Quellcodebeispiel 2.1 zeigt einen beispielhaften TestNG Testfall und die Einteilung in Gruppen. Im späteren Verlauf können durch gezielte Gruppierungen Testsuiten flexibler gebildet werden [22]. Hier könnten z.B. alle Testfälle der Gruppe "$functional$", exklusive derer, die zur Gruppe "$database$" gehören, ausgeführt werden. Wie in [22] erwähnt, eignet sich TestNG durch dieses Feature besonders gut für Tests, die viele Abhängigkeiten zueinander besitzen und mehreren Testsuiten zugeordnet werden sollen.

```
1  public class Test1{
2    @Test(groups = {"functional","database"}
3    public void method1(){
4      ...
5    }
6
7    @Test(groups = {"functional","web"}
8    public void method2(){
9      ...
10   }
11 }
```

**Quellcode 2.1:** TestNG Testfall

Eine weiterer erwähnenswerter Punkt nach Beust und Suleiman [5] ist das Erstellen von Reports. TestNG erstellt Testreports im HTML oder XML Format. Die XML-Version kann ohne Probleme in das Ausgabeformat eines JUnit Reports transformiert werden. Zusätzlich bietet TestNG eine Schnittstelle zur individuellen Anpassung der erzeugten Reports.

# 3 Grundlagen: Usability

In diesem Kapitel werden die Grundlagen der Themenbereiche Usability, Usability-Engineering, sowie Design, Design-Richtlinien und die Evaluierung einer Applikation bezüglich guter Usability erläutert. Im weiteren Verlauf wird auf die Relevanz im Zusammenhang mit der Software-Entwicklung eingegangen und der Prozess des benutzerzentrierten Usability-Engineerings näher betrachtet. Abschließend werden allgemeine Design-Richtlinien erläutert und die wichtigsten Evaluierungsmethoden vorgestellt.

## 3.1 Motivation und Relevanz

Die Popularität und der häufige Gebrauch von Software lassen sich nicht einzig und allein durch deren Funktionalität gewährleisten. Wie von der ISTQB in den Grundsätzen des Softwaretestens (siehe 2.2) bereits gezeigt, ist auch die Usability ein wesentlicher Faktor für den Verkauf und letztlich die Benutzung von Software.

Der Begriff Usability lässt sich jedoch nicht einfach in Worte fassen. Eine abstrakte Definition bietet hier die International Organization of Standardization (ISO)[25] :

> *„Usability ist das Ausmaß, in dem ein Produkt durch bestimmte Benutzer in einem bestimmten Nutzungskontext genutzt werden kann, um bestimmte Ziele effektiv, effizient und zufriedenstellend zu erreichen.“*

Nielsen zeigt in [35] bereits eine detaillierte Definition, indem der Begriff *Usability* in fünf verschiedene Attribute aufgeteilt wird. Er unterscheidet zusätzlich zu *Effektivität* und *Zufriedenheit* noch zwischen *Einprägsamkeit*, *Erlernbarkeit* und *Fehlerrate*.
Bei genauerer Betrachtung von Nielsens Attributen lässt sich sagen, dass die *Erlernbarkeit* ebenfalls der Effizienz zugeordnet werden kann. Nielsen versteht hierunter die Komplexität und Dauer, die ein Benutzer benötigt, um mit einem Programm das erste Mal zu interagieren.
Weiters lässt sich die *Einprägsamkeit* mit *Effektivität* gleichsetzen. Die *Einprägsamkeit* bezeichnet gemäß [35], wie einfach es für einen Benutzer auch nach längerer Nichtbenutzung des Programms ist, eine Aufgabe zu erfüllen.
Mit der *Fehlerrate* wird ein zusätzliches Qualitätskriterium erzeugt, das beschreibt wie viele und wie schwere Fehler ein Benutzer während der Interaktion mit dem Programm begeht und wie diese sich auf den weiteren Verlauf auswirken.

Beide Definitionen stimmen also im Wesentlichen überein und verdeutlichen die in [35] genannte grundlegende Fragestellung des Usability-Engineering:

> *„[. . . ] the question of whether the system is good enough to satisfy all the needs and requirements of the users and other potentional stakeholders, such as the users' clients and managers. “*

Usability-Engineering verläuft, wie in [21] gezeigt, parallel unterstützend zum Prozess des Software-Engineerings. Die Gestaltung eines gut benutzbaren und interaktiven Systems in Zusammenarbeit mit dem späteren Benutzer steht im Vordergrund. Gute Usability stellt somit einen Erfolgsfaktor in der Softwareentwicklung dar.

Dadurch ergeben sich Auswirkungen auf Kostenfaktoren, die allerdings nicht unmittelbar sichtbar sind. So kann ein gut umgesetztes Design dem Benutzer Zeit ersparen, da er die Software leichter bedienen kann. Gleichzeitig resultiert dies in gesenkten Supportkosten, weniger Redesigns für spätere Versionen und somit ebenfalls in einer Zeitersparnis des Projektteams. Außerdem steht die Kundenzufriedenheit und damit einhergehend eine Steigerung der Umsätze im Mittelpunkt der Usability [14, 35].

## 3.2 Richtlinien

Die Erstellung von Benutzeroberflächen ist für jedes Software-Projekt ein individueller Prozess mit unterschiedlichen Anforderungen und Bedingungen. Allerdings ist es hier nicht nötig, das Rad von Grund auf neu zu erfinden. Mittels Richtlinien, Guidelines oder Heuristiken werden allgemeine Design-Prinzipien dargestellt, um in späterer Folge Usability-Probleme zu vermeiden und einheitliche Standards zu schaffen.

In diesem Bereich sind besonders die acht goldenen Regeln von Shneiderman und Plaisant [48], und auch die Heuristiken von Nielsen [35] prägend.

### 3.2.1 Shneidermans acht goldene Regeln

Shneiderman [48, 46] beschreibt mit seinen acht goldenen Regeln einfache Hilfestellungen für Designer. Diese Prinzipien müssen allerdings für spezifische Domänen angepasst und überprüft werden. Auch sagt er, dass eine derartige Liste weder komplett erfüllt werden kann noch soll. Diese Regeln dienen als Hilfestellung und grundlegende Ratschläge [47].

1. Konsistenz
   Diese Regel ist mitunter schwierig einzuhalten und wird am häufigsten missachtet. Handlungen und Sequenzen sollten systemübergreifend konsistent sein. Weiters sollten Eingabeaufforderungen, Menüs sowie Farbwahl, Schriftart und Layout konsistent gehalten werden. Allerdings können bestimmte Aktionen, wie zum Beispiel das Bestätigen einer Löschaktion oder das wiederholte Eingeben eines Passworts im Rahmen von anderen Regeln, Ausnahmen für die Konsistenz darstellen.

2. Universelle Benutzbarkeit
   Auf unterschiedliche Benutzergruppen, wie Anfänger und Experten, einzugehen, kann eine Bereicherung des User Interface Designs mit sich bringen. So sollten für unerfahrene Benutzer ausreichende Erklärungen und für fortgeschrittene Benutzer Shortcuts zur schnelleren Bedienung des Programms verfügbar sein.

3. Informatives Feedback
   Der Benutzer soll bei jeder von ihm durchgeführten Aktion eine Antwort erhalten. Er soll bemerken, dass er gerade etwas getan hat. Die Antwort selbst kann je nach Gewichtung und

Häufigkeit der Aktionen unterschiedlich ausfallen. Bei häufigen Aktionen ist also kurzes Feedback ausreichend, bei großen Aktionen wird auch eine äquivalente Antwort benötigt.

4. Abgeschlossenheit
   Unter Berücksichtigung des gegebenen Feedbacks sollte der Benutzer immer wissen, wann seine Handlung abgeschlossen ist. Dies vermittelt ein Gefühl des ordnugsgemäßen Abschlusses einer Aktion und lenkt den Fokus auf nachfolgende Aktionen.

5. Fehler vermeiden
   Das System sollte so gestaltet sein, dass der Benutzer keine Fehler begehen kann. Auswahlmenüs sollten Eingabefeldern vorgezogen werden, unerlaubte Eingabezeichen sollten direkt bei der Eingabe abgefangen werden. Wenn jedoch ein Fehler auftritt, müssen klare und präzise Formulierungen den Benutzer auf diesen hinweisen und einen Lösungsansatz bieten.

6. Umkehrbarkeit
   Sämtliche Aktionen, die ein Benutzer durchführen kann, sollten nach Möglichkeit auch wieder rückgängig gemacht werden können. Dies soll den Benutzer dazu verleiten, ohne Bedenken unbekannte Funktionen zu nutzen und tiefer in das Programm vorzudringen.

7. Benutzerkontrolle
   Erfahrenen Benutzern soll das Gefühl vermittelt werden, dass sie das System aktiv bedienen und nicht darauf reagieren.

8. Kurzzeitgedächtnis entlasten
   Aufgrund der begrenzten Kapazität des menschlichen Kurzzeitgedächtnisses ist eine Reduktion und Simplifikation von Bedienelementen erforderlich. Die gleichzeitige Darstellung mehrerer Seiten oder Fenster sollte vermieden werden. Die Fensterbewegungsfrequenz sollte weitgehendst reduziert werden.

### 3.2.2 Nielsons Heuristiken

Nielsen [35] präsentierte bereits im Jahr 1994 sehr allgemein gehaltene Heuristiken, die auf jede Art von interaktiven Systemen angewandt werden können. Trotz ihres Alters wahren diese Heuristiken bis heute ihre Gültigkeit und werden häufig in Werken erwähnt [21, 36, 43]. Sie sind allerdings lediglich als Hilfestellungen und Ratschläge zu verstehen.

1. Feedback geben
   Der Benutzer muss immer wissen, welche Aktion gerade vom System ausgeführt wird. Geeignete Informationen zum Systemzustand müssen in angemessener Zeit bereitgestellt werden.

2. Die Sprache des Benutzers sprechen
   Das System muss die gleiche Sprache wie der Benutzer sprechen. Durch Verwendung von Metaphern oder anderen Konzepten soll der Benutzer etwas erkennen, das ihm aus der realen Welt vertraut ist. Konventionen aus der Realität müssen eingehalten werden.

3. Benutzerkontrolle
   Der Benutzer soll die direkte Kontrolle über das System haben. Er soll jegliche Aktionen rückgängig machen und wiederholen können.

4. Konsistenz
   Systeme sollen nicht nur in sich geschlossen und konsistent sein, sondern müssen auch Plattformkonventionen einhalten.

5. Fehler vermeiden
   Fehleingaben sollten bereits durch das System verhindert werden. Fehlermeldungen sollen spezifisch sein und den Benutzer zu richtigem Verhalten anleiten.

6. Erkennen statt Erinnern
   Das Gedächtnis des Benutzers sollte nach Möglichkeit entlastet werden. Sämtliche Informationen, die für eine Aktion benötigt werden, sollten auch dargestellt werden.

7. Effizienz und Flexibilität
   Häufig genutzte Funktionen sollen durch Shortcuts ausgeführt werden können, um erfahrenen Benutzern mehr Effizienz zu verschaffen.

8. Ästhetisches und minimalistisches Design
   Es sollen nur relevante Informationen dargestellt werden. Jede zusätzliche Information vermindert die Sichtbarkeit und lenkt den Benutzer ab.

9. Benutzer unterstützen Fehler zu verhindern
   Fehlermeldungen sollen verständlich sein und den Benutzer anleiten, den Fehler zu beheben.

10. Hilfe und Dokumentation
    Das System sollte so gestaltet sein, dass ein Benutzen ohne Dokumentation möglich ist. Sollte ein Benutzer doch die Hilfefunktion in Anspruch nehmen, soll diese so einfach wie möglich gestaltet sein.

Im direkten Vergleich zwischen den Regeln von Shneiderman und den Heuristiken von Nielsen sind deutliche Übereinstimmungen erkennbar. Beide Autoren kommen zu ähnlichen Ergebnissen und stellen somit eine umfangreiche Liste an Ratschlägen für die Gestaltung von Benutzeroberflächen zur Verfügung.

## 3.3 User Centered Design

Allein das Wissen um gutes Design und gute Usability bzw. der Vorsatz, diese zu erstellen, reichen für diesen Prozess nicht aus. Ebenfalls altbewährte Standards, wie Checklisten, Guidelines oder Heuristiken (siehe 3.2), versprechen keine allgemeingültige Lösung, sondern stellen nur allgemein formulierte Regeln für eine solide Grundlage dar, können aber die Bedürfnisse der Benutzer und den expliziten Nutzungskontext nicht in den Designprozess miteinbeziehen.

Ein wichtiger Ansatz des Design Prozesses ist daher das User Centered Design (UCD).
Die Idee bei UCD, wie in [21, 14] zu sehen, besteht darin, den zukünftigen Benutzern einer Software in den Entstehungsprozess miteinzubeziehen. Somit ist es möglich Anforderungen und Nutzungskontext des Benutzers sowie dessen spezifische Sichtweise und Feedback direkt in frühen Phasen des Entwicklungsprozesses zu erfassen und darauf zu reagieren.
UCD konzentriert sich also auf die Bedürfnisse des Benutzers und bietet dadurch eine schnelle und günstige Möglichkeit, Anpassungen an diese umzusetzen.

User Centered Design stellt ein iteratives Modell der Softwareentwicklung dar und unterteilt sich hierbei in folgende zyklisch durchlaufende Projektphasen [21, 43]:

**Phase 1 - Anforderungsanalyse**

Wie in [21, 43] beschrieben, ist das Ziel der ersten Phase des UCD vor allem die Bedürfnisse der Benutzer und den Benutzungskontext zu erfassen. Als erster Schritt werden die funktionalen sowie nicht-funktionalen Anforderungen erhoben. Eine sorgfältige Analyse ist unerlässlich, um gezielt auf die unterschiedlichen Bedürfnisse eingehen zu können. Natürlich ist nicht ausgeschlossen, dass sich manche Anforderungen während des gesamten Prozesses verändern oder neue Anforderungen aufkommen. Trotzdem gilt hier der Grundsatz: Je früher Anforderungen erkannt werden, desto effizienter und kostengünstiger können sie umgesetzt werden. In dieser Phase werden besonders Methodiken wie Interviews, Fragebögen, Szenarios, Gruppendiskussionen, Contextual Inquiry oder Card Sorting angewandt. Nähere Informationen zu den einzelnen Methodiken finden sich in [21, 43].

**Phase 2 - Design**

In der Designphase sollen nach [43, 14] die in der Anforderungsanalyse gewonnenen Erkenntnisse verarbeitet werden. Konzepte bezüglich des späteren Systems werden sowohl in technischer als auch in graphischer Hinsicht erstellt. Nachdem das UCD ein iterativer Prozess ist, ist der erste Entwurf meist sehr grob und wird mit zunehmendem Durchlaufen dieser Phase immer spezifischer und detaillierter. Häufig genutzte Hilfsmittel für die Designphase sind die in 3.2 beschriebenen Richtlinien sowie Entwurfsmuster.

**Phase 3 - Entwicklung**

Die Entwicklungsphase stellt die tatsächliche Umsetzung des Systems dar. Hier ist nicht nur die Funktionalität des Programms, sondern auch die Ästhetik und Handhabung der Benutzeroberfläche wichtig, um die Zufriedenheit des Benutzers zu gewährleisten. Durch das Erstellen von Prototypen lässt sich die Entwicklung in den iterativen Prozess eingliedern und ermöglicht dem Benutzer ein aktives Testen, wodurch neue Wünsche und Bedürfnisse erkannt und abgedeckt werden können [21, 43].
Bei Prototypen unterscheidet man, wie in [21] zu sehen, zwischen mehreren Kategorien:

- Papier Prototypen
  In diesem einfachen und kostengünstigen Prototyp wird die Benutzeroberfläche mittels Zeichnungen und Papier dargestellt. Durch Post-its und kleineren Papierlementen lässt sich eine Funktionalität vortäuschen und ermöglicht dadurch sogar Benutzertests.

- Explorative Prototypen
  Dies bezeichnet das sehr frühe Erstellen eines Prototypen in Projekten mit unklaren oder sehr komplexen Anforderungen. Durch einen Prototypen in einer frühen Projektphase wird eine gemeinsame Diskussionsgrundlage geschaffen. Anforderungen können so leichter spezifiziert werden.

- Experimentelle Prototypen
  Experimentelle Prototypen kommen zum Einsatz, wenn die Umsetzbarkeit nicht sicher scheint.

Sie werden nach der Evaluierung oft verworfen. Mit den bereits gewonnenen Erkenntnissen werden anschließend neue Prototypen gefertigt.

- Evolutionäre Prototypen
  Ein evolutionärer Prototyp entwickelt sich wie das tatsächliche System laufend weiter und wird mit jeder durchlaufenen Phase detaillierter.

**Phase 4 - Evaluierung**

Die Evaluierung stellt nach [21, 43] das zentrale und wesentliche Element im Usability-Engineering. Hier wird überprüft, ob Konzept, Design und Prototyp den Wünschen und Anforderungen des Benutzers entsprechen. Die Evaluierung ist der einzige wirklich zuverlässige Weg die Usability eines Produkts zu überprüfen. Aufgrund der Bedeutung dieser Projektphase ist eine genauere Betrachtung unerlässlich. In 3.4 werden Methodiken der Evaluierung näher vorgestellt.

## 3.4 Evaluierung

Wie zuvor erwähnt ist die Evaluierung die zentrale Phase des Usability-Engineering, deren Zielsetzung darin besteht, Fehler im Design zu erkennen und auszubessern. Die Methoden der Evaluierung sind prinzipiell in zwei Kategorien zu unterteilen [21, 43]:

- Empirische Methoden
  Bei empirischen Methoden wird auf direkten Kontakt mit dem Benutzer gesetzt. In Befragungen oder Beobachtungen mit Testpersonen sollen Rückschlüsse auf Fehler gezogen werden.
- Inspektionsmethoden
  Bei Inspektionsmethoden versuchen Usability-Experten sich in die Lage des Benutzers zu versetzen und anhand ihrer Erfahrung Probleme vorherzusagen. Aufgrund des Fehlens von tatsächlichen Benutzern sind die Ergebnisse von Inspektionsmethoden allerdings nur bedingt weiterverwertbar.

Zusätzlich sind Evaluierungsmethoden noch anhand des Zeitpunktes, an dem sie eingesetzt werden, zu unterscheiden [21, 43]:

- Formative Evaluierung
  Die formative Evaluierung kommt in den frühen Projektphasen zum Einsatz und wirkt entwicklungsbegleitend. Die Zielsetzung ist es, Usability und Design Probleme möglichst früh zu erkennen und konkrete Verbesserungsmöglichkeiten aufzuzeigen. Es findet also eine kontinuierliche Verbesserung des Systems während der gesamten Entwicklung statt.
- Summative Evaluierung
  Die summative Evaluierung findet gegen Ende des Projekts statt. Ziel ist es, die Qualität des fertigen Produkts zu ermitteln, wobei eine eher globale Bewertung gegeben wird und zumeist keine konkreten Verbesserungsvorschläge entstehen. Änderungen am System sind an diesem Punkt mit hohen Kosten und viel Aufwand verbunden.

Nur ein gutes Zusammenspiel zwischen empirischen Methoden und Inspektionsmethoden sowohl mit formativer als auch summativer Evaluierung führt zu einem ausgeglichenen und gut benutzbarem Produkt. Nachfolgend werden die wichtigsten Evaluierungsmethoden näher betrachtet.

### 3.4.1 Heuristische Evaluierung

Die heuristische Evaluierung, manchmal auch Experten Evaluierung genannt, zählt zu den Inspektionsmethoden. Diese Methode benutzt Heuristiken und Richtlinen (siehe 3.2), die von Experten am bestehenden Produkt angewandt und überprüft werden.

Wie in [21, 35] beschrieben, wird eine geringe Anzahl an Usability-Experten zum Projekt hinzugezogen und mit der Evaluierung beauftragt. Kosten-Nutzen Analysen (siehe [21]) haben ergeben, dass eine Anzahl von drei bis fünf Experten ausreichend ist, um eine Vielzahl an Problemen aufzudecken. Die Experten versuchen dabei sich in die Sichtweise des Benutzers zu versetzen und dessen Handlungen nachzuahmen.

Ziel ist es, Verstöße und Abweichungen von den beschriebenen Heuristiken zu finden und zu analysieren, wobei ein Verstoß gegen eine Heuristik nicht gleichbedeutend mit einem Usability-Problem ist. Hier ist der entsprechende Nutzungskontext entscheidend. Der Vorteil dieser Methode liegt in der schnellen Durchführbarkeit, die oft nicht mehr als einen Tag benötigt. Gleichzeitig gilt es zu beachten, dass die Resultate nur bedingt verwertbar sind, da keine realen Benutzer an der Evaluierung beteiligt sind [35, 43].

### 3.4.2 Usability Tests

Bereits Nielsen erkannte in [35], dass Usability-Tests die fundamentalste Methode darstellen, um die Usability eines Systems zu bewerten.

Bei dieser empirischen Methode werden unterschiedliche Testpersonen als Benutzer des Systems herangezogen und sollen vordefinierte Aufgaben lösen. Dies liefert sowohl qualitative als auch quantitative Ergebnisse, die einer näheren Analyse bedürfen. Trotzdem sind die hieraus gewonnenen Resultate die aussagekräftigsten, da sie direkte Informationen darüber bieten, wie reale Benutzer im Umgang mit dem System arbeiten und welche Probleme dabei entstehen [21, 35].

Die Auswahl der Testpersonen ist abhängig vom Ziel des Tests. So werden unerfahrene Benutzer vor allem beim Testen der Erlernbarkeit eines Systems herangezogen, während erfahrene Benutzer bei Tests um spezifischere Funktionen und Aufgaben eingesetzt werden. Während des Tests werden die Testpersonen von Experten beobachtet und ihr Verhalten aufgezeichnet und dokumentiert [21, 43]. Zusätzliche Techniken, wie laut gesprochene Gedanken der Testperson, Videos, Eye-Tracking, Screen-Records, Messungen der Klicks oder der Mausbewegungen bieten einen tieferen Einblick in die Sichtweise des Benutzers und helfen beim Identifizieren der Probleme. Die gewonnen Informationen allein bieten natürlich noch keine Aussagekraft. Sie müssen von den Experten strukturiert und analysiert werden [21].

Ein weiterer Faktor, der unbedingt zu berücksichtigen ist, wird in [35] beschrieben. Nielsen macht darauf aufmerksam, dass die Testpersonen ebenfalls ein Kriterium für die Auswertung sind. Verschiedene Testpersonen liefern individuelle Ergebnisse und können nicht untereinander verglichen werden. Nur weil Testperson A schneller war als Testperson B, bedeutet dies nicht zwingend, dass die von Person A gewählte Lösung die schnellste oder einfachste ist. B ist womöglich allgemein langsamer oder ungeübter als A.

Unter Beachtung dieser Kriterien ist es somit möglich, die häufigsten und gravierendsten Probleme zu ermitteln und zu korrigieren. Zusammenfassend lässt sich über das User-Testing sagen, dass trotz hoher Kosten und Zeitintensivität der Einsatz von realen Testpersonen und Benutzern die besten und aussagekräftigsten Resultate liefert und auf diese Methode keinesfalls verzichtet werden sollte.

# 4 Grundlagen: Eclipse Rich Client Plattform

Das nachfolgende Kapitel gibt eine Einführung in das Thema Eclipse und bietet einen Überblick über die Eclipse Rich Client Platform (RCP), die für den weiteren Verlauf dieser Arbeit relevant ist. Anschließend werden einige Vorteile und der Verwendungszweck der RCP erläutert sowie einige Beispielapplikationen genannt. Weiters werden die grundlegenden Komponenten einer RCP-Applikation vorgestellt und kurz beschrieben. Abschließend wird näher auf das Graphical Editing Framework (GEF) eingegangen und dessen Struktur erläutert.

## 4.1 Definition und Relevanz

McAffer, Lemieux und Aniszczyk geben in [32] die folgende Definition zum Begriff Eclipse:

> *„Eclipse is an open-source community of people building Java-based tools and infrastructure to help you solve your problems“*

Unter dem Begriff Eclipse versteht man also eine Gemeinschaft, deren Arbeit die Entwicklung Java basierter Software erleichtern und bereichern soll. Das bekannteste Produkt stellt hierbei die gleichnamige Entwicklungsumgebung dar. Abseits der Entwicklungsumgebung bieten die Eclipse-Produkte eine Vielzahl von Erweiterungsmöglichkeiten und kommen deshalb auch in anderen Bereichen zum Einsatz.

Die Eclipse Rich Client Platform (RCP) stellt nach [15] eine generische Plattform dar, die es zum einen ermöglicht, Plug-ins als Erweiterung für bereits bestehende RCP-Applikationen zu entwickeln und zum anderen die Erstellung von eigenständigen Rich-Client-Applikationen erlaubt. Die Plattform definiert dabei das Grundgerüst und kann um eigene Funktionalitäten erweitert werden. Wie in [30] beschrieben, soll es dadurch möglich werden, Applikationen aus verschiedenen Blöcken zusammenzusetzen und diese auf verschiedene Arten zu kombinieren, wiederzuverwenden oder zu verteilen.

Sowohl Ebert [15] als auch Silva [49] erläutern einige Vorteile und Stärken der Plattform. So bietet RCP etablierte Basiskomponenten für graphische Benutzeroberflächen, die ein natives Look&Feel vermitteln. Weiters sind die Applikationen plattformunabhängig und für ihre grundsätzliche Stabilität und Konsistenz bekannt. Nicht zuletzt zeichnet sich die Eclipse Gemeinschaft durch ihre Aktivität und Hilfsbereitschaft aus, die Produkte erfahren eine weite Verbreitung und Nutzung.

Markante Beispiele für die Nutzung von Eclipse RCP Projekten sind, wie in [32] erwähnt, neben der Entwicklungsumgebung selbst IBMs Lotus Notes oder das Maestro Project der NASA.

## 4.2 Komponenten

Silva zeigt in [49], dass RCP ein leichtgewichtiges Software-Komponenten-Framework basierend auf Plug-ins ist. Dieses Architekturkonzept ermöglicht eine einfache Erweiterbarkeit und nahtlose Integration in das Betriebssystem. Das bestehende Framework kann durch Bereitstellung von Plug-ins um eigene Funktionalität erweitert werden. Eine RCP-Applikation besteht somit, abgesehen von der Core-Plattform, ausschließlich aus Plug-ins. Sämtliche hinzugefügte Funktionen sind dabei gleichwertig, da alle hinzugefügten Plug-ins exakt gleich mit Eclipse interagieren.

Abbildung 4.1 zeigt die in [15] dargestellten Basiskomponenten und die grundsätzliche Zusammensetzung einer Eclipse-RCP-Applikation.

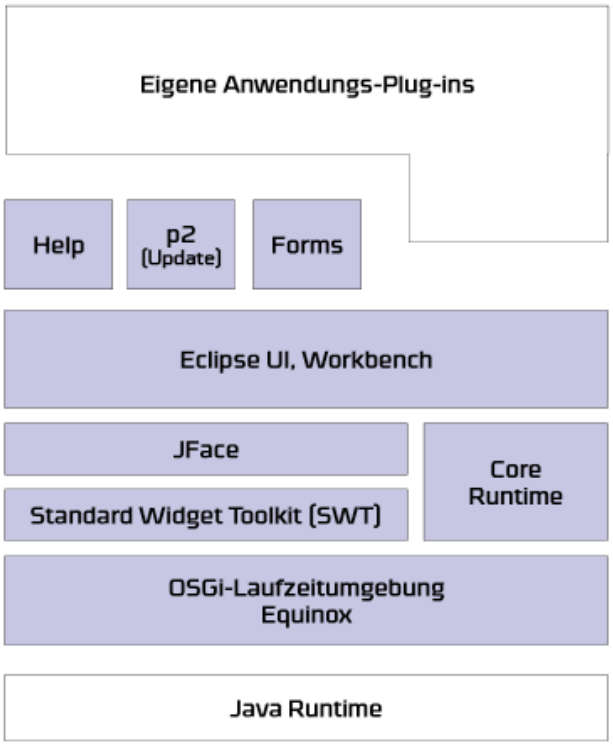



**Abbildung 4.1:** Bestandteile einer RCP Applikation nach [15]

- Equinox
  Der Begriff Open Services Gateway initiative (OSGi) bezeichnet nach [49] eine dynamische Softwareplattform für Java, deren Fokus auf der Entwicklung neuer und der Integration bestehender Software in neue Systeme liegt. Die Verwendung von OSGi steigert die Zuverlässigkeit und Wiederverwendbarkeit eines Sofwaresystems und ermöglicht es gleichzeitig, die Entwicklungskosten zu reduzieren.

  OSGi zeichnet sich dabei vor allem durch ein Komponentensystem aus. Wie Silva [49] beschreibt, werden Applikationen aus kleinen, wiederverwendbaren und zusammenarbeitenden Komponenten erstellt. Eine Komponente, auch Bundle genannt, kann mittels des OSGi-Frameworks installiert, gestartet, gestoppt, aktualisiert und deinstalliert werden, ohne einen Neustart des Systems vorauszusetzen.

Der Begriff „Equinox“ bezeichnet nach [49, 15] eine Eclipse-spezifische Variante des OSGi Frameworks, die sowohl das Lebenszyklusmodell der Applikationen definiert als auch eine Service Registry und ein Execution Environment definiert.

- Eclipse Core Plattform
  Die Core Plattform (oder Core Runtime) von Eclipse stellt allgemeine Funktionalitäten bereit. Die Hauptaufgabe besteht, wie in [49] erörtert, darin, den Lebenszyklus, also Start und Initialisierung der Anwendung, zu verwalten und dynamisch nach neuen Plug-ins zu suchen. Außerdem übernimmt die Core Plattform die Aufgabe der Wartung von Plug-ins, Festlegung der Erweiterungen und Erweiterungspunkte sowie die Bereitstellung einiger zusätzlicher Funktionen, wie zum Beispiel Logging oder Debugging.

  Die Eclipse Core Plattform stellt also den Kern einer RCP Applikation dar und hält alle Teile effektiv zusammen [49].

- Standard Widget Toolkit
  Das Standard Widget Toolkit (SWT) ist laut [32] eine Programmbibliothek für graphische Benutzeroberflächen, die einige standardisierte Kontrollelemente für die Erstellung von Benutzeroberflächen beinhaltet. Wie in [49] dargestellt, bietet SWT ein natives Look&Feel und ist zusätzlich durch eigene Komponenten erweiterbar.

- JFace
  Wie in [49] beschrieben, ist JFace ein UI Toolkit, das dazu verwendet wird, viele, häufig verwendete Programmkonstrukte zur Verfügung zu stellen. Das Toolkit ist darauf ausgelegt, mit SWT zusammenzuarbeiten, bietet zusätzliche Funktionalitäten und implementiert dabei eine Model-View-Controller (MVC) Architektur. Es findet also eine Unterteilung zwischen zugrundeliegendem Datenmodell (Model), Darstellung am Bildschirm (View) und der allgemeinen Logik (Controller) statt. Die Strukturen von JFace, besonders Actions und Viewer, bilden nach [32] die Basis der Eclipse UI.

  Viewer werden benutzt, um die Interaktion zwischen dem Datenmodell und den zugehörigen Elementen des Views zu ermöglichen. Actions geben eine Rückmeldung darüber, was der Benutzer gerade macht, welcher Button gedrückt wurde oder welche definierte Tastenkombination betätigt wurde [49].

- Workbench
  Die Workbench stellt nach [15] das Programmfenster der späteren Applikation dar und unterstützt sämtliche bekannten Bedienelemente, die mittels Plug-Ins eingebunden werden können. Plug-Ins erweitern die Workbench also um ihre Funktionalität.

## 4.3 Graphical Modeling Framework

Das Graphical Modeling Framework (GMF) dient dazu Editoren und graphische Darstellungen von Modellen zu entwickeln. GMF setzt sich hierbei aus dem Graphical Editing Framework (GEF) und dem Eclipse Modeling Framework (EMF) zusammen, wobei jeder Teil auch separat Anwendung finden kann [33].

Im Rahmen dieser Arbeit soll die RCP Applikation mittels GEF um graphische Elemente erweitert werden, die auch dem ursprünglichen Einsatzzweck von GEF entsprechen: der Darstellung von UML-Diagrammen.

GEF erweitert die Eclipse Workbench um die Möglichkeit graphische Bereiche in Editoren und Sichten zu integrieren und mit diesen zu arbeiten. Das Framework soll vor allem dabei helfen, eine Verbindung zwischen Daten und deren graphischen Repräsentationen darzustellen. Ziel ist es, Veränderungen in der graphischen Darstellung ebenfalls als Veränderung am zugrundeliegenden Datenmodell zu realisieren [33].

GEF besteht hierzu grundsätzlich aus drei Komponenten [10]:

- Draw2d - einem leichtgewichtigem Toolkit basierend auf SWT. Draw2d wird zur Darstellung von Graphiken in einem SWT Canvas Element verwendet.
- GEF (MVC) - ein auf dem Model-View-Controller (MVC)-Pattern aufbauendes Framework, das eine Vernetzung zwischen Datenelementen und deren graphischen Repräsentationen ermöglicht.
- Zest - einem Visualisierungstoolkit basierend auf Draw2d zur Darstellung von gerichteten Graphen.

GEF benutzt das Model-View-Controller (MVC)-Pattern. Dieses Entwurfsmuster wird verwendet, um die Verantwortlichkeiten beim Aufbau einer Benutzerschnittstelle auf drei verschiedene Rollen zu verteilen und vereinfacht somit die Darstellung der Informationen des Models. [17]. Der View ist hierbei die Darstellung in der Benutzerschnittstelle, unter Model versteht man die entsprechenden Datenelemente des Programms. Im Zusammenhang mit GEF stellen spezielle Elemente, die EditParts, den Controller dar, dieser bildet die Daten eines Modells als Figur im View ab [16]. Abbildung 4.2 zeigt die Bestandteile des MVC-Patterns und deren Datenfluss.

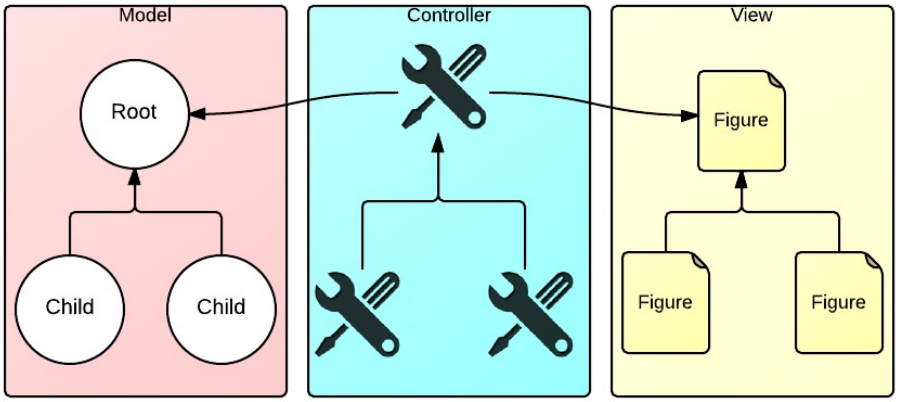



**Abbildung 4.2:** MVC Benutzung bei GEF
nach [16]

# 5 Motivation und Umfeldbeschreibung

Dieses Kapitel beschreibt nach den zuvor erläuterten Grundlagen der Bereiche Softwaretesten, Usability und dem Eclipse RCP-Framework das Umfeld der im Rahmen dieser Arbeit erstellten Applikation. Es folgen eine Projektbeschreibung, eine detaillierte Auseinandersetzung mit der Problematik hinter dem Begriff Softwarevolatilität sowie eine Betrachtung der Relevanz, Motivation und dem Ziel des Projekts. Eine Beschreibung der funktionalen und nicht-funktonalen Anforderung schließt dieses Kapitel ab.

## 5.1 Projektbeschreibung

Im Rahmen dieser Arbeit wird ein Tool zum Testen von Benutzeroberflächen entwickelt, das einen modellbasierten Testansatz mit einer intuitiven Arbeitsweise über eine graphische Benutzeroberflächen kombiniert. Ein generischer Ansatz ermöglicht, dass keine Einschränkungen hinsichtlich bestimmter Applikationstypen oder Abhängigkeiten auf Programmiersprachen bestehen und somit eine Vielzahl an verschiedenen Applikationen mit einer gleichbleibenden Vorgehensweise getestet werden kann. Gleichzeitig wird durch den generischen Ansatz der bei bestehenden Testframeworks vorliegenden Problematik der Technologievolatilität entgegengewirkt. Ziel ist es, ein Tool zu schaffen, das in den Softwareentwicklungsprozess integrierbar ist und die Arbeitsaufgaben verschiedener Rollen (z.B. Entwickler, Tester) unterstützt und erleichtert.

Das Tool soll die Spezifikation von Testfällen vereinfachen und auch Testbeauftragten ohne Programmierkenntnisse die Erstellung von Testfällen ermöglichen. Es verfolgt dabei einen modellbasierten Ansatz, d.h. die zu testende Benutzeroberfläche wird als Modell aufbereitet. Eine genauere Beschreibung und Erläuterung eines solchen Modells folgt in Abschnitt 6.1.3. Die Erstellung der Modelle wird von einem externen Tool vorgenommen, die Modelle werden von der Applikation ausschließlich als Eingabedateien benutzt und werden für den weiteren Workflow verarbeitet. Anschließend wird das Modell für den Benutzer aufbereitet, sodass dieser über eine einfach gestaltete und leicht zu bedienende Benutzeroberfläche konkrete Abläufe für das zu testende System definieren kann. Abschließend können die spezifizierten Abläufe als ausführbare Testfälle für wahlweise JUnit oder TestNG exportiert werden. Abbildung 5.1 zeigt den Informationsfluss der Applikation und verdeutlicht das Zusammenspiel der unterschiedlichen benötigen Komponenten.

Die Umsetzung des Projekts erfolgt als Eclipse Rich Client Platform (RCP)-Applikation mit graphischer Benutzeroberfläche. Vorerst beschränken sich die im Rahmen dieser Arbeit eingelesenen Modelle allerdings auf die Repräsentationen von Webseiten.

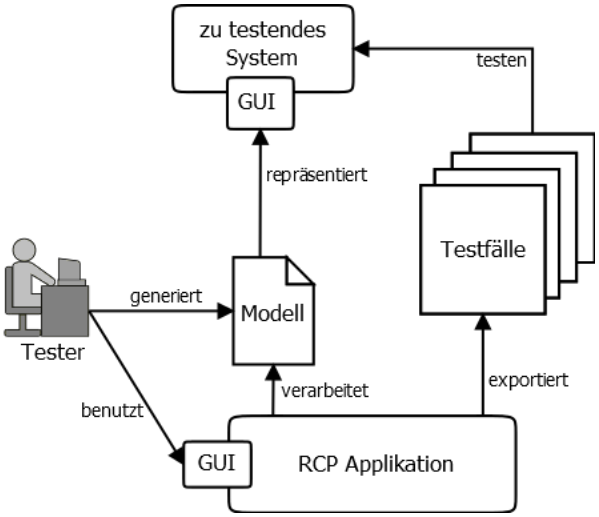



**Abbildung 5.1:** Informationsfluss und Komponenten

## 5.2 Software-Volatilität

Die ursprüngliche Bedeutung des Begriffs Volatilität bezeichnet gemäß [12] eigentlich finanzmärktliche Schwankungen innerhalb einer kurzen Zeitspanne. In der Software-Entwicklung hat sich dieser Begriff jedoch ebenfalls manifestiert. So definieren Zhang, Windsor und Pavur in [57] Software-Volatilität wie folgt:

> *„Software volatility is defined as the propensity for software to change over time in response to evolving requirements. “*

Barry und Slaughter kommen in [3] zu einer ähnlichen Definition:

> *„Software volatility, a characteristic of software behavior, describes the changeable nature of software. “*

Software Volatilität bezeichnet demnach das über einen bestimmten Zeitraum gemessene Maß an Veränderungen an einem Softwaresystem und steht somit in direkten Zusammenhang mit der Erweiterbarkeit und Wartbarkeit eines Systems.

In [57] wird eine Unterteilung in verschiedene Formen der Software-Wartbarkeit vorgenommen, diese werden unterteilt in: korrektive, adaptive und perfektionierende Wartung. Korrektive Wartung bezeichnet dabei das Beheben von Fehlern, unter adaptiver Wartung sind Anpassungen und Adaptionen zur Erfüllung von neuen Anforderungen gemeint, mit perfektionierender Wartung soll eine Steigerung der Software-Qualität erzielt werden.

Diese Arbeit wird im späteren Verlauf vor allem auf Auswirkungen der adaptiven Wartung Bezug nehmen. Wie bereits in den Abschnitten 2.4.3 und 2.6 erwähnt, besteht das Risiko, dass anschließend an Veränderungen und Adaptionen des Systems bereits bestehende Testfälle nicht mehr richtig arbeiten und daher angepasst oder komplett neu erstellt werden müssen. Dieser Prozess ist jedoch zeitintensiv und äußerst aufwändig. So muss sich der Tester nicht nur in bestehende Teile des Systems erneut einarbeiten und abermals Überblick über die, nun veränderte, interne Struktur verschaffen, er benötigt auch fundierte Kenntnisse über das System, die durchgeführten Veränderungen und die dabei eingesetzten Technologien. Weiters ist es wichtig, nach durchgeführten

Adaptionen auf Nebenwirkungen in anderen Systemteilen zu testen. Testautomatisierung (siehe Abschnitt2.5) ist auch hier ein entscheidendes Hilfsmittel.

## 5.3 Relevanz und Motivation des Projekts

Durch die steigende Verwendung von Testautomatisierung gelingt es zunehmend, durch Veränderung verursachte Nebenwirkungen schnell und effizient zu finden sowie die für die Testdurchführung benötigte Zeit zu senken und dadurch effizientere und qualitativ hochwertige Softwareprojekte zu erschaffen.

Bereits Dustin, Rashka und Paul zeigten in ihrer Studie in [13] einen Aufwandsvergleich zwischen manuellem und automatisiertem Testen von Benutzeroberflächen. Tabelle 5.1 stellt die Resultate dieser Studie dar und verdeutlicht somit, dass der Aufwand pro durchgeführtem Testfall wesentlich geringer ausfällt, obwohl Testautomatisierung einen erhöhten Initialaufwand besitzt.

| | Vorbereitung (in Stunden) | | | Durchführung (in Stunden) | | |
|---|---|---|---|---|---|---|
| | Mittelwert | Min. | Max. | Mittelwert | Min. | Max. |
| Manuell | 11,6 | 10,0 | 20,0 | 3,93 | 0,5 | 24,0 |
| Automatisiert | 19,2 | 10,6 | 56,0 | 0,21 | 0,1 | 1.0 |

**Tabelle 5.1:** Vergleich des Testaufwands zwischen manuellen und automatisierten GUI Tests nach einer Studie in [13].

Gerade aufgrund des steigenden Umfangs und der Komplexität eines Projekts sowie stetig wachsenden Anforderungen, ist eine Reduktion des Initialaufwands erstrebenswert. Bei Tests von Benutzerschnittstellen ergibt sich dabei eine besondere Problematik, wie Börjesson und Feldt in [7] bemerken. Aus umfangreichen und komplexen Benutzerschnittstellen mit vielen verschiedenen Ansichten ergeben sich eine Vielzahl von möglichen Kombinationen, die getestet werden sollten. Obwohl das Testen bei derartigen Systemen essentieller Bestandteil der Entwicklungsphase ist, kann es dennoch eine nahezu unüberwindbare Hürde darstellen. In [37] werden aufgrund dieser Problematik drei wesentliche Anforderungen an Tests spezifiziert:

- Tests sollen alle Funktionen des Systems umfassen.
- Tests sollen alle Aspekte der Benutzerschnittstelle umfassen.
- Tests sollen wartbar und wiederverwendbar sein.

Zusätzlich sollen Testfälle gemäß [37] möglichst schnell und einfach erstellt werden, Dokumentation bezüglich ihres Umfangs enthalten und Qualität und Gründlichkeit garantieren. Ein Ansatz um diese Anforderungen im Zuge von Benutzerschnittstellen-Tests zu erfüllen sind die in Abschnitt 2.6.2 beschriebenen Capture/Replay-Verfahren, welche auch bei der Studie von [13] eingesetzt wurden. Diese ermöglichen durch ihre simple Handhabung eine wenig komplexe Erstellung von ausführbaren Tests und erleichtern es so, große Sequenzen an aufgenommenen Testskripts erneut automatisiert auszuführen [37]. Die Erstellung der Testfälle setzt jedoch einen manuellen Durchlauf für die Aufnahme voraus.

Der Schwachpunkt von Capture/Replay-Verfahren wird spätestens bei Änderungen der Benutzerschnittstelle offengelegt. Zwar ist eine simple Erstellung von Testfällen gewährleistet, die aufgenommenen Testskripts sind jedoch schwierig zu bearbeiten und gegenüber Änderungen an der

Benutzeroberfläche äußerst fragil [7]. Die von Ostrand u. a. in [37] genannten Anforderungen sind somit nur teilweise erfüllt.

Veränderungen der Benutzeroberfläche resultieren darin, dass die Funktionalität bereits bestehender Testskripts nicht mehr gegeben ist und diese neu erstellt werden müssen. Trotz der Einfachheit der Erstellung von Testfällen mittels Capture/Replay-Tools sind diese als manuelle Tätigkeit zeit- und kostenintensiv, woraus sich die Notwendigkeit nach einer robusteren Lösung ergibt [7], welche im Rahmen dieser Arbeit vorgestellt und umgesetzt werden soll. Die hier vorgestellte Applikation soll die folgenden Vorteile aufweisen.

- Das Testen der zukünftigen Benutzerperspektive kann früh und kontinuierlich in den Testprozess aufgenommen werden.
- Testfälle werden direkt aus der Sicht der Benutzerperspektive geschrieben.
- Für das Schreiben von Testfällen sind weder Expertenwissen noch Programmierkenntnisse erforderlich.
- Zusammenarbeit zwischen Benutzer und Tester wird unterstützt.
- Fehler können früh gefunden und beseitigt werden.
- Testfälle werden in gängige Formate exportiert.
- Testfälle sind leicht zu adaptieren und an Änderungen in der Benutzeroberfläche anzupassen.
- Testfälle können direkt in bestehendes Testmanagement integriert werden.
- Gut nutzbar bei agilen Entwicklungsmodellen.
- Unterschiedliche Applikationstypen können mit derselben Vorgehensweise getestet werden.

## 5.4 Anforderungsdokumentation

Diese Arbeit beschäftigt sich mit einem Ansatz zur Erstellung von Tests der Benutzerschnittstelle, der die Vorteile von Capture/Replay-Tools innehat, jedoch gleichzeitig die Schwachpunkte ebendieser Tools vermeidet. Tabelle 5.2 zeigt die grundlegenden funktionalen Anforderungen im Überblick, während Tabelle 5.3 die nicht-funktionalen Anforderungen aufzeigt.

| | | |
|---|---|---|
| F1 | Einlesen und Verarbeiten von Modellen | Die Applikation muss Modellrepräsentationen der GUI des zu testenden Systems entgegennehmen und für die weitere Verwendung verarbeiten können. Dem Benutzer muss eine Darstellung der Methoden des Modells angezeigt werden. Eine nähe Beschreibung der Modelle befindet sich in Abschnitt 6.1.3 |

| | | |
|---|---|---|
| F2 | Erstellen von Testfällen | Dem Benutzer muss es möglich sein Testfälle zu erstellen. Ein Testfall umfasst dabei eine oder mehrere Instruktionen, die den Methoden eines Modells entsprechen. Mehrere Testfälle sollen weiters in einer Testdatei gekapselt werden. Zum Erstellen der Testfälle soll sowohl die Listendarstellung, als auch die Diagrammdarstellung genutzt werden können. |
| F2.1 | Drag&Drop Unterstützung | Das Hinzufügen der Instruktionen zu einem Testfall soll mittels Drag&Drop Aktionen erfolgen. |
| F2.2 | Darstellung in Listenform | Erstellte Testfälle sollen dem Benutzer in Form einer Liste dargestellt werden. Die Liste soll einen Überblick über die Reihenfolge und Art der Instruktion bieten. |
| F2.3 | Darstellung als Diagramm | Erstellte Testfälle sollen dem Benutzer in Form eines UML-Aktivitätsdiagramms dargestellt werden. |
| F3 | Bearbeiten von Testfällen | Der Benutzer soll zu einem Testfall beliebig viele Instruktionen hinzufügen sowie diese nach Belieben anordnen und löschen können. Zusätzlich sollen weitere Eigenschaften bezüglich benötigter Parameter oder für den Testfall relevanten Annotationen gesetzt werden können. |
| F4 | Export der Testfälle | Die erstellten Testfälle sollen als eigenständige und korrekte Datei im entsprechenden Format generiert und exportiert werden können. |
| F4.1 | Unterstützung des JUnit Frameworks | Das System soll das Testframework JUnit-4 unterstützen. Testfälle sollen in diesem Format erstellt und exportiert werden können. |
| F4.2 | Unterstützung des TestNG Frameworks | Das System soll das Testframework TestNG unterstützen. Testfälle sollen in diesem Format erstellt und exportiert werden können. |
| F5 | Automatisches Einlesen von Modellen | Das System soll Modelle aus einem spezifiziertem Ordner automatisch beim Start einlesen und verarbeiten. |
| F6 | Hilfefunktion | Dem Benutzer soll über eine Hilfefunktion ein Benutzerhandbuch zugänglich sein |
| F7 | Konfigurationsdatei | Das System soll grundlegende Einstellungen über eine Konfigurationsdatei bereitstellten. |

| | | |
|---|---|---|
| F8 | Mehrsprachigkeit | Die Applikation soll auf die Verwendung von Internationalisierung ausgelegt sein und mehrere Sprachen, bzw. die Erweiterung auf andere Sprachen unterstützten. |

**Tabelle 5.2:** funktionale Anforderungen

| | | |
|---|---|---|
| NF1 | Plattformunabhängigkeit | Die Applikation soll keine Einschränkungen bezüglich der gewählten Plattform und Ausführungsumgebung aufweisen. |
| NF2 | Benutzbarkeit | Ein mit Eclipse erfahrener Benutzer soll intuitiv mit der Applikation interagieren können und keine bzw. nur eine sehr geringe Einarbeitungszeit haben. |
| NF3 | Wartbarkeit | Im Zuge der Implementierung soll auf Modularität sowie die Verwendung von Entwurfsmustern und einer umfangreichen Dokumentation geachtet werden, was die zukünftige Wartung und Adaptierung der Applikation positiv beeinflussen soll. |
| NF4 | Erweiterbarkeit | Die Implementierung soll durch die Bereitstellung von Interfaces leicht um neue Bestandteile und zusätzliche Funktionen erweitert werden können. |
| NF5 | Modularität | Die einzelnen Bestandteile des Systems sollen in klar definierte Module getrennt sein und untereinander eine lose Kopplung aufweisen, wodurch die Austauschbarkeit der Module erhöht werden soll. |
| NF6 | Korrektheit | Das System soll korrekt, zuverlässig und fehlerfrei arbeiten. Generierte Testfälle sollen syntaktisch korrekt exportiert werden können und keine Fehler aufweisen. |
| NF7 | Look&Feel | Das Aussehen der Applikation soll das Look&Feel des entsprechenden Betriebssystems benutzen. Zusätzlich soll das Aussehen durch die Benutzung von Icons und Graphiken individuell angepasst werden können. Der Aufbau der Applikation soll sich an der Struktur der Eclipse-IDE orientieren. |

**Tabelle 5.3:** nicht-funktionale Anforderungen

# 6 Architektur und Design

Dieses Kapitel behandelt die Planung und den Entwurf der in Kapitel 5 beschriebenen Applikation. Im Nachfolgenden wird näher auf die Notwendigkeit einer gut geplanten Software-Architektur eingegangen sowie der grundlegende Aufbau einer RCP-Applikation erläutert. Im Anschluss wird ein Konzept für die zugrundeliegende Programm-Architektur vorgestellt und deren Vorzüge diskutiert. Weiters wird die Struktur und der Aufbau der Modelle, die als Schnittstelle zwischen der Applikation und den zu testenden Systemen agieren, erläutert. Abschließend folgt eine skizzenhafte Beschreibung, Darstellung und Evaluierung der geplanten Benutzeroberfläche.

## 6.1 Architektur

Die Architektur eines Software-Projekts stellt dessen fundamentalen Aufbau und dessen interne Struktur dar. Bass, Clements und Kazman [4] definieren den Begriff der Software-Architektur wie folgt:

> *The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.*

Hinzuzufügen ist, dass auch die Architektur eines Software-Projekts dem geplanten Einsatzzweck sowie den Anforderungen und dem Typ des Projekts angepasst werden muss. Abhängig von den gesetzten Prioritäten ist es somit möglich, für dasselbe Projekt unterschiedlich aufgebaute Architekturentwürfe anzufertigen, die jedoch dieselben Anforderungen gleichermaßen erfüllen. Bass, Clements und Kazman definieren in [4] einige Regeln und Anforderungen, deren Einhaltung eine gut entworfene Architektur gewährleisten sollen:

- Die Architektur soll die Struktur des Systems darstellen, aber konkrete Implementierungen verbergen.
- Die Architektur soll aus wohldefinierten Modulen bestehen und dadurch die Prinzipien der Datenkapselung und des Separation-of-Concerns erfüllen.
- Jedes Modul soll Schnittstellen bereitstellen, welche konkrete Implementierungsaspekte nach Außen hin verstecken. Somit können unterschiedliche Bereiche der Software voneinander losgelöst entwickelt und verwendet werden.
- Die Architektur soll unabhängig gegenüber bestimmten Versionen von Produkten, Tools und Frameworks sein. Das Austauschen derselben soll jederzeit und einfach durchführbar sein.
- Module, die Daten bereitstellen, sollen von denen die Daten konsumieren getrennt sein.
- Qualitätskriterien sollen durch die Verwendung von Entwurfsmustern erreicht werden.

Eine gut definierte Architektur ist ausschlaggebend für den erfolgreichen Verlauf eines Software-Projekts. Die Architektur soll sowohl Aspekte der funktionalen als auch nicht-funktionalen Anforderungen abbilden und somit eine stabile Grundlage für das Projekt darstellen. Passt die Architektur nicht zum umgesetzten Projekt, kann dies von einer instabilen Umgebung bis hin zur Nichterfüllung von Anforderungen und somit zum Scheitern des Projekts führen.

### 6.1.1 Aufbau einer RCP-Applikation

Ausschlaggebend für die gewählte Struktur und den Aufbau der Architektur sind die Eigenschaften, die die Applikation erfüllen soll. Das in dieser Arbeit beschriebene System soll dabei vor allem auf die in Kapitel 5.4 beschriebenen nicht-funktionalen Anforderungen abzielen:

Die grundlegende Struktur der Applikation wird bereits durch die Verwendung der Eclipse-Rich client Plattform und deren Bestandteile definiert und muss nicht selbstständig entwickelt werden. Dabei zeichnet sich die RCP-Plattform besonders durch ihre Modularität im Bezug auf Erweiterungspunkte aus. Diese sind für die nachfolgende Implementierung und somit die Gestaltung der Architektur von großer Bedeutung.

Über Erweiterungspunkte können die Funktionen von bereits bestehenden Plug-ins zur Verfügung gestellt werden. Jeder hinzugefügte Erweiterungspunkt muss dafür eigenes angelegt und konfiguriert werden, auch liefern einige Erweiterungen lediglich Interfaces und benötigen somit eine konkrete Implementierung, die in der Architektur berücksichtigt werden muss.

Das Herzstück einer Eclipse-RCP Applikation bilden dabei die Dateien *MANIFEST.MF*, *plugin.xml* und *build.properties*. Diese enthalten nicht nur Metadaten, wie zum Beispiel die Versionsnummer oder den Namen der Applikation, sondern ermöglichen es auch, benötigte Abhängigkeiten und Erweiterungspunkte zu spezifizieren und zum Projekt hinzuzufügen. Weiters werden die Eigenschaften des Buildprozesses definiert. Zur einfacheren Handhabung bietet die Eclipse-Plattform mit dem *Manifest-Editor* einen graphischen Editor an, wodurch dem Benutzer eine strukturierte Repräsentation dieser Daten geboten wird, welche deren Bearbeitung erheblich vereinfacht. Im Hintergrund wird dabei jede Aktion von der Plattform automatisch im entsprechenden Format in der jeweiligen Datei ergänzt. Abbildung 6.1 zeigt die Darstellung des *Manifest-Editors* mit bereits hinzugefügten Erweiterungen.

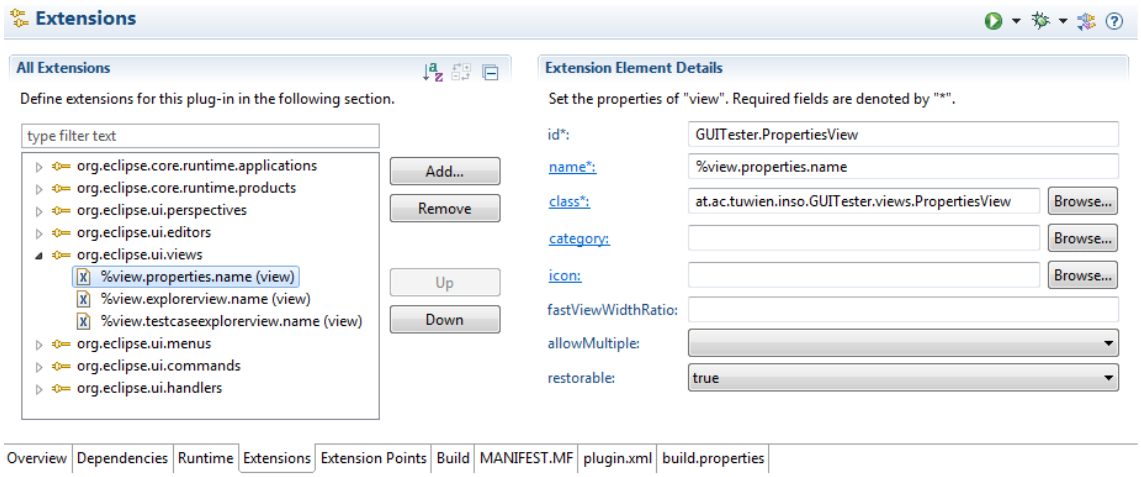



**Abbildung 6.1:** MANIFEST.MF mit gesetzen Erweiterungen

Die in Abbildung 6.1 dargestellte Erweiterung *org.eclipse.ui.views* erweitert die Applikation um Sichten, im hier dargestellten Fall ist die Eigenschaften-Sicht (beschrieben in Abschnitt 6.2.2) selektiert. Der Name der Sicht wird dabei mittels Internationalisierung aus einer Sprachdatei geladen und die referenzierte Klasse ist die Datei *PropertiesView.java*.

Die weiteren aus Abbildung 6.1 ersichtlichen Erweiterungspunkte sind:

**org.eclipse.ui.perspectives** - Definiert die Standardperspektive und ermöglicht die Erweiterung um zusätzliche Perspektiven.

**org.eclipse.ui.editors** - Ermöglicht das Benutzen von Editoren.

**org.eclipse.ui.menus** - Fügt Menüelemente in der Toolbar oder über Kontextmenüs zum Projekt hinzu.

**org.eclipse.ui.commands** - Definiert Befehle, die über Menüs ausgeführt werden.

**org.eclipse.ui.handlers** - Zugehörig zu den Befehlen. Jeder Befehl benötigt einen Handler, der seine Aktion durchführt, wenn der Befehl ausgelöst wird.

### 6.1.2 Aufbau der umgesetzten Applikation

Um die bereits erwähnten Anforderungen und Eigenschaften an die Architektur zu erfüllen wird diese in zwei Subprojekte unterteilt, wobei jedes mehrere Module enthält. Die Aufspaltung und Trennung dieser Teile in unterschiedliche Module unterstützt nicht nur das Prinzip des Separation-of-Concerns, sondern erhöht auch die Austauschbarkeit und ermöglicht somit eine flexiblere Anpassung der Projektteile.

Abbildung 6.2 zeigt die Module der beiden Projekte und deren hierarchische Struktur. Auf oberster Ebene befinden sich die folgenden zwei Projekte:

- RCP-Applikation
  Dieses Projekt ist das Hauptmodul der vorgestellten Applikation. Es umfasst die Struktur der Rich Client Plattform und stellt somit sowohl die Benutzeroberfläche als auch die ausführbare Basis des Projekts dar. Zusätzlich zu den in Abbildung 6.2 dargestellten Submodulen, befinden sich alle für Konfiguration oder Branding relevanten Strukturen innerhalb dieses Projekts.

  Die Submodule unterteilen sich in:

  - main
    Dieses Modul enthält alle für den Start der Applikation relevanten Dateien.
  - config
    Hier befindet sich die Mechanismen zum Auslesen und Verarbeiten der Konfigurationsdateien, die zum Beispiel für Internationalisierung oder das Laden von externen Dateien – wie Icons – benötigt werden.
  - util
    Im Modul *util* befinden sich die für kleinere Aufgaben benötigten Komponenten, bei denen eine Unterteilung in weitere Submodule keinen Nutzen mit sich brächte.

  - codegenerationadapter
    Dieser Teil der Applikation definiert die Schnittstelle zum *Code Generator* Projekt. Hier werden sämtliche Daten konvertiert und für die Kommunikation der beiden Module vorbereitet.
  - entity
    Dieses Modul enthält sowohl Entitäten, die Informationen bezüglich der Testdateien und Testfälle sowie der benutzen Modelle kapseln, als auch alle relevanten Daten weiterer, benutzerdefinierbarer Eigenschaften einer Entität, die in der Eigenschaften-Sicht (siehe Abschnitt 6.2.2) näher erläutert werden.
  - gui
    Im Modul *gui* befinden sich alle Komponenten, die mit der graphischen Benutzeroberfläche interagieren oder für deren korrekte Darstellung relevant sind.

- Code Generator
  Dieses Projekt umfasst die für die Generierung der Testklassen erforderliche Logik und wird als externe Abhängigkeit zum Hauptprojekt hinzugefügt, wodurch eine lose Kopplung erreicht wird. Hier wird ausschließlich die Problematik der Code-Generierung im Hinblick auf die durch die Test-Frameworks JUnit bzw. TestNG bereits definierte Struktur der Testfälle behandelt. Ein besonders wichtiger Aspekt ist dabei die Erweiterbarkeit des Moduls im Hinblick auf zusätzliche Funktionalität sowie hinsichtlich der Unterstützung zusätzlicher Test-Frameworks.

  Dieses Projekt setzt sich – wie in Abbildung 6.2 erkennbar – aus den folgenden drei Modulen zusammen:

  - codemodel.extension
    Für die Generierung des Quellcodes wird die Open-Source Bibliothek *CodeModel*[1] herangezogen und erweitert. Dieses Submodul enthält Adaptionen und Erweiterungen des Frameworks.
  - generator
    Das Modul *generator* enthält die Logik der Generierung und bietet Interfaces und abstrakte Klassen als Basis für zukünftige Erweiterungen.
  - entity
    Dieses Modul enthält Entitäten, die Informationen bezüglich der zu generierenden Testdateien und Testfälle kapseln und verarbeiten. Eine Konvertierung in die hier angeführten Entitäten ist für eine erfolgreiche Verwendung des Code-Generator-Projekts erforderlich.

[1] https://codemodel.java.net

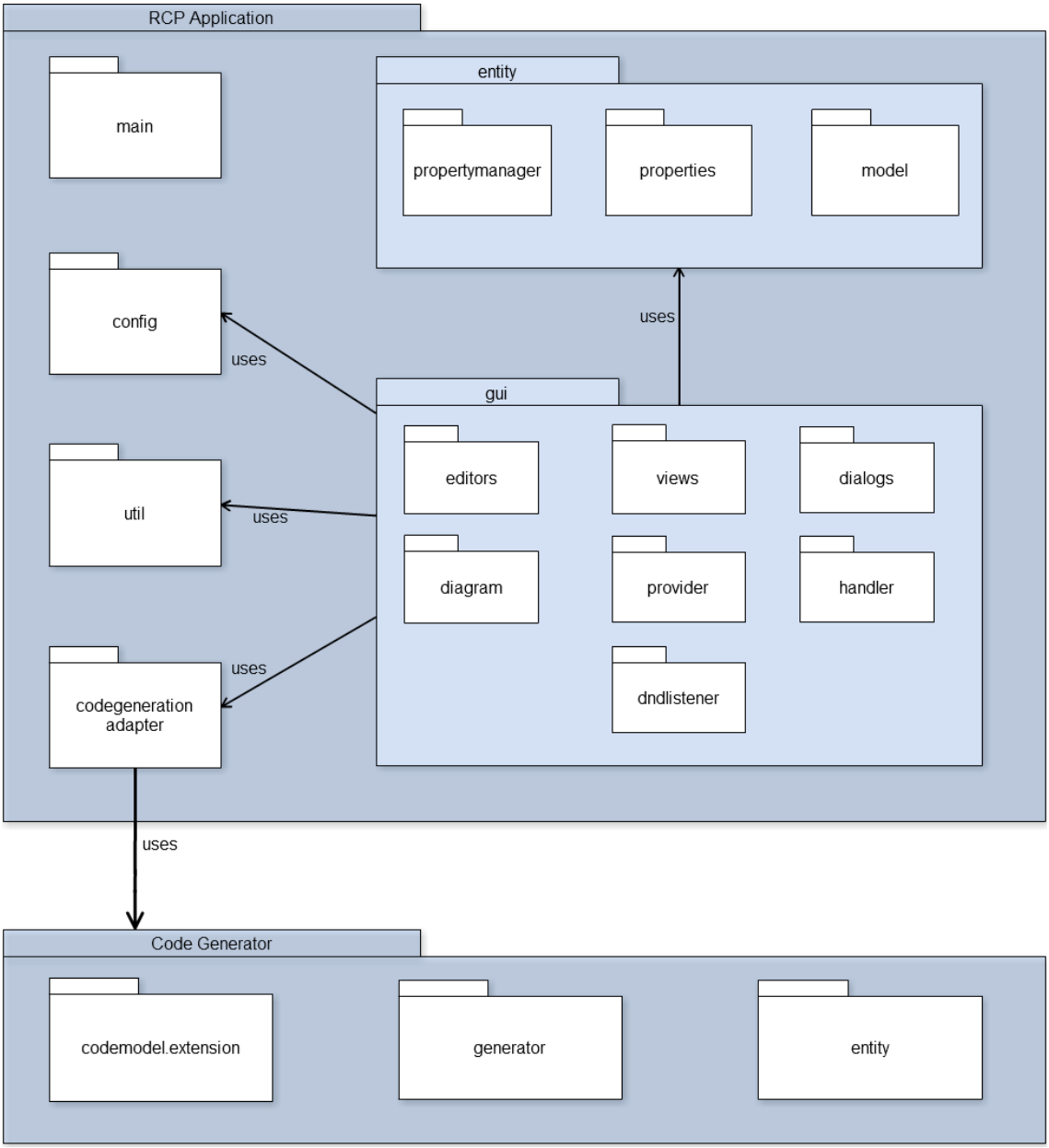



**Abbildung 6.2:** Module-View der Applikation

### 6.1.3 Modellbeschreibung

Dieser Abschnitt erläutert die verwendeten Modelle und gibt eine kurze Erklärung bezüglich deren Nutzen und Relevanz sowie die Vorteile dieser Herangehensweise.

Gemäß der von Roßner u. a. [42] in Abschnitt 2.6.4 vorgestellten Definition von Modellen handelt es sich bei diesem Ansatz um modellgetriebenes Testen mit Systemmodellen. Roßner u. a. definieren weitere Untertypen eines Systemmodells und spezifizieren in weiterer Folge den Begriff eines Testbasismodells:

> *„[Ein Testbasismodell] modelliert für den Test relevante Teile der Testbasis wie z.B. zu testende Struktur und Verhaltensmerkmale des SUT [Anm: dabei handelt es sich um das zu testende System] oder seiner Umgebung und dient zur Generierung der Testfälle.“*

Das Modell repräsentiert die graphische Benutzeroberfläche der zu testenden Applikation und stellt somit eine Schnittstelle zwischen dieser und dem Testtool selbst dar. Abbildung 6.3 verdeutlicht die Beziehung zwischen dem Modell, dem Testtool und der zu testenden Applikation. Weiters wird in Abbildung 6.3 der Vorteil gegenüber der in Abschnitt 5.3 erwähnten Problematik bezüglich Capture/Replay-Tools abgebildet. Wie hier deutlich zu erkennen ist, genügt bei einer veränderten graphischen Benutzerschnittstelle der zu testenden Applikation - solange die Funktionalität erhalten bleibt - eine Anpassung des Modells. In diesem Fall können bereits erstellte Testfälle weiterhin genutzt werden.

Zusätzlich fungiert das Modell als Abstraktionsschicht zwischen dem Testtool und der zu testenden Applikation, woraus sich eine lose Kopplung zwischen diesen ergibt. Außerdem hat dies den Vorteil, dass das Testtool nicht auf einen bestimmten Applikationstypen beschränkt ist.

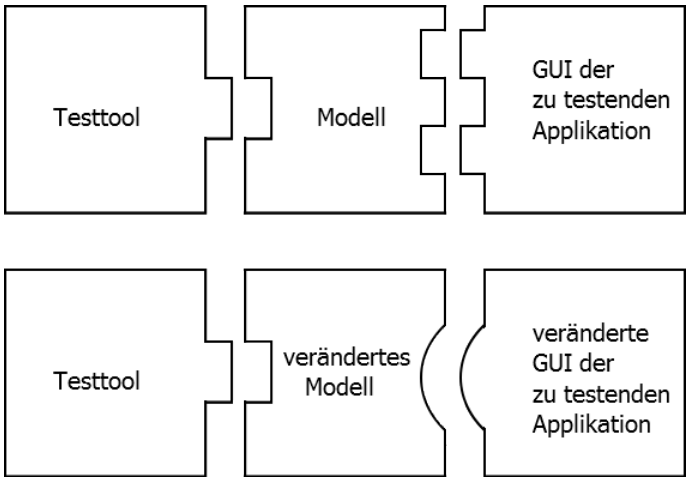



**Abbildung 6.3:** Darstellung des Modells als Schnittstelle zwischen Testtool und zu testender Applikation

Im Konkreten handelt es sich bei den hier erwähnten Modellenum Java-Klassen, die über ihre Methoden einen Zugriff auf die Benutzerschnittstelle des zu testenden Systems ermöglichen. Durch die standardisierte Struktur einer Java-Datei ergeben sich in diesem Bereich keinerlei Abhängigkeiten zwischen der Applikation und dem gewählten Modell, was abermals in einer losen Kopplung resultiert. In diesem Zusammenhang ist zu erwähnen, dass die in dieser Arbeit vorgestellte Applikation jegliche Arten von bereits kompilierten Java-Klassen als potentiellen Input akzeptiert und Testfälle basierend auf diesen Klassen erstellt werden können. Allerdings ist der Fokus dieser Applikation auf die Erstellung von GUI-Tests ausgelegt. Die Generierung von anderen Testarten scheint nicht zweckmäßig, da diese neben Methodenaufrufen und Zusicherungen noch andere programmatische Konstrukte erfordern können.

Quellcodebeispiel 6.1 zeigt ein beispielhaftes Modell einer Suchseite, Abbildung 6.4 bildet das zugehörige Java-Modell ab, wie es zum Beispiel mit *Selenium* [45], einer Testumgebung für Webapplikationen, erzeugt werden könnte. Das hier abgebildete Modell stellt lediglich eine der Möglichkeiten der späteren Benutzung der hier geschaffenen Applikation dar. Aufgrund der Struktur einer Java-Klasse ist es möglich jegliche Art von Java-Dateien als Eingabedatei für die Applikation zu verwenden und basierend auf den definierten Methoden Testfälle bzw. Testskripts zu spezifizieren und zu generieren. Eine detailliertere Darstellung des Modells, sowie dessen Weiterverarbeitung findet sich in Anhang A.


**Abbildung 6.4:** Darstellung der Webseite, des in Quellcodebeispiel 6.1 abgebildeten Modells.

```
1  /**
2  * Model of the Search Page
3  */
4  @ModelDesc(name="SearchPage")
5  public class SearchPage extends WebPage<Driver>
6  {
7    @Field(id="searchField", locator="ByName||q")
8    private TextField searchField;
9
10   @Field(id="submit", locator="ByName||btnG",
11     navigateTo = ResultPage.class)
12   private Button searchButton;
13
14   public ResultPage search(String text)
15     throws DriverException
16   {
17     searchField.setValue(text);
18     return (ResultPage) searchButton.click();
19   }
20 }
```

**Quellcode 6.1:** Implementierungsausschnitt - Modell der Suchseite

## 6.2 Benutzeroberfläche

Dieser Abschnitt bietet einen Einblick in den geplanten Aufbau und die Gestaltung der Benutzeroberfläche unter besonderer Betrachtung der in Kapitel 3 erwähnten Heuristiken und Guidelines. Weiters wird auf die für die Java-Programmierung häufig benutzte Entwicklungsumgebung Eclipse [54] eingegangen, welche als zentraler Ausgangspunkt für die zu entwerfende GUI dient. Abschließend folgt eine Evaluierung detaillierter Skizzen der in dieser Arbeit geplanten Benutzeroberfläche.

### 6.2.1 Eclipse Entwicklungsumgebung

Die integrierte Entwicklungsumgebung (IDE) Eclipse [54], ebenfalls eine RCP-Applikation, stellt eine der am meisten verwendeten Entwicklungsumgebungen für die Java-Programmierung dar [31]. Die primäre Benutzergruppe der Eclipse-IDE sind also hauptsächlich Entwickler. Die in dieser Arbeit erstellte Applikation soll sowohl für Software-Entwickler als auch Tester förderlich sein und von beiden Benutzergruppen verwendet werden können. Da die Tätigkeiten des Testens und Entwickelns oft nahe beieinander liegen und kombiniert werden können und die Applikation ebenso auf dem Eclipse-RCP Framework basiert, soll die GUI auch an die Eclipse-IDE erinnern und einen ähnlichen Aufbau aufweisen.

Diese Ähnlichkeit in der Benutzeroberfläche soll zum einen die von Nielsen und Shneiderman in Abschnitt 3.2 erwähnte Richtlinie nach universeller Benutzbarkeit bzw. Konsistenz erfüllen, zum anderen soll ein Umstieg von der Tätigkeit des Testens zum Entwickeln und vice versa einfach und schnell ermöglicht werden, indem keine zusätzliche Einarbeitungszeit benötigt wird und ein konsistenter Aufbau der Benutzeroberfläche ein schnelles und sicheres Arbeiten ermöglicht.

Die GUI der Eclipse-IDE besteht dabei wie in [31] beschrieben aus fünf verschiedenen Komponenten. Abbildung 6.5 zeigt einen Screenshot von Eclipse und hinterlegt die beschriebenen Bereiche farbig.

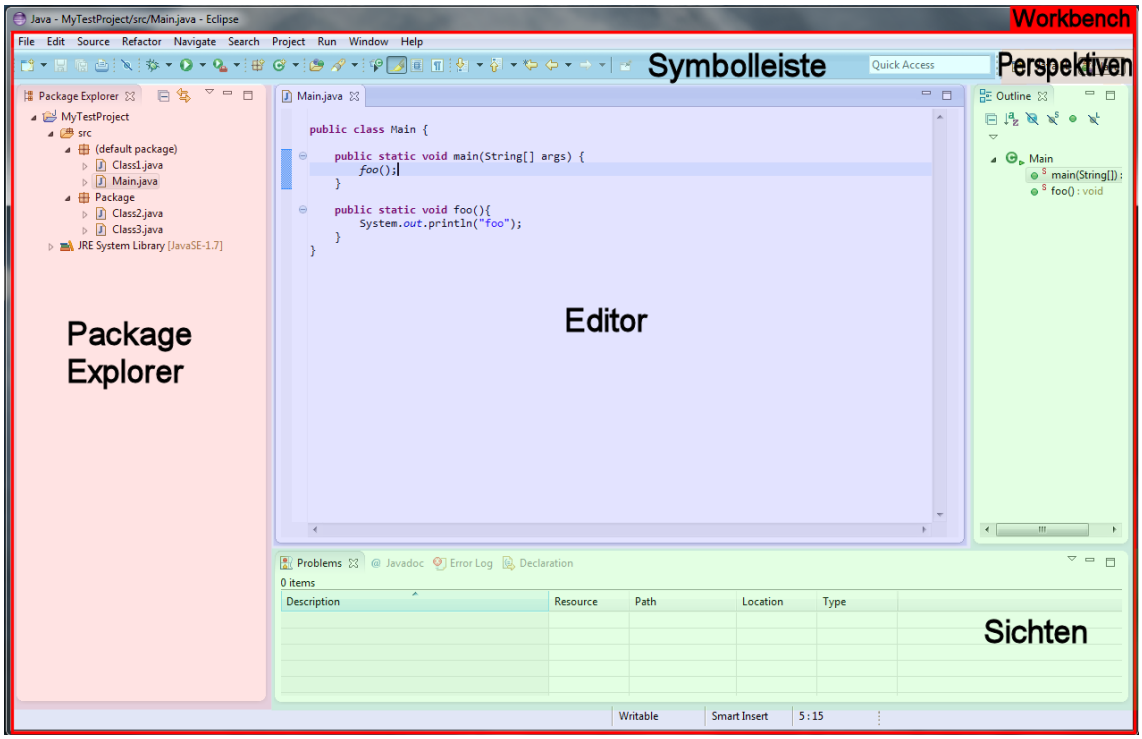



**Abbildung 6.5:** Komponenteneinteilung der GUI der Eclipse-IDE

Die farbig hervorgehobenen Bereiche sind

- Workbench:
  Die Workbench integriert alle Bestandteile der Applikation innerhalb eines Fensters und stellt somit den Rahmen der Applikation dar.

- Perspektiven:
  Als Perspektive wird ein Container für Sichten und Editoren, der die Anordnung derselben beschriebt, bezeichnet. Perspektiven beinhalten je nach Funktionalität unterschiedliche Sichten und Editoren, zum Beispiel beinhaltet die Perspektive für das Debuggen einer Applikation andere Sichten als die für die Entwicklung. Der Benutzer kann wählen, welche Perspektive er gerade benötigt, und diese darstellen lassen.

- Symbolleiste:
  Die hier vorhandenen Buttons und Elemente bieten einen schnellen Zugriff auf wichtige und häufig verwendete Funktionen und ermöglichen dadurch eine effizientere Arbeitsweise.

- Package Explorer:
  Der Package Explorer bietet eine Übersicht über unterschiedliche Projekte, deren hierarchischen Aufbau und über die zugehörigen Dateien.

- Editor:
  Der Editor ermöglicht das Bearbeiten von bestehenden Dateien und das Schreiben von Programmcode und stellt den zentralen Bereich der Applikation dar. Mehrere Editoren werden innerhalb des Editorbereichs mittels mehrerer Tabs angeordnet und können je nach Bedarf angezeigt oder zur besseren Darstellung anders angeordnet werden.

- Sichten:
  Unterschiedliche Sichten stellen Informationen in unterschiedlichen Detailgraden dar. Diese können nach Belieben angezeigt, ausgeblendet und innerhalb des Sichten-Bereichs angeordnet werden. Typische Beispiele für häufig benutzte Sichten sind die Konsole zum Ausführen eines Programms, der Fortschritt bei Komponententests oder detailliertere Informationen eines Objekts während des Debuggens. Über zusätzliche Plug-Ins können weitere Sichten zur Eclipse-IDE hinzugefügt werden.

### 6.2.2 GUI-Skizzen

In diesem Abschnitt werden im Rahmen des in Abschnitt 3.3 genannten Designprozesses Skizzen der zukünftigen Benutzeroberfläche dargestellt, näher betrachtet und gemäß der Richtlinien und Heuristiken aus Abschnitt 3.2 evaluiert.

Abbildung 6.6 zeigt, bezugnehmend auf Abbildung 6.5, den Aufbau der geplanten Applikation und die Übereinstimmung der Bereiche mit der Eclipse GUI.

Dabei ist gut erkennbar, dass Abbildung 6.6 in den farbig hervorgehobenen Bereichen mit Abbildung 6.5 nahezu übereinstimmt. Eine Ausnahme bilden dabei die Perspektiven, was damit begründet ist, dass in der geplanten Applikation keine zusätzlichen Perspektiven vorgesehen sind. Eine Erweiterungsmöglichkeit für zusätzliche Perspektiven ist jedoch vorhanden.

Nachfolgend wird die entsprechende Funktionalität der jeweiligen Bereiche erklärt und anhand detaillierterer Graphiken erläutert.

#### Symbolleiste

Die Symbolleiste befindet sich im oberen Bereich des Applikationsfensters und ermöglicht dem Benutzer einen schnellen Zugriff auf häufig benutzte Funktionen des Systems. Die Skizze in Abbildung 6.7 zeigt die geplante Symbolleiste mit den wesentlichsten Funktionen des Systems.

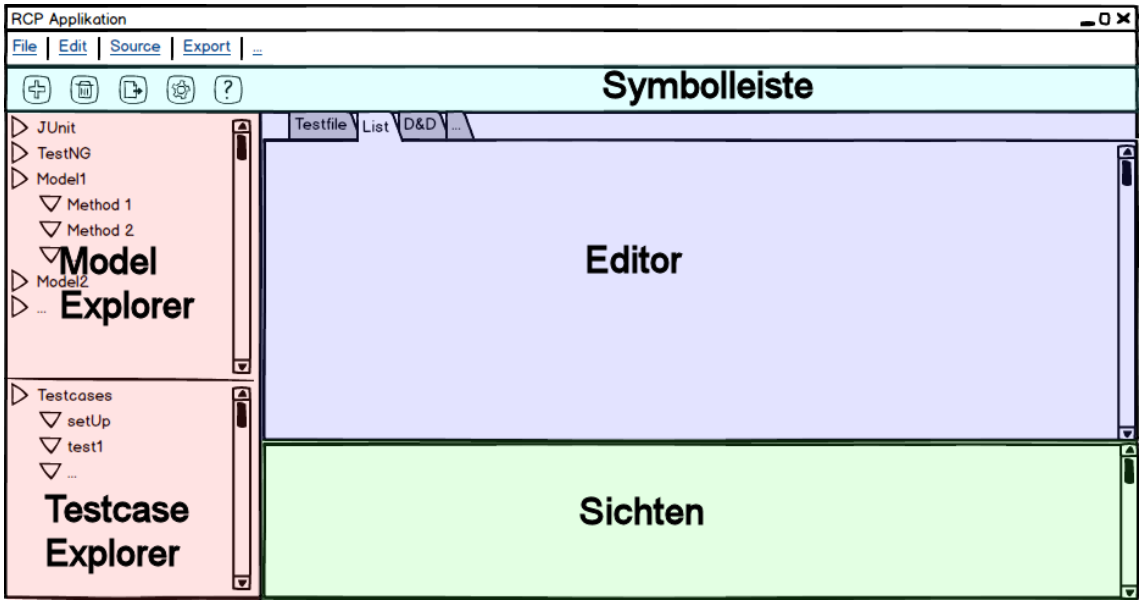



**Abbildung 6.6:** Komponenteneinteilung der GUI des Testtools

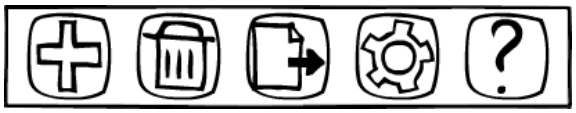


**Abbildung 6.7:** GUI Skizze der Symbolleiste

Die Symbolleiste soll unter anderem die folgenden Funktionen beinhalten:

- Das Hinzufügen von neuen Elementen, wie zum Beispiel Modellen.
- Das Löschen von ausgewählten Elementen.
- Das Exportieren von Testfällen.
- Einen schnellen Zugriff auf Konfigurationen des Systems.
- Einen Verweis auf eine Hilfefunktion oder das Benutzerhandbuch.

Bezüglich der in Abschnitt 3.2 genannten Richtlinien würde eine Toolbar die folgenden Kriterien erfüllen:

- Konsistenz
  Diese Heuristik ist aufgrund der hohen Ähnlichkeit zur Eclipse-IDE sowie eingehaltenen Konventionen bezüglich verschiedener Plattformen und der allgemeinen Struktur von Programmen als erfüllt zu betrachten.
- Effizienz und Flexibilität & Benutzerkontrolle
  Symbolleisten sind ein häufig verwendetes Element von Client-Applikationen, sie ermöglichen einen schnellen Zugriff auf konkrete Funktionen und bieten dem Benutzer somit eine gezielte Kontrolle sowie ein effizientes Arbeiten.

- Universelle Benutzbarkeit & die Sprache des Benutzers sprechen
  Durch aussagekräftige Icons und Tooltipps ist gewährleistet, dass sowohl Einsteiger, als auch erfahrene Benutzer einen einfachen Umgang mit der Symbolleiste erleben. Meist repräsentieren Icons eine Metapher der realen Welt und sind einer konkreten Funktion zugewiesen. Diese Umstände spiegeln die Heuristiken nach universeller Benutzbarkeit und der Sprache des Benutzers wider.

- Erkennen statt Erinnern
  Durch effizienten Einsatz der Symbolleiste ist es nicht notwendig, dass sich der Benutzer Shortcuts oder explizite Reihenfolgen merkt, um eine gewisse Funktion ausführen zu können, was in der Erfüllung der Heuristik des Erkennens statt Erinnerns resultiert.

#### Explorer

Die in Abbildung 6.8 dargestellte Explorer-Sicht ist in zwei wesentliche Bestandteile getrennt: den Model und den Testcase Explorer. Der obere Bereich (Model-Explorer) stellt die in der Applikation geladenen Modelle dar und ermöglicht den Zugriff auf deren Methoden. Zusätzlich befinden sich im Model-Explorer unter dem Punkt JUnit bzw. TestNG testspezifische Methoden und Funktionen, wie zum Beispiel Zusicherungen.

Der untere Teil der Explorer-Sicht stellt den Testcase-Explorer dar. Hier werden alle aktuell erstellten Testfälle angezeigt. Die hier dargestellten Informationen dienen im späteren Verlauf dazu, einzelne Testfälle nachträglich auszuwählen und zu bearbeiten bzw. diese Testfälle einer Testklasse zuzuteilen sowie zusätzliche Eigenschaften zu konfigurieren.

In diesem Bereich beschränkt sich die Anzahl der genutzten GUI Elemente auf einen Tree-Viewer zur hierarchischen Darstellung der Modelle und ihrer Methoden bzw. der Testklassen und der Testfälle. Die gewählte Gestaltung dieses Bereichs der Benutzeroberfläche erfüllt die folgenden in Abschnitt 3.2 erwähnten Heuristiken:

- Konsistenz
  Diese Heuristik ist aufgrund der hohen Ähnlichkeit zur Eclipse-IDE sowie eingehaltenen Konventionen bezüglich verschiedenen Plattformen und GUI-Elementen gewährleistet. Der Explorer-Bereich stellt eine Übersicht der bereits existierenden Elemente und ein Auswahlmenü für die weiteren Aktionen dar und ist im linken Bereich der Applikation platziert, was gängigen Konventionen der Applikationsgestaltung entspricht.

- Universelle Benutzbarkeit & Benutzerkontrolle
  Baumstrukturen repräsentieren eine gebräuchliche Darstellungsform für hierarchisch zusammenhängende Elemente und sind für einen Benutzer keine Neuerung. Die Heuristik der universellen Benutzbarkeit ist somit als erfüllt zu betrachten.
  Zusätzlich bieten sie dem Benutzer die Kontrolle über ihre Darstellung. Nach Belieben können Elemente ein- oder ausgeklappt werden und erleichtern somit den Überblick bzw. stellen nur vom Benutzer gerade benötigte Elemente dar.

- Erkennen statt Erinnern & Minimalistisches Design
  Der natürliche hierarchische Aufbau von Baumdarstellungen hilft dem Benutzer zusammenhängende Komponenten zu erkennen und benötigt keinerlei weitere Erklärungen, wodurch die Heuristik des Erkennens statt Erinnerns als erfüllt zu betrachten ist. Weiters bietet die Darstellung einer Baumstruktur ein einfaches und übersichtliches Design, welches aufgrund seiner Beschaffenheit äußerst minimalistisch gehalten werden kann.

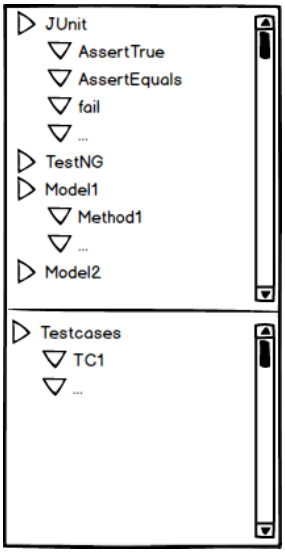



**Abbildung 6.8:** GUI Skizze der Explorer Sicht

### Editor

Wie auch in der Eclipse-IDE stellt der Editor in der vorgestellten Applikation den zentralen Arbeitsbereich dar. Hier ist es dem Benutzer möglich, die Testfälle anhand der Methoden der importierten Modelle zu erstellen und zu bearbeiten. Äquivalent zur Eclipse-IDE werden mehrere geöffnete Editoren mittels Tabs voneinander getrennt.

Jeder neu erstellte Testfall wird dabei in einem eigenen Tab dargestellt, dabei unterteilt sich die Darstellung eines Testfalls in zwei unterschiedliche Formen. Eine Listendarstellung, wie in Abbildung 6.9 zu sehen und einer an einem UML-Aktivitätsdiagramm orientierten Repräsentation, wie in Abbildung 6.10 dargestellt.

Die Interaktion beläuft sich bei beiden Darstellungsformen auf eine einfache Drag&Drop-Aktion. Die Methoden eines Modells können via Drag&Drop aus dem Explorer in den Editor gezogen werden und werden dort gemäß ihrer Reihenfolge im Testfall platziert.

Der in Listing 6.2 abgebildete Testfall ist zum besseren Verständnis der Abbildungen 6.9 und 6.10 ebenfalls dargestellt. So beinhaltet Abbildung 6.9 die sequentielle Abfolge der im Testfall ausgeführten Aktionen als Listendarstellung und stellt detailliertere Informationen bezüglich des Testfalls und der aufgerufenen Methoden dar. Abbildung 6.10 orientiert sich an der Repräsentation eines UML-Aktivitätsdiagramms. Dieses ist hierfür besonders geeignet, da hier, wie in [29] beschrieben, der Kontroll- und Datenfluss zwischen Aktionen einer Aktivität dargestellt wird.

```
1 @Test
2 public void testLoginFunction_shouldSuccesfullyLogin(){
3   clickOnLogin();
4   enterUsername("Administrator");
5   enterPassword("1a2b3c4d");
6   clickOnOK();
7   AsserTrue(isLoggedIn());
8 }
```

**Quellcode 6.2:** Beispiel Testfall

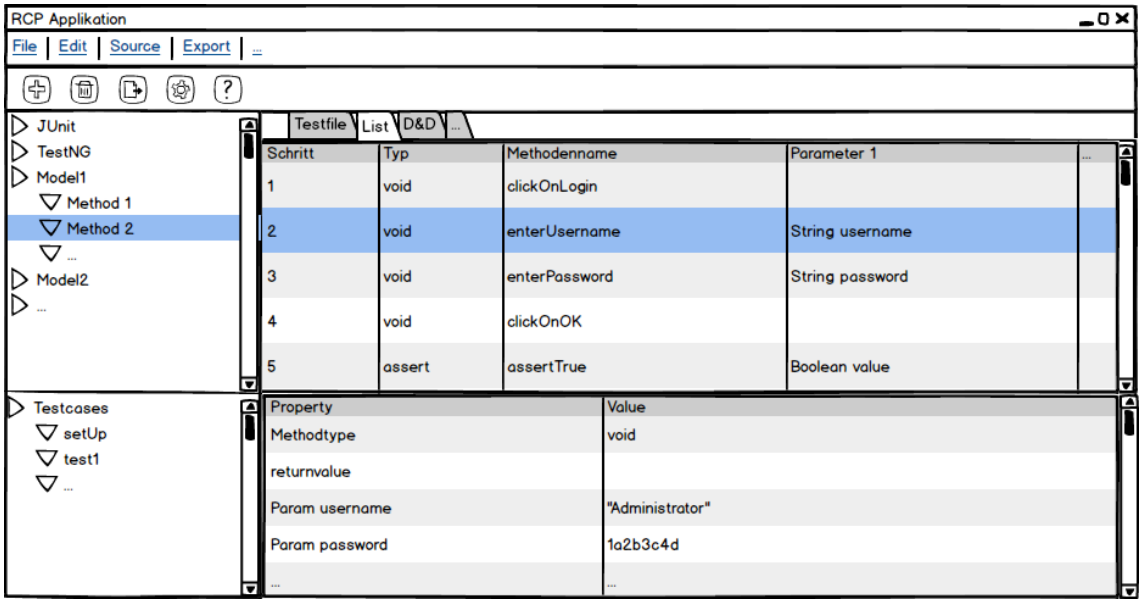



**Abbildung 6.9:** GUI Skizze mit Testfall in Listendarstellung

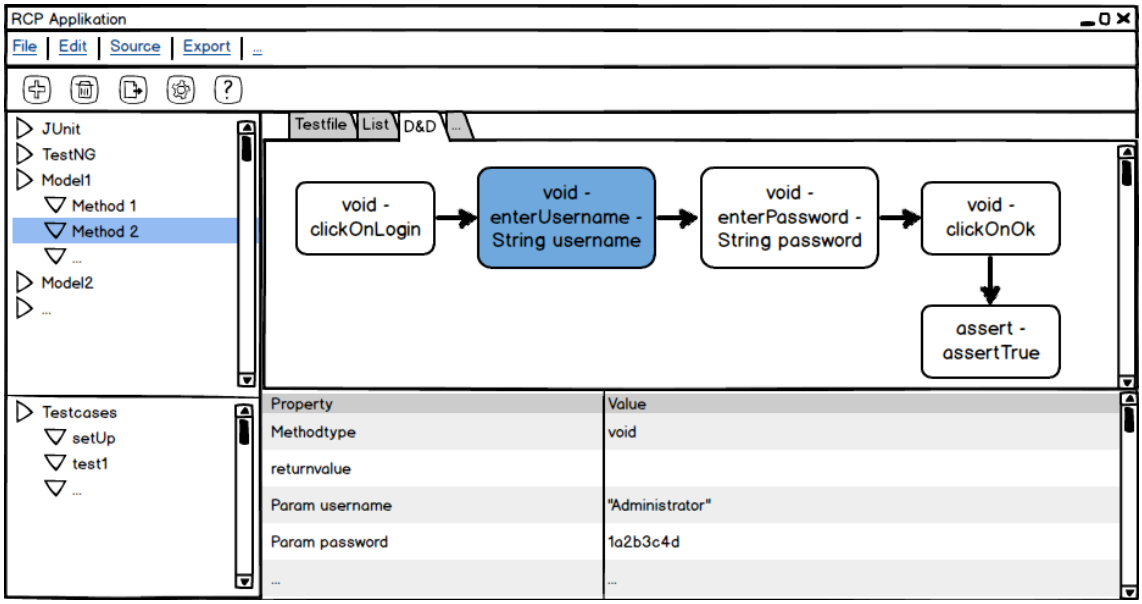



**Abbildung 6.10:** GUI Skizze mit Testfall in der Diagramm-Ansicht

Zusätzlich zum Erstellen von einzelnen Testfällen ist es für das spätere Exportieren der Tests notwendig, Testfälle in Testdateien bzw. Testklassen zu gruppieren. Hierzu wird der Tab *Testfile* benötigt, welcher eine konkrete Java-Datei – bestehend aus mehreren Tests – repräsentiert. In diesem Editor soll es möglich sein, aus mehreren bereits erstellten Testfällen eine Testdatei zusammenzustellen und diese anschließend als Java-Datei zu exportieren. Abbildung 6.11 zeigt eine beispielhafte Darstellung einer Datei, bestehend aus vier Testfällen.

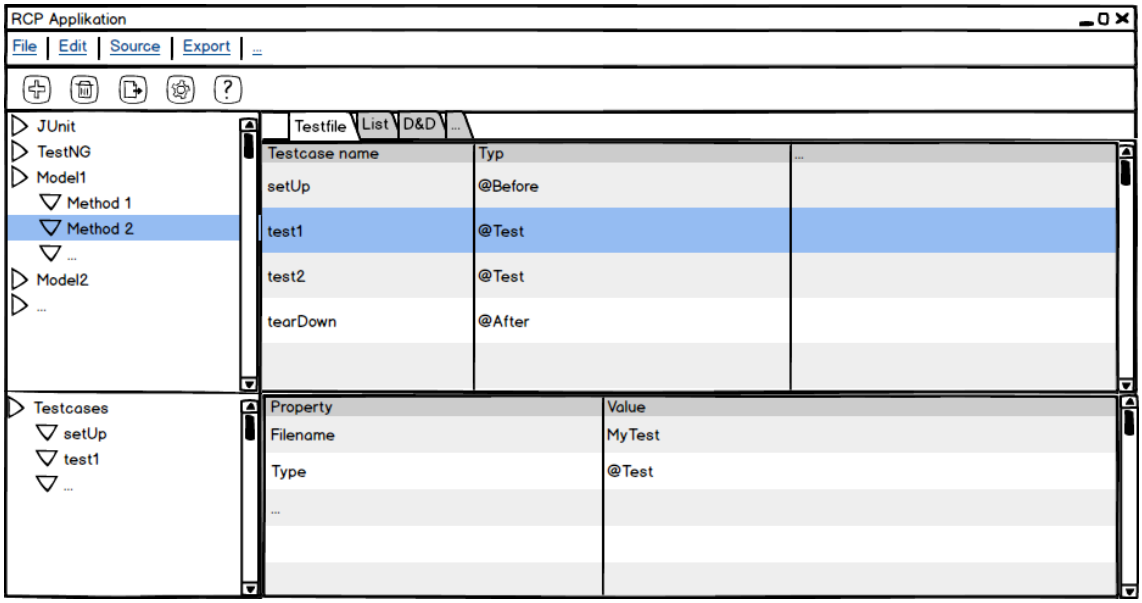



**Abbildung 6.11:** GUI Skizze der Übersicht der Erstellten Testfälle

Die gewählte Gestaltung dieses Bereichs der Benutzeroberfläche erfüllt die folgenden in Abschnitt 3.2 erwähnten Heuristiken:

- Universelle Benutzbarkeit
  Unterschiedliche Benutzergruppen können je nach individueller Vorliebe die Listen- oder Diagrammrepräsentation eines Testfalles benutzen und auf exakt die gleiche Art und Weise bearbeiten. Auch ein Wechsel zwischen den beiden Repräsentationsformen ist ohne Probleme möglich. Weiters ermöglicht die Eclipse-Plattform eine individuelle Anordnung der Menü-Elemente. So könnten erfahrene Benutzer zum Beispiel mehrere Editoren gleichzeitig geöffnet haben, die Sichten minimieren, um einen größeren Editorbereich zur Verfügung zu haben oder die Anordnung der Elemente selbstständig neu gestalten. Diese Eigenschaften kommen der Richtlinie für universelle Benutzbarkeit und Unterscheidung zwischen verschiedenen Benutzergruppen zugute und erfüllen diese.

- Informatives Feedback & Abgeschlossenheit
  Die Heuristiken hinsichtlich dem Benutzer gegebenen Feedback werden aufgrund der folgenden Eigenschaften als erfüllt betrachtet. Nimmt der Benutzer eine Aktion im Editor-Bereich vor, werden die Auswirkungen derselben unmittelbar nach deren Abschluss im Editor dargestellt. Der Benutzer ist somit immer darüber informiert, welche Aktionen er gerade getätigt hat und weiß, dass diese bereits beendet sind.

- Fehler vermeiden, Umkehrbarkeit & Benutzerkontrolle
  Diese Heuristiken werden durch simple Validierungsmechanismen erfüllt. Der Benutzer hat zwar die Kontrolle über das System, dieses hilft ihm jedoch, indem es potentielle Fehler nicht aufkommen lässt. So wird beispielsweise Drag&Drop mit nicht dafür vorgesehenen Elementen unterbunden, der Editor würde diese ignorieren und verhindert somit einen Fehler des Benutzers. Weiters sind alle vom Benutzer durchgeführten Aktionen erneut durchführbar und dadurch auch umkehrbar.

- Ästhetisches und Minimalistisches Design
  Besonders die Diagrammdarstellung eines Testfalls zeigt die Erfüllung dieser Heuristik. Alle im Editor dargestellten Informationen sind für den Benutzer relevant und ausschlaggebend. Zusätzliche Informationen stehen dem Benutzer allerdings über die Sichten zur Verfügung.

### Sichten

In diesem Bereich werden jene Details dargestellt, die im Editor zwecks Übersichtlichkeit und minimalistischen Designs nicht enthalten sind. Der Sichten Bereich wird unter anderem zur Darstellung und Vervollständigung der Eigenschaften eines Testfalls, einer Testdatei oder eines Methodenaufrufs benötigt. Abbildung 6.12 zeigt die Darstellung der Eigenschaften-Sicht für einen Testfall. Hier werden zum ausgewählten Testfall, Testdatei oder Methode zusätzliche vom Benutzer festgelegte Eigenschaften hinzugefügt. Zum Beispiel kann ausgewählt werden, wann der Testfall ausgeführt werden soll bzw. welche Annotation er besitzt. Weiters können etwaige benötigte Parameter, wie zum Beispiel eine erwartete Exception, die maximale Dauer des Testfalls gesetzt oder andere Eigenschaften ergänzt werden, sowie konkrete Werte für Parameter der aufgerufenen Methoden gesetzt werden. Listing 6.3 zeigt den in Abbildung 6.12 dargestellten Testfall.

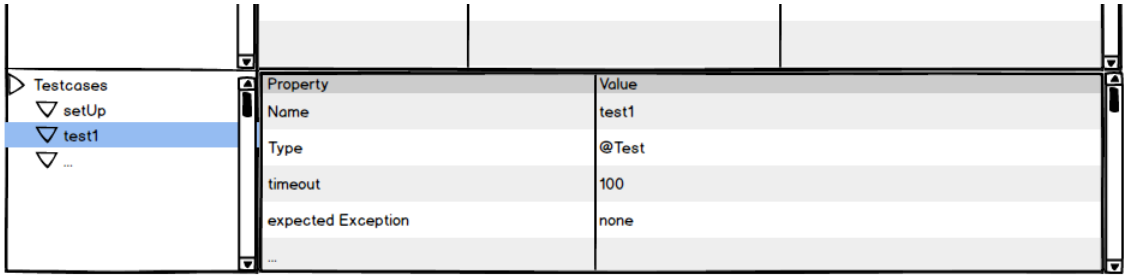



**Abbildung 6.12:** GUI Skizze der Eigenschaften-Sicht eines Testfalls

```
1 @Test(timeout=100)
2 public void test1(){
3    ...
4 }
```

**Quellcode 6.3:** Testfall mit gesetzen Eigenschaften in der Eigenschaften-Sicht

Die gewählte Gestaltung dieses Bereichs der Benutzeroberfläche erfüllt somit die folgenden in Abschnitt 3.2 erwähnten Heuristiken:

- Fehler vermeiden & Umkehrbarkeit
  Benutzer können mittels standardisierter Eingabeelemente wie Textfeldern oder Comboboxen die Eigenschaften der ausgewählten Konstrukte verändern. Mittels standardmäßig gesetzter Werte, Validatoren und Comboboxen ist sichergestellt, dass der Benutzer bei kritischen Elementen ausschließlich korrekte Eingaben tätigt. Die Applikation hilft dem Benutzer somit, Fehler zu vermeiden. Weiters sind alle vom Benutzer getätigten Änderungen erneut durchführbar und können dadurch rückgängig gemacht werden.

- Informatives Feedback & Abgeschlossenheit
  Durch eine unmittelbare Aktualisierung der Benutzeroberfläche erkennt der Benutzer, wann die von ihm durchgeführten Veränderungen abgeschlossen sind. Der Benutzer sieht die Auswirkungen der von ihm getätigten Veränderungen und weiß somit, welche Aktion er gesetzt hat.

- Erkennen statt Erinnern & Benutzerkontrolle
  Die Gestaltung und Strukturierung der Sicht als Tabelle mit Wertpaaren in der Form *Eigenschaft – Wert* ist übersichtlich und bietet direkte Einsicht in die jeweilige Eigenschaft. Der Benutzer erkennt unmittelbar welche Wertbereiche welche Auswirkungen hervorrufen und hat die absolute Kontrolle über diese.

Natürlich ist der Sichten-Bereich nicht auf die Eigenschaften-Sicht beschränkt. Zukünftige Erweiterungen der Applikation können in diesem Bereich zusätzliche Sichten definieren.

# 7 Implementierung

Das Thema dieses Kapitels ist die konkrete Umsetzung des in Kapitel 5 beschriebenen Projekts als Eclipse-RCP-Applikation. Zu Beginn wird eine grundlegende Beschreibung der verwendeten Entitäten und deren Relationen zum besseren Verständnis der Implementierung gegeben. Anschließend folgt ein Vergleich der umgesetzten Applikation mit dem in Kapitel 6 präsentierten Konzept. Besonders auf die in Abschnittt 5.4 erwähnten funktionalen Anforderungen wird dabei näher eingegangen. Des Weiteren werden Besonderheiten der Applikation sowie nennenswerte Details der Implementierung abgebildet und näher erklärt. Abschließend werden die während der Umsetzung aufgetretenen Probleme erläutert und detailliert betrachtet. Abschließend wird eine Übersicht zu vergleichbaren Werkzeugen im Bereich der Testautomatisierung und des modellbasierten Testens gegeben.

## 7.1 Beschreibung der Entitäten

Die im Rahmen der Implementierung erstellten Entity-Klassen kapseln die für das Projekt benötigten Daten und werden aufgrund ihrer Bedeutung für nachfolgende Erläuterungen an dieser Stelle näher betrachtet. Abbildung 7.1 zeigt einen Ausschnitt der Entitäten als Klassendiagramm und stellt die Relationen der Klassen zueinander dar.

Den jeweiligen Klassen kommen dabei die folgenden Bedeutungen zu:

- `AEntity`
  Stellt eine abstrakte Basisklasse dar. Jede Entität speichert hierbei eine eindeutige ID sowie ihr entsprechendes Elternelement. Weiters enthält diese Klasse Mechanismen, damit Kindelemente zu einer Entität hinzugefügt werden können.

- `Projekt`
  Das Projekt stellt lediglich eine übergeordnete Datenstruktur dar und dient als logische Einheit für eine Vielzahl an Modellen und Testdateien.

- `Model`
  Diese Entität stellt das eingelesene Modell dar, aus welchem die enthaltenen Methoden entnommen und als Instruktion gespeichert werden.

- `Instruction`
  Eine Instruktion stellt eine konkrete Methode eines Modells dar. Diese wird als `java.lang.reflections.Method` gespeichert und enthält somit alle Informationen einer Methode. Weiters bietet eine Instruction die Möglichkeit, Werte für die Parameter der enthaltenen Methode zu definieren.

- `TestFile`
  Repräsentiert eine Testklasse, die nach Erstellung von Testfällen generiert werden kann. Eine Testdatei beinhaltet beliebig viele Testfälle und kann in verschiedene Formate exportiert werden.

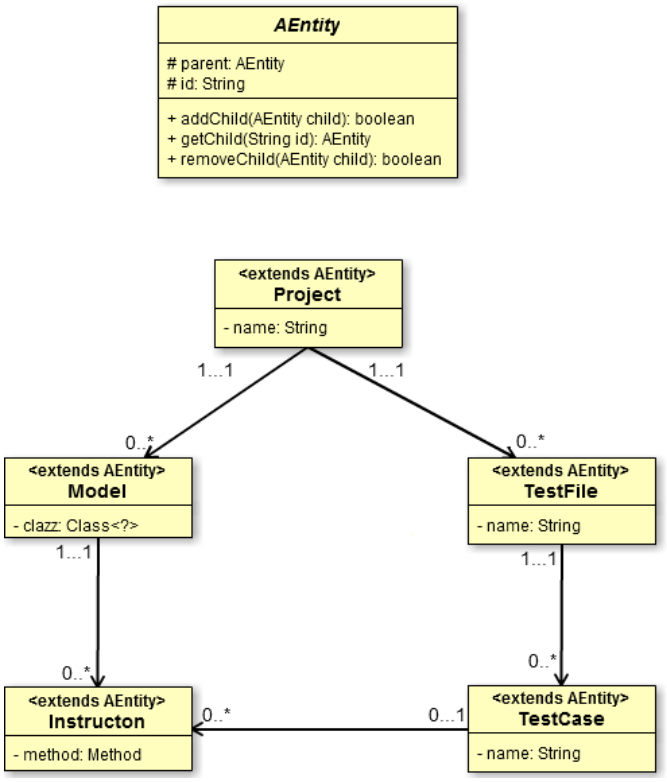



**Abbildung 7.1:** Klassendiagramm der verwendeten Entitäten

- `TestCase`
  Stellt einen einzelnen Testfall dar. Dieser besteht hierbei aus mehreren Instruktionen, die ausgeführt werden. Zusätzlich beinhaltet ein Testfall Informationen bezüglich gesetzter Annotationen und deren Parameter.

## 7.2 Umsetzung der funktionalen Anforderungen

Dieser Abschnitt beschreibt den Funktionsumfang des Systems und stellt sowohl die Umsetzung als auch konkrete Implementierungsdetails vor. Ziel ist es zum einen ein tieferes Verständnis der Applikation und deren Entstehung zu ermöglichen und zum anderen eine Gegenüberstellung des Prototypen im Bezug auf die in Kapitel 6 beschriebene Planungs- und Designphase zu bieten. Der Funktionsumfang wird dabei anhand der in Tabelle 5.2 abgebildeten funktionalen Anforderungen betrachtet.

### 7.2.1 Einlesen und Verarbeiten von Modellen

Wie bereits in Abschnitt 6.1.3 beschrieben, handelt es sich bei den verwendeten Modellen um konkrete Java-Klassen. Damit die syntaktische Korrektheit der Modelle gewährleistet werden kann, wird das Eingabeformat auf bereits kompilierte Java-Klassen (*.class Dateien) beschränkt. Die Implementierung erweitert den `java.lang.ClassLoader` und benutzt `org.apache.commons.io.FileUtils` für eine vereinfachte Handhabung der Dateien. Quellcodebeispiel 7.1 zeigt das

Laden der `class` Datei und die Verarbeitung zu einem `Class`-Objekt.

```
1  public class FileClassLoader extends ClassLoader {
2
3    public Class<?> createClass(String pkgName, File file)
4       throws IOException{
5      byte[] bytes = FileUtils.readFileToByteArray(file);
6      String name =
7        FilenameUtils.removeExtension(file.getName());
8
9      //check if the class is already loaded
10     if(findLoadedClass(name) != null)
11       return findLoadedClass(pckgName + name);
12
13     //resolve the class
14     else{
15       Class<?> clazz = defineClass(name, bytes, 0, bytes.length);
16       resolveClass(clazz);
17       return clazz;
18     }
19   }
20 }
```

**Quellcode 7.1:** Implementierungsausschnitt - FileClassLoader

Der `FileClassLoader` bekommt dabei den optionalen Packagenamen der Klasse sowie ein Objekt des Typs `File` übergeben. Die Datei wird durch Verwendung eines `FileChoosers`, welcher die Selektierung bereits auf .class Dateien beschränkt, ausgewählt. Besonders auf eine Unterscheidung zwischen bereits geladenen und neu zu ladenden Klassen ist zu achten, da das doppelte Laden zu Fehlern führt. Die hierfür benötige Überprüfung ist in Zeile 10 zu sehen. Weiters ist darauf zu achten, dass das Laden Klassen mit Abhängigkeiten zu anderen Klassen zu Problemen führen kann. Diese werden in Abschnitt 7.4 gesondert betrachtet. Sollte dieser Fall eintreten, erhält der Benutzer eine aussagekräftige Fehlermeldung und wird über die zusätzliche benötigte Klassen informiert.

### 7.2.2 Erstellen von Testfällen

Da die Erstellung von Testfällen die zentrale Funktion der Applikation darstellt, hat diese eine immense Bedeutung für den Benutzer und muss sowohl vom Aufbau als auch bei der Ausführung eine intuitive Struktur aufweisen. Aus diesem Grund wurde eine übergeordnete Datenstruktur mit Namen Projekt ergänzt, die sowohl Modelle als auch Testfälle enthält. Abbildung 7.2 zeigt die Struktur eines Projekts mit hinzugefügtem Modell, Testdateien und Testfällen im *Model-* und *Testcase-Explorer*. Projekte dienen dabei lediglich der Kapselung zusammengehöriger Elemente und ermöglichen ein strukturierteres Arbeiten mit mehreren Modellen und Testfällen.

Das Anlegen und Hinzufügen sämtlicher Elemente erfolgt dabei über das Kontextmenü der Applikation. Die Eclipse-Plattform bietet die Einbettung eines Kontextmenüs durch das Hinzufügen von Erweiterungspunkten an. Hierzu werden im *MANIFEST.MF* im Menü *Extensions* die Erweiterungspunkte `org.eclipse.ui.menus`, `org.eclipse.ui.commands` und `org.eclipse.ui.handlers` benötigt. Der Erweiterungspunkt `menus` definiert hierbei die tatsächlichen Einträge inklusive Beschriftung und Icon innerhalb eines Menüs und verweist auf ein auszuführendes `command`. Jedem `command` ist dabei ein `handler` zugeordnet, der darüber entscheidet ob die zugehörige Aktion ausgeführt werden kann.

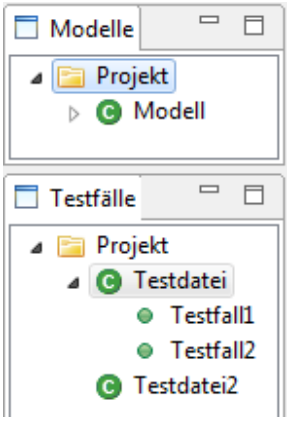
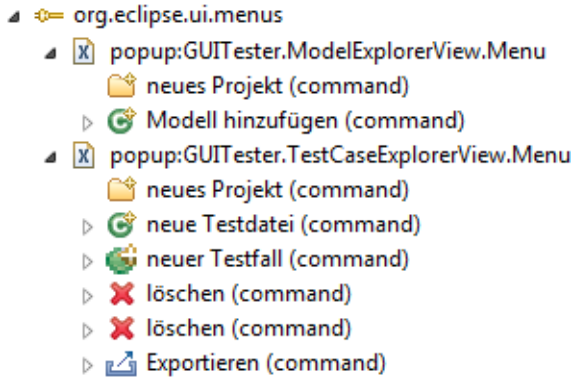


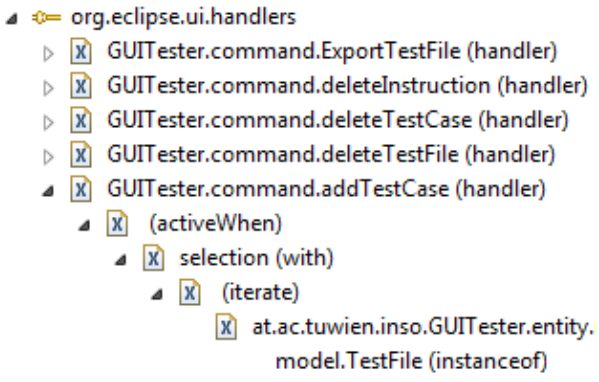

**Abbildung 7.2:** Struktur der Datenelemente

Abbildung 7.3 zeigt einen Ausschnitt der für die Implementierung des Kontextmenüs verwendeten Erweiterungspunkte. Dabei besitzen alle genannten Erweiterungspunkte einen *commandId*-Eintrag (7.3c), über den die Erweiterungspunkte aufeinander referenziert werden. Weiterhin sticht der Menüeintrag *löschen* besonders hervor, da dieser doppelt vorhanden ist. Obwohl beide Menüeinträge in Beschriftung und Icon übereinstimmen, werden im Hintergrund unterschiedliche *commands* ausgeführt. So ist es sowohl möglich, mit einem Meüeintrag Testfälle zu löschen, als auch mit einem anderen ganze Testdateien zu entfernen.

Figur 7.3b zeigt im weiteren Detail eine Regelung für das Aktivieren der Commands. In der dargestellten Abbildung wird zum Beispiel der Command `addTestCase` nur dann aktiviert, falls das selektierte Element eine Instanz der Klasse `TestFile` ist.




**(a)** Erweiterungspunkt menus




**(b)** Erweiterungspunkt handlers


**(c)** Verbindung der Erweiterungspunkte anhand der CommandId

**Abbildung 7.3:** Erweiterungspunkte für das Kontextmenü

### 7.2.3 Drag&Drop Unterstützung

Damit eine intuitive Benutzbarkeit gewährleistet werden kann, ist es dem Benutzer möglich, Methoden der Modelle aus dem Model-Explorer mittels Drag&Drop in den Editor eines Testfalls zu ziehen und dadurch ausgewählte Aktionen zum Testfall hinzuzufügen. Die Umsetzung erfolgt durch Implementierung der Interfaces `DragSourceListener` bzw. `DropTarget-Listener` aus dem Standard Widget Toolkit, wodurch das konkrete Verhalten des Listeners spezifiziert wird. Quellcodebeispiele 7.2 und 7.3 zeigen Ausschnitte der Drag&Drop Listener für die Listendarstellung.

```
1  public class InstructionDragListener implements DragSourceListener {
2
3    //check if the element is supported
4    public void dragStart(DragSourceEvent event) {
5      if (!(selection.getFirstElement() instanceof Instruction)) {
6        event.doit = false;
7      event.doit = true;
8    }
9
10   //set the data for this event
11   public void dragSetData(DragSourceEvent event) {
12     ...
13     event.data = instruction.getID();
14   }
15 }
```

**Quellcode 7.2:** Implementierungsausschnitt - DragSourceListener

Beispiel 7.2 zeigt einen Ausschnitt des `DragSourceListeners`. Hierbei wird in der Methode `dragStart` überprüft, ob eine Drag-Aktion gestartet wurde bzw. ob das ausgewählte Element für diese Aktion gültig ist. Die Methode `dragSetData` ist dafür zuständig, dem Drag-Event die benötigten Informationen zu setzen, in diesem Fall die ID des `Instruction` Objekts.

```
1  public class InstructionDropListener extends ViewerDropAdapter {
2
3      public void drop(DropTargetEvent event) {
4        location = this.determineLocation(event);
5        target = (Instruction) determineTarget(event);
6      }
7
8      public boolean performDrop(Object data) {
9        //perform the action with respect to location and target.
10     }
11 }
```

**Quellcode 7.3:** Implementierungsausschnitt - DropSourceListener

Beispiel 7.3 zeigt den `DropSoruceListener`. Dabei wird zuerst bestimmt ob die Drop-Aktion vor, nach oder exakt auf einem bereits eingefügten Element stattfindet und abhängig davon in der Methode `performDrop` das neue Element entsprechend hinzugefügt.

Für den Benutzer hat es den Anschein, als würde das vollständige Objekt mittels Drag&Drop verschoben werden, tatsächlich wird jedoch im Hintergrund nur ein spezifizierter Teil der Daten bewegt. Für die Implementierung ist dabei die Art der übertragenen Daten entscheidend, SWT bietet hierzu mittels Subklassen von `org.eclipse.swt.dnd.ByteArrayTransfer` vorgefertigte Implementierungen für verschiedene Datentypen. Diese können zwar nach Belieben erweitert werden, setzen allerdings die Konvertierung der Daten in ein Bytearray voraus. Eine solche

Konvertierung ist jedoch nur für serialisierbare Objekte möglich. Da die eingelesenen Methoden vom Typ `java.lang.reflections.Method` sind, ist die Serialisierbarkeit nicht gegeben. Die Drag&Drop Aktion überträgt deshalb nur die eindeutige Instruktions-ID in Form eines `org.eclipse.swt.dnd.TextTransfer`, welche ausreicht um das Objekt anschließend zu laden.

### 7.2.4 Darstellungsform der Testfälle

Wie bereits in Abschnitt 6.2.2 und Abbildung 6.10 gezeigt wurde, werden die Testfälle im Editor-Bereich sowohl in Form einer Liste als auch als Diagramm dargestellt. Abbildung 7.4 zeigt die Umsetzung dieser Listenform, Abbildung 7.5 die Darstellung eines Beispieltestfalls als Diagramm.

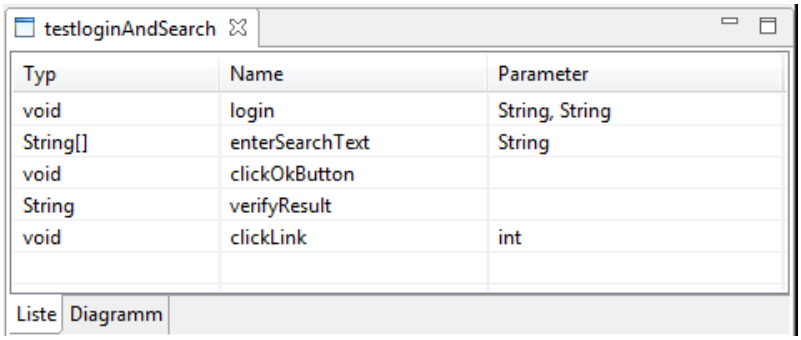



**Abbildung 7.4:** Testfall in Listendarstellung

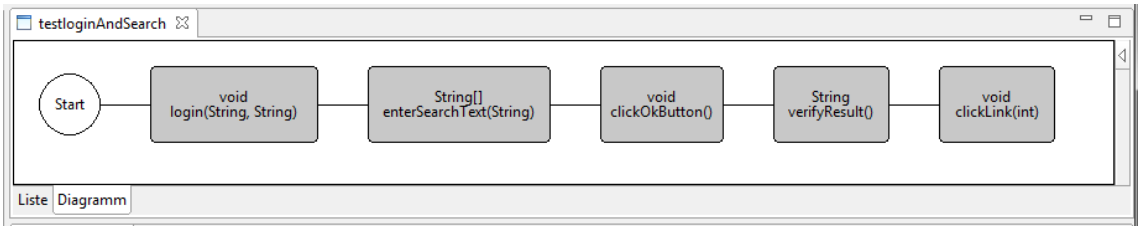



**Abbildung 7.5:** Testfall als Diagramm

Nachdem beide Darstellungsformen den selben Inhalt repräsentieren und somit eine logische Einheit darstellen, wurde an dieser Stelle ein `MultiPageEditor` verwendet, wodurch beide Darstellungsformen in einem Editor-Tab vereint werden können und die Konsistenz der Darstellung gewährleistet wird.

Während bei der Listendarstellung ein einfacher `TableViewer` zur Darstellung der `Instructions` ausreicht, gestaltet sich das Zeichnen derselben als Diagramm wesentlich aufwändiger. Die Implementierung der Diagrammdarstellung erfolgt mit Hilfe des in Abschnitt 4.3 vorgestellten Graphical Editing Frameworks. Im Rahmen des hier verwendeten MVC-Patterns erweitert GEF den für Editoren benötigten `EditorPart` um einen `GraphicalEditor`, welcher die komplette Darstellung im Editor übernimmt und somit den MVC-View darstellt. Zusätzlich wird durch eine `EditPartFactory` für jedes MVC-Modell eine entsprechender `EditPart` erzeugt. `EditParts` stellen den MVC-Controller dar und verbinden das Modell mit dem View, dabei entspricht die Struktur der `EditParts` der Struktur des Modells, wie Abbildung 7.6 verdeutlicht. Das Zeichnen der im View dargestellten Figur wird an dieser Stelle vom `EditPart` übernommen.

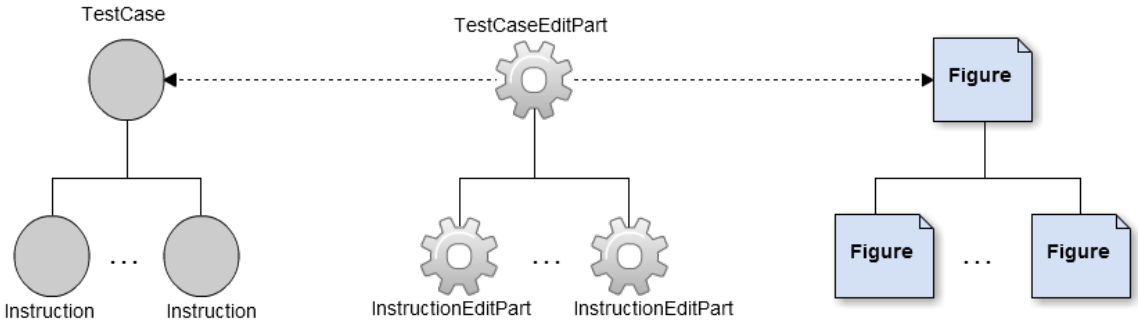



**Abbildung 7.6:** MVC Pattern für die Verwendung von GEF

Quellcodebeispiel 7.4 zeigt den `InstructionEditPart` und das Zeichnen einer `Instruction`. Die Methode `createFigure` definiert das Aussehen und die Form der Graphik, im dargestellten Beispiel ein Rechteck mit abgerundeten Ecken und Hintergrundsfarbe. Die Methode `refreshVisuals` ist für das tatsächliche Zeichnen der Figur verantwortlich. Hier wird abhängig von der dargestellten Instruktion sowohl die aktuelle Größe des Rechtecks, als auch der Inhalt Labels festgelegt.

```
1  public class InstructionEditPart extends AbstractGraphicalEditPart {
2
3    protected IFigure createFigure() {
4      IFigure rectangle = new RoundedRectangle();
5      rectangle.setBackgroundColor(new Color(null, 200, 200, 200));
6      ...
7      return rectangle;
8    }
9
10   //This is where the actual drawing is done
11   protected void refreshVisuals() {
12     Instruction instruction = (Instruction)getModel();
13     ...
14
15     //setting the size of the recangle
16     //width depends on the contained text
17     Rectangle bounds = new Rectangle(50, 50, width, 50);
18     getFigure().setBounds(bounds);
19
20     //adding the labels to the figure
21     Label typeLabel = new Label(instruction.getType());
22     typeLabel.setTextAlignment(PositionConstants.CENTER);
23     getFigure().add(typeLabel);
24     ...
25   }
26 }
```

**Quellcode 7.4:** Implementierungsausschnitt - InstructionEditPart

### 7.2.5 Bearbeiten von Testfällen

Eine Folge von ausgeführten Aktionen stellt zwar den Kern eines Testfalls dar, ist allerdings nicht ausreichend um diesen zur Gänze zu definieren. Neben dem Setzen von Parametern für einzelne Aktionen ist es notwendig, Annotationen, Annotationsparameter und andere Eigenschaften für Testfälle oder Instruktionen zu vergeben und diese aufeinander abzustimmen. Da die Anzahl der möglichen zusätzlichen Eigenschaften jedoch variiert und abhängig von Entität und verwendetem Testframework ist, wurde die Implementierung dieser Eigenschaften abseits der Darstellungs-

formen durch eine eigene Sicht umgesetzt. Dadurch entsteht eine klare Abgrenzung zwischen Testfall-Erstellung und Testfall-Bearbeitung, wodurch der Überblick zwischen den Elementen bewahrt wird.

Für die Implementierung dieser Funktion wurde ein `PropertySheet` verwendet, da diese Art von Sicht exakt die benötigten Anforderungen erfüllt. Ein `PropertySheet` stellt eine einfache Tabelle in der Form *Property - Value* dar und bietet somit einen strukturierten Überblick. Abbildung 7.7 zeigt einen selektierten Testfall mit der Eigenschaften-Sicht und den darin gesetzten Werten.

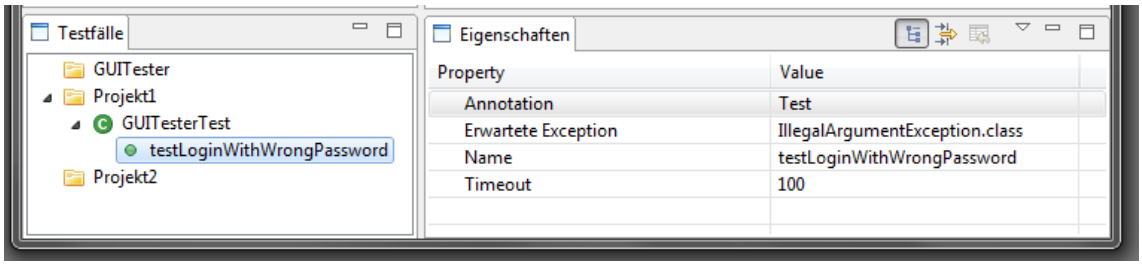



**Abbildung 7.7:** Eigenschaften-Sicht für selektierten Testfall

Wie in der dargestellten Abbildung ersichtlich, wurden für den Testfall *testLoginWithWrongPassword* die folgenden Parameter gesetzt: Annotation als `Test`, Erwartung einer `IllegalArgumentException` sowie ein Timeoutlimit von 100 Millisekunden. Quellcodebeispiel 7.5 zeigt den generierten Testfall mit den gesetzten Parametern.

```
1 @Test(timeout=100, expected=IllegalArgumentException.class)
2 public void testLoginWithWrongPassword() {
3   ...
4 }
```

**Quellcode 7.5:** Testfall mit gesetzten Annotationsparametern

Bei einem Testfall ist es demnach möglich, verschiedene Parameter für eine Annotation zu ergänzen sowie den Typ der gesetzten Annotation zu verändern. Des Weiteren unterstützen auch Testdateien und Instruktionen das Setzen zusätzlicher Eigenschaften über ein `PropertySheet`.

Bei Testdateien beschränken sich diese jedoch auf die Wahl des Testframeworks, für welches die Testfälle erstellt werden sollen. Instruktionen hingegen ermöglichen, wie Abbildung 7.8 verdeutlicht, die konkreten Werte für die Parameter einer Methode zu definieren.

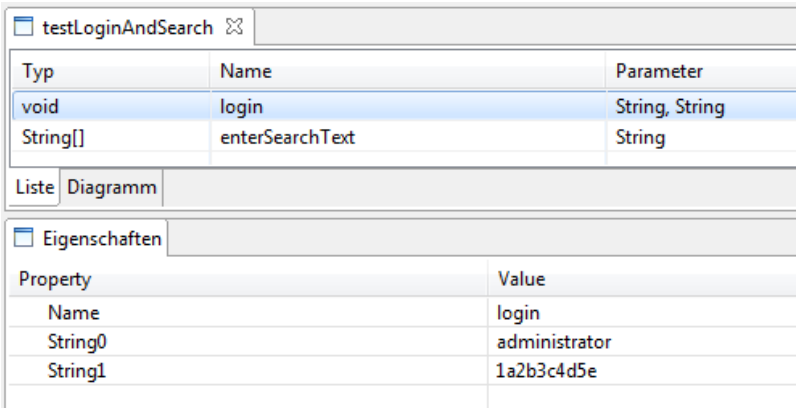



**Abbildung 7.8:** Eigenschaften-Sicht mit selektierter Instruktion

Die Implementierung der Einträge eines `PropertySheets` erfolgt durch Einbindung der Interfaces `IAdaptable` bzw. `IPropertySource`. Üblicherweise besitzt jede definierte Property eine eindeutige ID, über welche die zugehörigen Werte ausgelesen werden können. Eine typische Implementierung findet sich in Quellcodebeispiel 7.6.

```
1 public Object getPropertyValue(Object id) {
2   if(id.equals{Property_ID_Name})
3     return this.getName();
4   else if(id.equals{Property_ID_Value1})
5     return this.getValue1();
6   else if(id.equals{Property_ID_Value2})
7       return this.getValue2();
8   ...
9 }
```

**Quellcode 7.6:** Implementierungsausschnitt - Laden der Property-Werte

Zusätzlich müssen an dieser Stelle weitere Methoden des Interfaces `IPropertySource` implementiert werden, wie z.B. das Überprüfen, ob eine Property gesetzt ist (`isPropertySet(Object id)`) und das Setzen und Zurücksetzen eines Property-Werts (`resetPropertyValue(Object id)`, `setPropertyValue(Object id)`). Die Umsetzung mit der hier beschriebenen Methodik führt unweigerlich zu einem nicht sonderlich eleganten Quellcode, der sich an vielen Stellen wiederholt.

Um die Implementierung der Properties und deren Darstellung zu vereinfachen und den Quellcode eleganter und übersichtlicher zu gestalten, erfolgte diese mit Hilfe einer Erweiterung der RCP-Properties des Autors *Dan Phifer*[1]. Diesers stellt ein Set an Klassen für eine einfachere Integration der Properties zur Verfügung. Die jeweilige Klasse implementiert weiterhin das Interface `IAdaptable`, enthält als Adapter jedoch eine Instanz von *Dan Phifers* `PropertyManager`, wie in Quellcodebeispiel 7.7 dargestellt. Weiters existiert für unterschiedliche Eigenschaften entsprechende Klassen, die alle benötigen Methoden bereits implementieren.

```
1 public Object getAdapter(Class adapter) {
2   if(IPropertySource.class.equals(adapter))
3     return propertyManager;
4 }
```

**Quellcode 7.7:** Implementierungsausschnitt - Adapter bei Nutzung des PropertyManagers

Die wesentliche Vereinfachung besteht darin, dass jede Property als einzelnes Objekt zum `PropertyManager` hinzugefügt und nur mit Werten befüllt wird. Die zuvor durch Einbindung des Interfaces `IPropertySource` geforderten Methoden werden somit ausgelagert. Quellcodebeispiel 7.8 zeigt die benötigten Schritte, um ein editierbares Feld für den Namen der Entität zu den Properties hinzuzufügen.

```
1 //EditableProperty mit ID und Key-Name erstellen
2 EditableProperty nameProperty =
3   new EditableProperty("Property_Name_ID", "Name"));
4 nameProperty.setValue(name);
5 propertyManager.addProperty(nameProperty);
```

**Quellcode 7.8:** Implementierungsausschnitt - Properties mit dem PropertyManager

[1] siehe: http://www.eclipsezone.com/eclipse/forums/t53882.html

### 7.2.6 Export der Testfälle

Damit erstellte Testfälle ausgeführt und in einen bestehenden Testprozess integriert werden können, ist es notwendig, diese automatisiert als eigenständige Datei in einem syntaktisch korrekten Format zu generieren. Hierfür wird das in Abschnitt 6.1.2 definierte *Code Generator*-Projekt verwendet.

Als Grundlage des Projekts wird die Open-Source Bibliothek *CodeModel*[2] verwendet, welche Mechanismen zur Generierung von javakonformen Quellcode enthält. Zur Erhaltung der Eigenständigkeit der Projekte und zur Gewährleistung eines möglichst flexiblen Grads an Erweiterbarkeit und Austauschbarkeit wurde der Aufbau des *Code Generator*-Projekts unabhängig vom *RCP*-Projekt gestaltet. Der Aufbau wird im nachfolgenden Text näher ausgeführt.

Obwohl sich die Entitäten der beiden Projekte grundlegend überschneiden, wurden diese bewusst für jedes Projekt separat definiert. Dies erzielt zum einen eine lose Kopplung und sorgt weiters für Erweiterbarkeit, Datenkapselung und eine erhöhte Austauschbarkeit. Abbildung 7.9 zeigt die grundlegenden Entitäten dieses Projekts. Es lässt sich erkennen, dass Ähnlichkeiten in der definierten Struktur zum bereits vorgestellten RCP-Projekt bestehen.

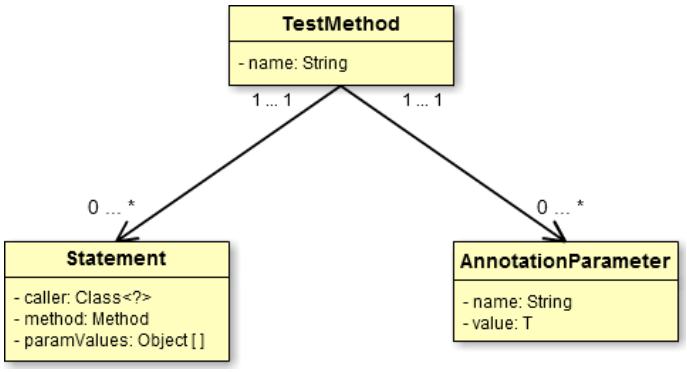



**Abbildung 7.9:** Klassendiagramm: Entitäten des *Code Generator* Projekts

Den einzelnen Klassen kommt dabei die folgende Bedeutung zu:

- `TestMethod`
  Stellt eine Methode der Java-Datei dar. Sie besteht aus mehreren Statements und kann eine oder mehrere Annotationen mit Annotationsparametern beinhalten. Diese Entität ist äquivalent zur `TestCase`-Entität des RCP-Projekts.

- `Statement`
  Stellt eine auszuführende Anweisung innerhalb einer Testmethode dar und ist äquivalent zur `Instruction`-Entität des RCP-Projekts.

- `AnnoationParameter`
  Enthält einen Parameter einer konkreten Annotation, zum Beispiel die Exception für Testfälle mit erwarteter Exception oder die Zeitspanne eines Timeoutlimits.

[2] siehe: https://codemodel.java.net

Die für das Generieren von Quellcode verantwortlichen Klassen wurden ähnlich der in Abbildung 7.10 dargestellten Struktur aufgebaut. Hier wurde besonders auf einen hohen Grad an Erweiterbarkeit geachtet, sodass sich die zukünftige Integration weiterer Testframeworks möglichst einfach gestaltet. Die abstrakte Klasse `AClassGenerator` stellt dabei die für die Generierung benötigten Funktionen bereit, während die Klassen `JUnitTestGenerator` bzw. `TestNGTestGenerator` lediglich frameworkspezifische Ergänzungen zu diesen Funktionen enthalten.

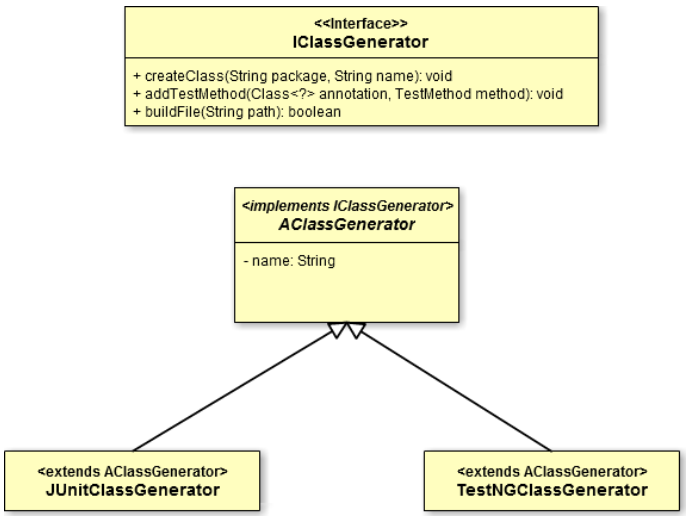



**Abbildung 7.10:** Klassendiagramm: Aufbau der Code-Generatoren

Der Schwerpunkt der Implementierung des *Code Generator*-Projekts liegt dabei in der Methode `addTestMethod`, da hier die Daten der vorhandenen Entitäten zusammengesetzt werden und eine syntaktisch korrekte Java-Datei generiert wird. Die Quellcodebeispiele 7.9 bis 7.11 zeigen Ausschnitte aus der Implementierung des Generators. Dabei wird besonders auf die im `AClassGenerator` enthaltenen Hilfsmethoden `generateMethod`, `getInvocation` und `addAnnotation` eingegangen.

```
1  //Here a simple invocation of a statement with parameters is
   ↪ generated and added to the class
2  private JInvocation getInvocation(Statement statement) {
3    //callerName references to the object which calls this method.
4    JInvocation invocation = JExpr.ref(statement.getCallerName())
5      .invoke(statement.getName());
6
7    //handle parameter
8    Object[] params = statement.getParamValues();
9    for(int i=0; i<params.length; i++) {
10     invocation.arg(JExpr.ref(params[i].toString()));
11   }
12   return invocation;
13 }
```

**Quellcode 7.9:** Implementierungsausschnitt - Hilfsmethode getInvocation

Die Methode `getInvocation` wird für jedes `Statement` einer `TestMethod` aufgerufen. Hier wird eine `JInvocation` für das übergebende `Statement` erzeugt und deren Parameterwerte, falls vorhanden, hinzugefügt. Die Hilfsmethode `addAnnotation` dient dazu, eine `Annotation` inklusive Annotationsparametern zu einer Methode hinzuzufügen.

Obwohl der momentanene Prototyp der Applikation ausschließlich auf testbezogene Annotationen ausgelegt ist, ist die Implementierung bereits generisch gehalten und würde somit auch andere Annotationen verarbeiten können.

```
protected void addAnnotation(JMethod method,
  Class<? extends Annotation> annotation,
  List<AnnotationParameter<?>> annotationParams){

    codemodel.ref(annotation);
    JAnnotationUse anno = method.annotate(annotation);

    //add parameter to the annotation
    if (annotationParams != null) {
      for (AnnotationParameter<?> p : annotationParams) {
        JAnnotationUseWrapper wrapper =
          new JAnnotationUseWrapper(anno);
        wrapper.addParameter(p.getName(), p.getValue());
    }
  }
}
```

**Quellcode 7.10:** Implementierungsausschnitt - Hilfsmethode addAnnotation

```
protected JMethod generateMethod(
  int modifier,
  String name,
  List<Statement> body,
  Class<? extends Annotation> annotation,
  List<AnnotationParameter<?>>> annotationParams) {

  //add the method to the clazz (CodeModels - JDefinedClass)
  JMethod method = clazz.method(modifier, codemodel.VOID, name);

  //add the annotation to the method
  addAnnotation(method, annotation, annotationParams);

  //generate the methods body
  JBlock block = method.body();

  for(Statement statement : body){
    //instantiates the caller class for this statement
    clazz.field(JMod.PRIVATE, statement.getClazz(),
      statement.getCallerName());

    //invoke the statements
    if(statement.getReturnType() != null &&
    !statement.getReturnType().equals(void.class)){

      //handle returnvalues
      JClass returnClazz = codemodel.
        directClass(statement.getReturnType().getSimpleName());

      JVar returnVal = block.decl(returnClazz,
        statement.getName().toLowerCase()+"_returnValue");

      JInvocation invocation = getInvocation(statement);
      block.assign(returnVal, invocation);
    }
    else{
      JInvocation invocation = getInvocation(statement);
      block.add(invocation);
    }
}
```

**Quellcode 7.11:** Implementierungsausschnitt - Quellcodegenerierung

Die Methode `generateMethod` generiert aus den übergebenen Parametern eine vollständige Java-Methode. Die übergebenen Parameter enthalten Informationen bezüglich des Namens der Methode sowie der gesetzten Modifier. Die Modifier werden in einer `int` Variable gespeichert, stellen jedoch eine Verkettung von `Bytes` dar und sind vor allem für das JUnit-Framework relevant, da hier sowohl die Modifier `private`, als auch `static private` benötigt werden. Weiters erhält die Methode die zu verwendende `Annotation` sowie Listen der `AnnotationParameter` und der ausführenden `Statements`. Ein Beispiel einer mit dem hier vorgestellten Projekt generierten Klasse findet sich in Anhang A.

Anzumerken ist, dass der in Quellcodebeispiel 7.11 abgebildete Ausschnitt der Implementierung lediglich eine prototypische Implementierung darstellt und aufgrund der hohen Komplexität des automatisiert generierten Quellcodes nicht vollständig ist. Näheres bezüglich der Problematik und Komplexität bei Codegeneratoren ist in Abschnitt 7.4 beschrieben.

### 7.2.7 Automatisches Einlesen von Modellen

Um dem Benutzer redundante und wiederkehrende Aktionen abzunehmen und die Arbeitsweise auf den Fokus der Applikation zu legen, stehen Mechanismen zum automatisierten Laden bereits erstellter Projekte und Modelle zur Verfügung. Hierfür wurde in der in *Anforderung F7* beschriebenen Konfigurationsdatei die Möglichkeit gegeben, durch eine direkte Pfadangabe einen Ordner mit den entsprechenden Inhalten anzugeben, welcher beim Programmstart durchsucht wird und dessen Inhalte verarbeitet werden.

Wesentlich sind dabei die Struktur des ausgewählten Ordners sowie die beinhalteten Dateien (siehe Abbildung 7.11). Der Inhalt des Ordners entspricht dem Aufbau der in der Applikation beschriebenen Struktur. Ein Ordner repräsentiert dabei ein Projekt, eine kompilierte Java-Klasse stellt ein einzulesendes Modell dar. Weiters befindet sich eine XML-Datei in jedem Ordner, die eine Reihenfolge für das Laden der Modelle definiert. Eine detailliertere Betrachtung des Ladens von Testdateien und Testfällen sowie der Notwendigkeit der spezifizierten Reihenfolge werden in Abschnitt 7.4 näher ausgeführt.

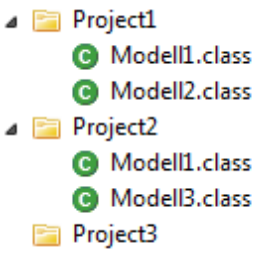



**Abbildung 7.11:** Struktur des Startup-Ordners

### 7.2.8 Hilfefunktion

In modernen Applikationen ist es üblich, dem Benutzer eine integrierte Hilfefunktion zur Verfügung zu stellen und so die Benutzbarkeit des Systems zu erhöhen. Der Benutzer wird durch die Verwendung des Benutzerhandbuchs sowohl in der Einarbeitung in das System als auch in der Handhabung der Funktionen desselben unterstützt.

Auch die Integration einer Hilfefunktion ist über die RCP-Plattform bereits definiert. Die Funktion ist dabei wie eine Webseite aufgebaut und wird aus mehreren HTML-Seiten zusammengesetzt, deren Struktur über den Erweiterungspunkt `org.eclipse.help.toc` definiert wird. Quellcodebeispiel 7.12 zeigt wie der genaue Aufbau der Webseite in Form einer XML-Datei durch den Erweiterungspunkt spezifiziert wird.

```
<toc label="GUITester User Guide" topic="toc.html">
  <topic label="Description" href="/description.html"/>
  <!-- section with subsections -->
  <topic label="Getting started"  href="/main.html">
    <topic label="Model-View" href="/modelview.html"/>
    ...
    <topic label="Editors" href="/editor.html">
      <!-- and subsubsections-->
      <topic label="List-Editor" href="/listeditor.html"/>
      <topic label="Diagram-Editor" href="/diagrameditor.html"/>
    </topic>
  </topic>
</toc>
```

**Quellcode 7.12:** Implementierungsausschnitt - toc.xml

Die Hilfefunktion wird standardmäßig von der Eclipse-Plattform als eigenständige Webapplikation bereitgestellt und verfügt bereits über eine Suchfunktion sowie Möglichkeiten zum Setzen von Lesezeichen. Ihre Verwendung in einer RCP-Applikation erfordert allerdings das Hinzufügen der Bibliotheken `org.eclipse.help.ui` , `org.eclipse.help.webapp` und `org.eclipse.equinox.http.jetty`. Beim Zugriff auf die Hilfefunktion wird im Hintergrund automatisch ein Jetty-Server gestartet, auf welchem die Webapplikation ausgeführt wird. Abbildung 7.12 zeigt das Benutzerhandbuch in der zur Verfügung gestellten Hilfefunktion.

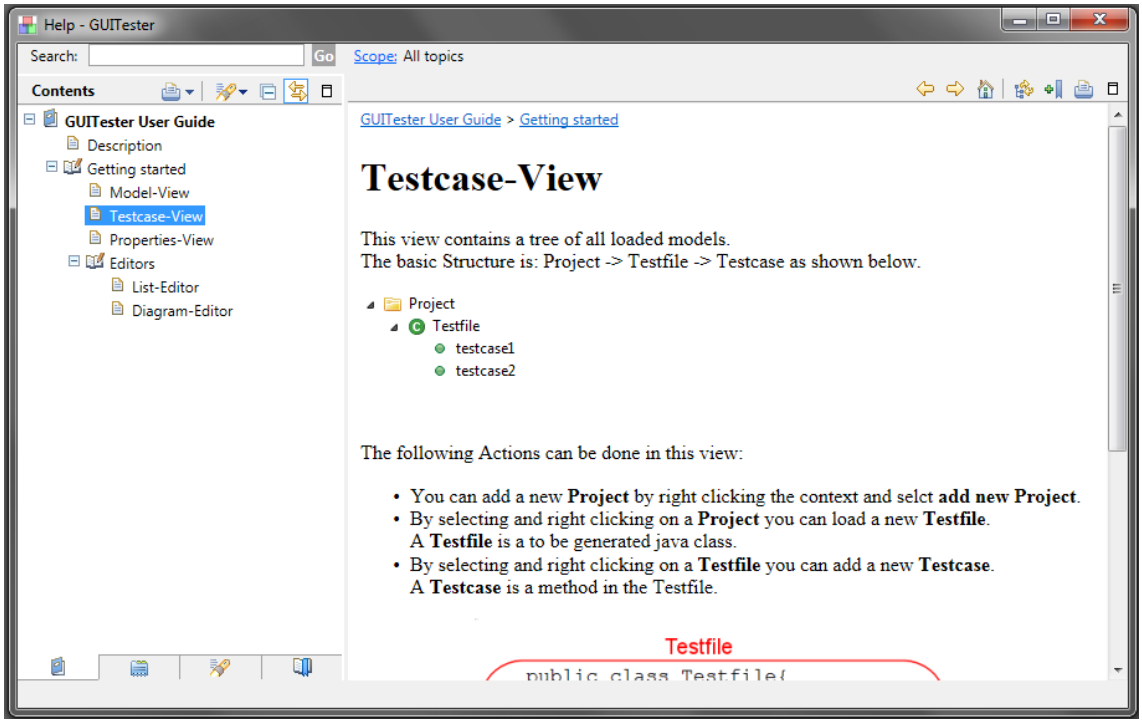



**Abbildung 7.12:** In die Hilfefunktion integriertes Benutzerhandbuch

### 7.2.9 Konfigurationsdatei

Damit grundlegende Konfigurationen sowie individuelle Anpassungen innerhalb der Applikation durch den Benutzer vorgenommen werden können, wurden die in Quellcodebeispiel 7.13 dargestellten Eigenschaften in eine Konfigurationsdatei ausgelagert. Diese legt fest, welche Dateiformate für das in *Anforderung F1* spezifizierte Einlesen der Modelle unterstützt werden und ist in einer *Key* = *Value*-Struktur aufgebaut. Weiters kann jeder Benutzer individuelle Anpassungen vornehmen, wie zum Beispiel die Definition des in *Anforderung F5* definierten *startup* Ordners, die Beschränkung der maximalen Spaltenbreite der Diagrammdarstellung oder individueller Icons für die Darstellung von *Projekten*, *Modellen*, *Dateien* und *Methoden*.

```
1  #possible file extensions for a model, separated by commas
2  dialog.addModelDialog.filter = *.class
3
4  #Directory for Models which are automatically loaded on startup
5  startup.load.model.directory =
6    C:\\Users\\Markus\\Desktop\\GUITester\\Startup
7
8  #number of columns for the diagram
9  diagram.column.count = 6
10
11 #Icons for ExplorerViews - TreeViewer
12 projectImage  = project.png
13 modelImage    = model.gif
14 testfileImage = model.gif
15 methodImage   = method.gif
```

**Quellcode 7.13:** Konfigurationsdatei

Zum Laden der Werte aus der Konfigurationsdatei stellt die Eclipse-Plattform in der Klasse `org.eclipse.osgi.util.NLS` geeignete Zugriffsmethoden bereit. Quellcodebeispiel 7.14 veranschaulicht diese Umsetzung. Wie in Zeile zwei zu sehen, zeigt das Codebeispiel deutlich, dass lediglich der relative Pfad der Properties-Datei angeführt werden muss. Das mit dem `NLS` erzeugte `RESOURCE_BUNDLE` ermöglicht anschließend den Zugriff auf die Einträge der Datei. Im Fehlerfall wird in der GUI der übergebene Key mit einem ? dargestellt sowie in einem Logfile der Zugriff auf eine nicht existente Property vermerkt.

```
1  public class ConfigLoader extends NLS {
2    private static final String BUNDLE_NAME = "//resources.config";
3    private static final ResourceBundle RESOURCE_BUNDLE
4      = ResourceBundle.getBundle(BUNDLE_NAME);
5
6    static {
7      NLS.initializeMessages(BUNDLE_NAME, ConfigLoader.class);
8    }
9
10   public static String loadPropertie(String key) {
11     try {
12       return RESOURCE_BUNDLE.getString(key);
13     } catch (MissingResourceException e) {
14       //log the error
15     }
16     return "?"+key+"?";
17   }
18 }
```

**Quellcode 7.14:** Implementierungsausschnitt - PropertyLoader

### 7.2.10 Mehrsprachigkeit

Um Einschränkungen in der Bedienung und sprachliche Barrieren zu verhindern, wurde für die Applikation die Internationalisierung durch Bereitstellung separater Sprachdateien vorgenommen. Weiters wurde eine Lokalisierung vorgenommen, wodurch das Hinzufügen weiterer Sprachdateien einfach und ohne Anpassungen am Quellcode umgesetzt werden kann. Hierfür wird derselbe Mechanismus benutzt, der auch bereits beim Laden der Konfigurationsdatei verwendet wurde und in Quellcodebeispiel 7.14 abgebildet ist. Die Quellcodebeispiele 7.15 und 7.16 veranschaulichen einen Ausschnitt aus der bereits bestehenden deutschen und englischen Sprachdatei, wobei jeweils in Zeile vier das Kürzel *{0}* auffällt, welches als Platzhalter für Parameter dient. Hier ist es also möglich, den Inhalt des Textes dynamisch anzupassen.

```
1 dialog.AddTestCase.label.name     = Name
2 dialog.AddTestCase.message.empty  = Name darf nicht leer sein
3 dialog.AddTestCase.message.exists =
4   ein Testfall mit dem Namen : {0} existiert in dieser Datei bereits
```

**Quellcode 7.15:** Implementierungsausschnitt - deutsche Sprachdatei

```
1 dialog.AddTestCase.label.name     = Name
2 dialog.AddTestCase.message.empty  = Name should not be empty
3 dialog.AddTestCase.message.exists =
4   Testcase with name: {0} already exists in this Testfile
```

**Quellcode 7.16:** Implementierungsausschnitt - englische Sprachdatei

Die Dateien sind dabei durch ein eindeutiges Sprachkürzel gekennzeichnet und werden anhand dieses Kürzels geladen. Die englische und deutsche Sprachdateien haben demnach den Dateinamen
*messages_en.properties* bzw. *messages_de.properties*. Für das Hinzufügen einer zusätzlichen Sprachdatei wird lediglich eine neue Datei mit den entsprechenden Inhalten benötigt, die das zugehörige Sprachkürzel trägt, beispielsweise *messages_fr.properties* für eine französische Übersetzung.

## 7.3 Umsetzung der nicht-funktionalen Anforderungen

Der nachfolgende Abschnitt gibt Aufschluss bezüglich der Erfüllung einiger in Tabelle 5.3 abgebildeten nicht-funktionalen Anforderungen und deren Umsetzung. Auf die Anforderung nach *Plattformunabhängigkeit* wird an dieser Stelle nicht ausführlicher eingegangen, da diese bereits durch die Benutzung der plattformunabhängigen Programmiersprache *Java* erreicht wird. Weiters wird die Anforderung der *Korrektheit* nicht näher diskutiert, da ein derartiger Beweis formaler Methoden bedürfte und ein solches Vorgehen den Rahmen dieser Arbeit sprengen würde und der Aufwand für einen Prototypen nicht angemessen scheint.

### 7.3.1 Modularität, Erweiterbarkeit & Wartbarkeitt

Die nicht-funktionale Anforderung *Modularität* wird durch die in Abschnitt 6.1.2 beschriebene Trennung der Architektur in Subprojekte und Module erfüllt und hat positive Auswirkungen auf die Erweiterbarkeit und Wartbarkeit des Systems.

Vor allem die bereits erwähnte Anwendung der Prinzipien Separation-of-Concerns und Datenkapselung unterstützt die Erfüllung der Anforderungen *Erweiterbarkeit* und *Wartbarkeit*. Damit eine

effiziente Weiterentwicklung und Wartung des Prototyps möglich ist, wurde auf die Verwendung von Entwurfsmustern sowie die Erstellung umfassender Dokumentation innerhalb des Quellcodes geachtet.

### 7.3.2 Benutzbarkeit

Die Struktur der Rich Client Platform (RCP)-Plattform und auch der in Abbildung 6.6 dargestellte Aufbau der GUI zeigen eine Übereinstimmung der Benutzeroberfläche mit der Eclipse-IDE. Die in den Abbildungen 6.5 und 6.6 hervorgehobenen farbigen Bereiche stimmen zum größten Teil sowohl im Aussehen als auch in ihrer Funktionalität überein und sorgen somit für einen möglichen Transfer des Wissens des Benutzers von der Eclipse-IDE auf die neu geschaffene Applikation. Durch diese grundlegende Übereinstimmung ist die Anforderung der Benutzbarkeit erfüllt, da ein besonders hoher Wiedererkennungswert besteht und für erfahrene Benutzer eine schnelle Einarbeitung und effiziente Verwendung gewährleistet wird.

### 7.3.3 Look&Feel

Die individuelle Anpassung des Systems, auch Branding genannt, zielt auf die Veränderung des Erscheinungsbilds der Applikation ab. Die Möglichkeit zur Erfüllung dieser nicht-funktionalen Eigenschaft wird bereits durch die RCP-Plattform bereitgestellt. Ein standardisiertes Look&Feel übernimmt dabei die grundlegende Aufgabe, die Benutzeroberfläche als native Applikation des zugrundeliegenden Betriebssystems darzustellen.

Die für Branding erforderlichen individuellen Texte wurden bereits in Abschnitt 7.2.10 erläutert. Die Einbettung individueller Icons wird in Abbildung 7.13 dargestellt und erfolgt über das `Manifest.mf` . Somit kann eine individuelle Anpassung des Systems erfolgen und die nicht-funktionale Anforderung *Look&Feel* ist erfüllt.

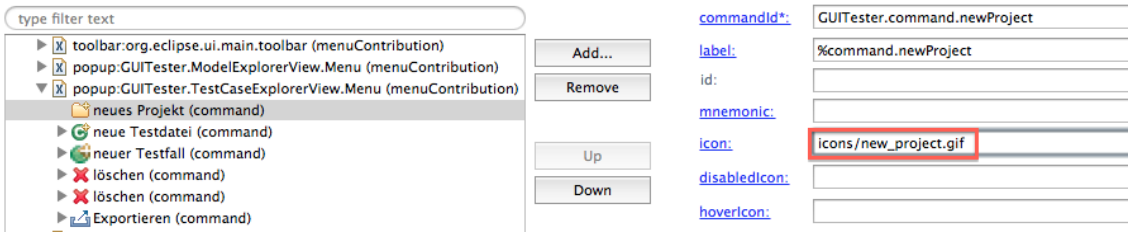
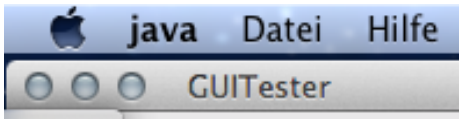



**Abbildung 7.13:** Individualisierung der Icons für die Menüeinträge

Abbildung 7.14 zeigt den ausgeführten Prototypen auf verschiedenen Betriebssystemen.




**(a)** Mac OSX


**(b)** Windows

**Abbildung 7.14:** natives Look&Feel

## 7.4 Besonderheiten und Probleme

Der nachfolgende Abschnitt geht näher auf potentielle Erweiterungen, Besonderheiten des Programmierens mit der RCP-Plattform sowie auf die im Zuge der Implementierung aufgetretenen Komplikationen und Schwierigkeiten ein und erläutert diese detailliert.

### 7.4.1 Verbesserte Modellinformationen

Eine mögliche Erweiterung wäre die Verbesserung der Modelle durch Kompilierung mit Debuginformationen. Während des Kompiliervorgangs einer Java-Datei gehen die für den Kompiler irrelevanten Informationen wie zum Beispiel Kommentare oder Parameternamen verloren. Diese Informationen würden allerdings eine sinnvolle Ergänzung in der Handhabung der Modelle darstellen. Durch das Kompilieren der Dateien mit Debuginformationen (`javac -g *.java`) kann gewährleistet werden, dass zumindest gesetzte Parameternamen in der *.class-Datei erhalten bleiben. Diese Informationen würden die Erstellung der Testfälle erheblich vereinfachen, da der Benutzer durch eindeutige Parameterbezeichnungen auf die Bedeutung des Parameters schließen könnte.

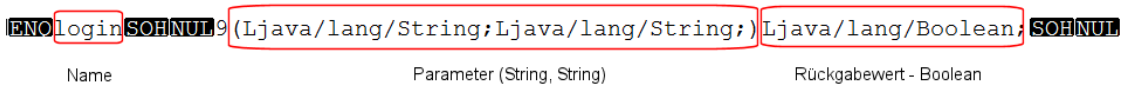
Die Abbildungen 7.15a und 7.15b veranschaulichen den Unterschied der Kompilierung inklusive und exklusive Debuginformationen und zeigen die kompilierte Form[3] der in 7.17 abgebildeten Methode.

```
1 public boolean login(String username, String password){
2   ...
3 }
```

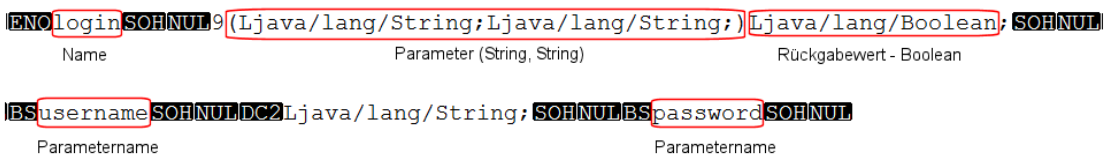
**Quellcode 7.17:** Simple Login Methode

Wie in den Abbildungen deutlich zu erkennen ist, verfügt die Variante mit inkludierten Debuginformationen über die Bezeichnungen der lokalen Variablennamen. Im Falle von Abbildung 7.15 lässt sich erkennen, dass beim ersten Parameter der Benutzername und beim zweiten das Passwort erwartet wird.

[3] Die hier dargestellten Klassen wurden in einem Texteditor geöffnet, wodurch die Darstellungsmöglichkeit eingeschränkt wird.




**(a)** exklusive Debuginformationen




**(b)** inklusive Debuginformationen

**Abbildung 7.15:** Kompilierte Login Methode

Als Alternative zur Kompilierung mit Debuginformationen besteht ab Java 8 eine neue Möglicheit des Zugriffs auf die Parameternamen zur Laufzeit. Wie in [27] beschrieben, wird hierfür eine Klasse `java.lang.reflect.Parameter` für die Parameterinformationen bereitgestellt und u.a. in die `Method` Klasse von `reflections` integriert. Allerdings müssen die Klassen durch den Befehl `javac -parameter` kompiliert werden. Die Verwendung dieser Neuerung hätte nur eine geringfügige Anpassung der Erstellung der `Instruction` zur Folge, da, wie in 7.1 beschrieben, an dieser Stelle `reflections` bereit benutzt wird.

### 7.4.2 Laden von Klassen mit Abhängigkeiten

Wie bereits in Abschnitt 7.2 erwähnt, kann das Laden von Klassen mit Abhängigkeiten zu Problemen führen. Wird eine solche Klasse als Modell eingelesen und wurde deren Abhängigkeit nicht bereits geladen, wird vom `ClassLoader` ein `java.lang.NoClassDefFoundError` geworfen.

Bei Subklassen von `java.lang.Error` handelt es sich im Normalfall um ernsthafte Probleme, die nicht durch ein `try-catch`-Konstrukt abgefangen werden sollten. Im Falle der hier vorgestellten Applikation, im speziellen Fall des Ladens einer bereits kompilierten Klasse im Zusammenhang mit einem `java.lang.NoClassDefFoundError` ist diese Norm allerdings vernachlässigbar.

Quellcodebeispiel 7.18 zeigt eine beispielhafte Klasse mit Abhängigkeit. Im dargestellten Beispiel besitzt die Klasse `A` eine Abhängigkeit zu Klasse `B`. Würde nun die Klasse `A` geladen und im Zuge ihrer Verarbeitung zu einem Modell mit `java.lang.reflections` auf deren Methoden zugegriffen, würde dies in einem `java.lang.NoClassDefFoundError` resultieren, da dem `ClassLoader` die Klasse `B` nicht bekannt ist.

```
1 class A {
2
3   public B getB() {
4     ...
5   }
6   ...
7 }
```

**Quellcode 7.18:** Beispielhafte Klasse `A` mit Abhängigkeit zu Klasse `B`

In diesem Fall ist es durch das Fangen des Errors und dessen Fehlertext möglich, dem Benutzer gezielt Informationen bezüglich der Abhängigkeiten und der dadurch benötigten Klassen zu liefern. Der Benutzer erhält konkrete Anweisungen und erkennt welche Klassen benötigt werden, um seine gewünschte Aktion durchzuführen.

Daraus ergibt sich unweigerlich eine fest definierte Reihenfolge für das Laden von Klassen und bedeutet zudem, dass möglicherweise zusätzliche Klassen aufgrund der Abhängigkeiten als Modell hinzugefügt werden müssen, auch wenn diese für die spätere Testfallgenerierung keinerlei Bedeutung haben. Für das Festlegen der Reihenfolge wurde, wie in Quellcodebeispiel 7.19 dargestellt, eine XML-Datei spezifiziert, in der sowohl die Reihenfolge, als auch der Packagename der Klasse definiert werden können.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes">
<models>
  <model package="" file="B.class"/>
  <model package="" file="A.class"/>
  ...
</models>
```

**Quellcode 7.19:** XML-Datei mit definierter Reihenfolge der zu ladenden Klassen

Voraussetzung für das automatisierte Laden von Modellen ist, dass eine solche XML-Datei vorliegt. Ist diese Datei nicht vorhanden, werden also auch keine Modelle eingelesen. Quellcodebeispiel 7.20 zeigt die Verarbeitung der Daten aus der XML-Datei durch Benutzen der Bibliotheken `org.w3c.dom` und `javax.xml`.

```
1  //parsing and normalizing the XML-File
2  DocumentBuilderFactory dbFactory =
3    DocumentBuilderFactory.newInstance();
4  DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
5  Document xml = dBuilder.parse(file);
6  xml.getDocumentElement().normalize();
7
8  //collect the information
9  NodeList nList = xml.getElementsByTagName("model");
10 for (int i = 0; i < nList.getLength(); i++) {
11   Node node = nList.item(i);
12
13   if (node.getNodeType() == Node.ELEMENT_NODE) {
14     Element element = (Element) node;
15     String path = FilenameUtils.getFullPath(file.getAbsolutePath()
16       + element.getAttribute("file")));
17
18       //check if file exists and load the model
19       addModel(parent, element.getAttribute("package"), path);
20   }
21 }
```

**Quellcode 7.20:** Implementierungsausschnitt - XML Parser

Dabei wird die eingelesene Datei durch den `javax.xml.DocumentBuilder` in ein Document Object Model (DOM)-Objekt umgewandelt und anschließend normalisiert. Wie in Zeile neun dargestellt, wird eine Liste aller Elemente mit den Namen *model* erstellt und über diese iteriert. Dabei werden die Attribute des Packagenamens und des Dateinamens ausgelesen und mit diesen Informationen das entsprechende Modell geladen.

### 7.4.3 Standardisierte GUI-Elemente

Wie bereits in Abschnitt 6.1.1 erwähnt, sind der Aufbau und die Struktur der Benutzeroberfläche bereits durch die RCP-Plattform definiert. Dies ist allerdings nicht nur auf die Bestandteile der GUI wie Sichten oder Editoren beschränkt, sondern schließt konkrete Elemente der Toolbar mit ein. So kann zum Beispiel eine *Quick Access* Leiste, welche einen schnellen Zugriff auf definierte Features und Funktionen über deren Namen ermöglicht, zur Applikation hinzugefügt werden. Diese Funktion bietet allerdings Zugriff auf weitere durch die RCP-Plattform standardisierte Elemente, welche kein Bestandteil der im Rahmen dieser Arbeit umgesetzten Applikation sind, und beeinträchtigt somit die Benutzbarkeit des Systems. Abbildung 7.16 zeigt die Toolbar der Applikation mit aktiver *Quick Access* Leiste.

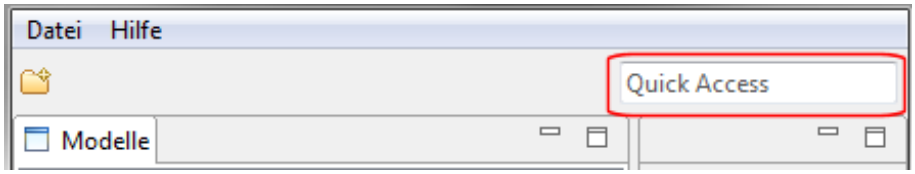



**Abbildung 7.16:** Benutzeroberfläche mit aktivierter Quick Access Leiste

Das Ausblenden der Quick Access Leiste ist in der aktuellen Version der RCP-Plattform allerdings nicht möglich, was den Entwicklern bereits als Bug[4] gemeldet wurde. Die Leiste kann dennoch durch Verwendung von Cascading Style Sheets (CSS) vor dem Benutzer verborgen werden. Der hierfür benötigte Quellcode ist in Quellcodebeispiel 7.21 dargestellt.

```
1 #SearchField {
2     visibility: hidden;
3 }
```

**Quellcode 7.21:** CSS zum Verstecken der Quick Acces Leiste

### 7.4.4 Speichern der letzten Session

Eine weitere Besonderheit der Eclipse-Plattform ist, dass der aktuelle Zustand der GUI beim Schließen einer Anwendung gespeichert und beim nächsten Öffnen der Applikation wiederhergestellt wird. Diese Besonderheit vermittelt zwar dem Benutzer das Gefühl der Kontrolle über die Benutzeroberfläche, kann während der Implementierung allerdings Verwirrung stiften. Wird zum Beispiel eine neu erstellte Sicht zur Applikation hinzugefügt, so wird diese nicht automatisch bei Neustart der Applikation dargestellt, auch wenn keinerlei Fehler im Programmcode vorliegen. Um eine aktuelle Version der Benutzeroberfläche zu erzeugen, ist es vor dem Starten der Applikation notwendig, den Workspace zu leeren.

Dieselbe Tatsache wirkt sich auf die Sprachdateien aus. Eclipse speichert bereits benutzte Sprachvariablen in einem lokalen Cache anstatt diese neu zu laden. Dies wirkt sich zwar positiv auf die Performanz der Applikation aus, bedeutet allerdings für den Entwickler, dass Änderungen in den Sprachdateien nicht neu geladen werden. Um die aktuellsten Sprachdaten zu laden, ist ebenfalls ein Entleeren der Konfigurationsdateien der RCP-Plattform notwendig.

[4] siehe: https://bugs.eclipse.org/bugs/show_bug.cgi?id=362420

### 7.4.5 Automatisierte Generierung von Quellcode

Die automatisierte Generierung von Quellcode stellt einen sehr umfangreichen und komplexen Themenbereich dar, da besonders auf die folgenden Aspekte zu achten ist:

- Die syntaktische Korrektheit muss gewährleistet sein.
- Die semantische Korrektheit muss gewährleistet sein.
- Die Wartbarkeit des generierten Codes muss gewährleistet sein.
- Der generierte Quellcode muss in lesbarer Form generiert werden.

Besonders im Hinblick auf generierte Testfälle und deren weiteren Einsatzzweck sowie künftige Adaptionen und Erweiterungen des Codes wird ein hohes Maß an Wartbarkeit benötigt, andernfalls würde die Generierung des Quellcodes die Wartung und Adaption der Testfälle erschweren und somit ihr Ziel nicht erfüllen. Die Komplexität der Generierung wird im Rahmen dieser Applikation zusätzlich durch die Tatsache verstärkt, dass sämtliche zu generierende Aktionen in der Benutzeroberfläche dargestellt und ausgewählt werden können.

## 7.5 Vergleich mit bestehenden Ansätzen

Dieser Abschnitt präsentiert bestehende Werkzeuge und vergleicht diese mit dem in dieser Arbeit vorgestellten Projekt.

### 7.5.1 Tosca Testsuite

Die Tosca Testsuite [55] ist ein Werkzeug für das Softwaretesten und wird unter einer kommerziellen Lizenz von der Firma Tricentis vermarktet. Die Testsuite bildet dabei ein Set an Werkzeugen für Testmanagement, Testfalldesign, Testdurchführung, Testdatengenerierung und Testautomatisierung und stellt gemäß [40] eine der meist verwendetsten kommerziellen Lösungen für Testautomatisierung dar. Getestet werden damit sowohl Webanwendungen, mobile Applikationen als auch Client Applikationen.

Die Tosca Testsuite setzt dabei sowohl auf einen modellbasierten Ansatz als auch auf eine graphische Benutzeroberfläche und intuitive Handhabung. Durch die Kombination dieser Konzepte ist es den Benutzern laut [40] möglich, auch ohne Expertenwissen schnell und einfach Testfälle zu spezifizieren und über die Testsuite zu verwalten, zu dokumentieren und auszuführen.

Im Vergleich mit dem in dieser Arbeit vorgestellten Projekt findet sich in der Tosca Testsuite ein ähnlicher Ansatz. Beide Projekte setzen auf eine einfach zu bedienende Darstellung über eine graphische Benutzeroberfläche und einen modellbasierten Ansatz für den Hintergrund. Ein wesentlicher Unterschied besteht allerdings in der Offenheit der Systeme. Während es sich bei der Tosca Testsuite um ein kommerzielles Produkt handelt und die Benutzung auf das Produkt selbst beschränkt werden soll, ist es in dem hier vorgestellten Projekt möglich sowohl Modelle als auch die generierten Testfälle in andere Projekte oder Werkzeuge zu integrieren und weiterzuverarbeiten.

### 7.5.2 Selenium

Selenium [45] ist ein Open-Source Capture-Replay Tool für Webapplikationen, mit dem es möglich ist, sowohl Testfälle als auch normale Arbeitsschritte mit einer Webanwendung aufzunehmen und diese zu automatisieren. Selenium ermöglicht es nicht nur Testfälle zu erstellen und diese zu automatisieren, sondern bietet auch den Export der Testfälle als JUnit oder TestNG Testfälle an und bietet somit eine Möglichkeit Tests von Webanwendungen in einen bestehenden Testprozess zu integrieren. Weiters bietet Selenium mittels einer API die Möglichkeit `Page Objects` testrelevante Bereiche einer Webseite zu modellieren und somit während des Tests den Zugriff auf Funktionen zu vereinfachen.

Im Vergleich lassen sich auch zwischen Selenium und der in dieser Arbeit vorgestellten Applikation Ähnlichkeiten erkennen. Trotz der Capture-Replay Eigenschaften von Selenium besteht die Möglichkeit, bereits erstellte Testfälle in gängigen Formaten zu exportieren bzw. zu generieren und diese unabhängig von Selenium in einen bestehenden Testprozess zu integrieren.

### 7.5.3 IBM Rational Functional Tester

Der Rational Functional Tester [23] ist ein Capture-Replay Werkzeug für automatisierte Tests von graphischen Benutzeroberflächen, der sowohl Webanwendungen als auch unterschiedliche Technologien für die GUIs von Rich-Client Applikationen unterstützt. Weiters können erfahrene Benutzer mit Programmierkenntnissen in beispielsweise Java die Testfälle adaptieren und verfeinern. Auch datengetriebenes Testen wird unterstützt. Bereits aufgezeichnete Testfälle können mit einem Set an verschiednen Testdaten automatisiert ausgeführt werden.

Im Vergleich zu der in dieser Arbeit vorgestellten Applikation lässt sich der bereits erwähnte Nachteil reiner Capture-Replay Tools erkennen. Auch wenn die Robustheit der Testfälle beim Rational Functional Tester durch natürliche Sprache und Screenshots erhöht werden soll, können Änderungen an der Benutzeroberfläche gravierende Auswirkungen auf die Testfälle nach sich ziehen. Weiters ist für gewisse Funktionen im Rational Functional Tester Expertenwissen in verschiedenen Programmiersprachen notwendig [23].

# 8 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein modellbasierter Ansatz zum Erstellen von Testfällen für Benutzeroberflächen vorgestellt und prototypisch implementiert. Zu Beginn wurde eine theoretische Einführung in das Gebiet des Softwaretestens und der Gestaltung von Benutzeroberflächen gegeben. Besonders wurde dabei auf die verschiedenen Realisierungsansätze wie Capture/Replay-Werkzeuge, skriptbasierte sowie modellbasierte Verfahren zur Testfallerstellung eingegangen. In diesem Zusammenhang wurde gezeigt, dass Capture/Replay-Werkzeuge für das Testen von Benutzeroberflächen nicht gut geeignet sind, da sich die Wartung der aufgezeichneten Testskripts als sehr zeitaufwändig und schwierig herausstellen kann. Zudem könnten Veränderungen am zugrundeliegenden System eine komplette Neuerstellung der Testfälle nach sich ziehen.

Ein modellbasierter Ansatz trägt zur Unabhängigkeit zwischen der Benutzeroberfläche und ihren Testfällen bei, da bei Änderungen der Benutzeroberfläche lediglich das Modell, nicht jedoch die Testfälle angepasst werden müssen. Somit verlagert sich die Abhängigkeit auf das Modell, woraus sich der Vorteil ergibt, dass bei Veränderungen der Benutzeroberfläche lediglich einige wenige Modelle angepasst werden müssen, um die Funktionalität aller darauf basierender Testfälle zu erhalten.

Ziel dieser Arbeit war es, unter Verwendung der von einem bereits bestehendem Projekt bereitgestellten Modelle eine prototypische Applikation zu entwerfen, welche die Spezifikation und anschließend die automatische Generierung von Testfällen über eine graphische Benutzeroberfläche unterstützt. Eine weitere Anforderung bestand darin, die Applikation über die Eclipse-RCP-Plattform zu implementieren. Im Rahmen der Projektplanung wurde neben einer Anforderungsanalyse ein detailliertes Architekturkonzept erstellt. Es wurde entschieden, die Applikation in zwei unabhängige Module zu gliedern, um dem Konzept des Separation-of-Concerns gerecht zu werden sowie einen hohen Grad an Austauschbarkeit zu erreichen. Das Hauptmodul beinhaltet die Implementierung der RCP-Plattform und stellt die Benutzeroberfläche sowie die benötigte Logik zum Einlesen der Modelle und zur Spezifikation der Abläufe für Testfälle bereit. Das zweite Modul, der Code Generator, übernimmt sämtliche Aufgaben für die Generierung von ausführbaren Testskripts aus den spezifizierten Testfällen. Zusätzlich wurden im Rahmen der Planung detaillierte Skizzen der geplanten Benutzeroberfläche erstellt, welche im Rahmen einer heuristischen Evaluierung auf die Übereinstimmung mit Designguidelines geprüft wurden.

Anschließend wurde mit der Implementierung der Applikation begonnen. Neben der Erfüllung der funktionalen Anforderungen wurde dabei besonders auf die Erweiterbarkeit und Austauschbarkeit der Module geachtet, um zukünftige Verbesserungen zu erleichtern. Die Applikation unterstützt im momentanen Stand des Prototypen die Testframeworks JUnit und TestNG und ermöglicht die Generierung von Testfällen für diese. Weiters wurde die Internationalisierung der Applikation mittels Lokalisierung von Sprachdateien gelöst sowie eine individuelle Anpassung des Look&Feels durch Konfigurationsdateien ermöglicht. Außerdem ist anzumerken, dass die automatisierte Generierung von Quellcode aufgrund ihres Umfangs und ihrer Komplexität nur prototypisch implementiert ist und sich auf einfache Anweisungen beschränkt.

Bei der im Rahmen dieser Arbeit vorgestellten Applikation handelt es sich lediglich um einen Prototypen, der die Vorteile eines modellbasierten Ansatzes für die Erstellung von Testfällen von Benutzeroberflächen untersucht. Eine komplette Implementierung aller Teilaspekte der Testfaller-

stellung hätte den Rahmen dieser Arbeit beträchtlich gesprengt. Um einen praxisnahen Einsatz zu ermöglichen und die Unterstützung der Testtätigkeiten möglichst vollständig zu gestalten, ist der Umfang des hier vorgestellten Prototyps noch nicht ausreichend.

Ein wichtiger Punkt für die Benutzbarkeit und den praktischen Einsatz wäre die Erweiterung des Funktionsumfangs, sodass auch komplexere Abläufe wie Schleifen oder Verzweigungen über die Benutzeroberfläche spezifiziert werden können. Die Unterstützung derartiger Konstrukte würde auch eine Modifikation des Code Generators erfordern. Die automatisierte Generierung von Quellcode hat sich im Rahmen der Implementierung allerdings als ein komplexes Themengebiet herausgestellt. Es ist also davon auszugehen, dass für seine Erweiterung ein nicht zu unterschätzender Aufwand erforderlich wäre.

Besonders im Hinblick auf die Benutzbarkeit wäre weiters das Speichern, Laden und Adaptieren von bereits erstellen Testfällen relevant. Eine Variante zur Implementierung dieses Features wäre die Integration der spezifizierten Testfälle und ihrer Struktur in das in Abschnitt 7.4.2 erwähnte XML-Dokument. Alternativ würde sich auch die Verwendung von JavaScript Object Notation (JSON) anbieten.

Für einen rentablen Einsatz in realen Projekten wäre weiters die Erweiterung der Eigenschaften-Sicht um zusätzliche Annotationen bzw. Annotationsparameter sowie die Möglichkeit für datengetriebenes Testen erforderlich. Auch die Eingabeunterstützung (zum Beispiel für die Auswahl von Exceptions) würde einen Beitrag zur Benutzerfreundlichkeit leisten.

Sobald ein erweiterter Prototyp vorliegt, sollte auch die Verifizierung der Praxistauglichkeit und Einsatzbereitschaft durch Anwendung des in Abschnitt 3.3 beschriebenen benutzerbasierten Designs angedacht werden. Der Prototyp sollte demnach einer Untersuchung durch erfahrene Tester unterzogen werden, um Feedback über das Design bzw. etwaige zusätzlich benötigte Funktionen oder bestehende Mängel zu erhalten.

# Literatur


[1] S. Arlt u. a. “Trends in Model-based GUI Testing”. In: *Advances in Computers* 86 (2013), S. 183 –222.

[2] P. Baker u. a. *Model-Driven Testing: Using the UML Testing Profile*. Springer, 2008.

[3] E. Barry und S. A. Slaughter. “Measuring Software Volatility: A Multi-dimensional Approach”. In: *Proceedings of the Twenty First International Conference on Information Systems*. Association for Information Systems, 2000, S. 412 –413.

[4] L. Bass, P. Clements und R. Kazman. *Software Architecture in Practice*. 2. Aufl. Addison-Wesley, 2003.

[5] C. Beust und H. Suleiman. *Next Generation Java Testing: TestNG and Advanced Conepts*. Addison-Wesley, 2008.

[6] Cédric Beust, Hrsg. *TestNG*. März 2013. URL: http://www.testng.org/doc (besucht am 28. 10. 2013).

[7] E. Börjesson und R. Feldt. “Automated System Testing Using Visual GUI Testing Tools: A Comparative Study in Industry”. In: *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*. 2012, S. 350 –359.

[8] T. Cleff. *Basiswissen Testen von Software*. w3l GmbH, 2010.

[9] S.R. Dalal u. a. “Model-based testing in practice”. In: *Proceedings of the 1999 International Conference on Software Engineering, 1999.* 1999, S. 285 –294.

[10] B. Daum. *Rich-Client-Entwicklung mit Eclipse 3.3: Anwendungen entwickeln mit Eclipse RCP, SWT, Forms, GEF, BIRT, JPA u.a.m.* 3. Aufl. dpunkt Verlag, 2008.

[11] E. Denert und J Siedersleben. *Software-Engineering: Methodische Projektabwicklung*. Springer, 1991.

[12] *Duden*. URL: http://www.duden.de/rechtschreibung/Volatilitaet (besucht am 31. 10. 2013).

[13] E. Dustin, J. Rashka und J. Paul. *Software automatisch testen*. Springer, 2001.

[14] M. Eberhard-Yom. *Usability als Erfolgsfaktor*. 1. Aufl. Cornelsen Verlag, 2010.

[15] R. Ebert. “Eclipse RCP: Entwicklung von Desktop-Anwendungen mit der Eclipse Rich Client Platform 3.7”. E-Book. 2011. URL: http://www.ralfebert.de/archive/eclipse_rcp/EclipseRCP.pdf (besucht am 31. 10. 2013).

[16] *Eclipse GEF Tutorial - GEF MVC*. URL: www.programcreek.com/2013/03/eclipse-gef-tutorial (besucht am 01. 02. 2014).

[17] K. Eilbrecht und G. Starke. *Patterns kompakt: Entwurfsmuster für effektive Software-Entwicklung*. 4. Aufl. Springer, 2013.

[18] *Fehlstart der Ariane 5 durch Software-Fehler*. Juli 1996. URL: http://www.welt.de/print-welt/article650740/Fehlstart-der-Ariane-5-durch-Software-Fehler.html (besucht am 10. 10. 2013).

[19] H. Freeman. “Software testing”. In: *IEEE Instrumentation & Measurement Magazine* 5.3 (2002), S. 48 –50.

[20] D. Graham u. a. *Foundations of Software Testing: ISTQB Certification*. Cengage Learning EMEA, 2008.

[21] T. Grechenig u. a. *Softwaretechnik: Mit Fallbeispielen aus realen Entwicklungsprojekten*. Pearson Studium, 2010.

[22] M. Hüttermann. *Agile Java-Entwicklung in der Praxis*. O'Reilly, 2007.

[23] *IBM Rational Functional Tester*. URL: http://www-03.ibm.com/software/products/de/functional (besucht am 27. 06. 2013).

[24] *ISO/IEC 9126: Software engineering - Product quality - Part 1: Quality mode*. International Organization of Standardization, 2001.

[25] *ISO/IEC 9241-11: Ergonomic requirements for office work with visual display terminals (VDTs)*. International Organization of Standardization, 1998.

[26] R. Jackson. "Testmanagement: Professionelles Testen". In: *Informatik Spektrum* 1.32 (2009), S. 37 –41.

[27] *Java 8: Access to Parameter Names at Runtime*. URL: http://openjdk.java.net/jeps/118 (besucht am 27. 06. 2013).

[28] *JUnit*. URL: http://junit.org (besucht am 24. 10. 2013).

[29] G. Kappel u. a. *UML@Work: Objektorientierte Modellierung mit UML 2*. 3. Aufl. dpunkt Verlag, 2005.

[30] A. Kornstaedt und E. Reiswich. "Staying afloat in an expanding sea of choices: emerging best practices for eclipse rich client platform development". In: *ACM/IEEE 32nd International Conference on Software Engineering, 2010*. Bd. 2. 2010, S. 59 –67.

[31] T. Künneth und Y. Wolf. *Einstieg in Eclipse 3.7: Aktuell zu Indigo und Java 7*. Galileo Computing, 2012.

[32] J. McAffer, J.M. Lemieux und C. Aniszczyk. *Eclipse Rich client Platform*. 2. Aufl. 2010, Pearson Education.

[33] B. Moore u. a. *Eclipse Development: using theGraphical Editing Framework and the Eclipse Modeling Framework*. 1. Aufl. International Business Machines Corporateion (IBM), 2004.

[34] G. J. Myers. *The Art of Software Testing*. 2. Aufl. John Wiley & Sons, Inc, 2004.

[35] J. Nielsen. *Usability Engineering*. Academic Press, 1994.

[36] J. Nielsen und R. Molich. "Improving a human-computer dialogue". In: *Communications of the ACM* 33.3 (1990), S. 338 –348.

[37] T. Ostrand u. a. "A Visual Test Development Environment for GUI Systems". In: *SIGSOFT Softw. Eng. Notes* 23.2 (1998), S. 82 –92.

[38] peterparkes. *Skype downtime today*. Dez. 2010. URL: http://blogs.skype.com/2010/12/22/skype-downtime-today/ (besucht am 10. 10. 2013).

[39] M. Pezzé und M. Young. *Software testen und analysieren: Prozesse, Prinzipien und Techniken*. Oldenbourg Verlag München, 2009.

[40] Pressebox. *TOSCA Testsuite ist eine der meist verwendeten automatisierten Softwaretest-Lösungen*. URL: http://www.pressebox.de/pressemitteilung/tricentis/TOSCA-Testsuite-ist-eine-der-meist-verwendeten-automatisierten-Softwaretest-Loesungen/boxid/565424 (besucht am 27. 06. 2013).

[41] A. Pretschner. “Model-based testing”. In: *27th International Conference on Software Engineering, 2005. ICSE 2005. Proceedings.* 2005, S. 722 –723.

[42] T. Roßner u. a. *Basiswissen modellbasierter Test*. Bd. 1. dpunkt Verlag, 2010.

[43] F. Sarodnick und H. Brau. *Methoden der Usability Evaluation: Wissenschaftliche Grundlagen und praktische Anwendung*. 2. Aufl. Verlag Hans Huber, 2011.

[44] R. Seidl, M. Baumgartner und T. Bucsics. *Basiswissen Testautomatisierung: Konzepte, Methoden und Techniken*. dpunkt Verlag, 2012.

[45] *Selenium - Web Browser Automation*. URL: docs.seleniumhq.org (besucht am 20. 04. 2014).

[46] B. Shneiderman. “Designing for fun: how can we design user interfaces to be more fun?” In: *interactions - Funology* 11.5 (2004), S. 48 –50.

[47] B. Shneiderman. *User Interface Design*. mitp-Verlag Bonn, 2002.

[48] B. Shneiderman und C. Plaisant. *Designing the user Interface*. 4. Aufl. Addison-Wesley, 2005.

[49] V. Silva. *Practical Eclipse Rich Client Plattform Projects*. Apress, 2009.

[50] H. M. Sneed, M. Baumgartner und R. Seidl. *Der Systemtest: Von den Anforderungen zum Qualitätsnachweis*. 3. Aufl. Carl Hanser Verlag München, 2012.

[51] A. Spillner und T. Linz. *Basiswissen Softwaretest*. 5. Aufl. dpunkt Verlag, 2012.

[52] Der Standard, Hrsg. *Software-Fehler legte Telefone und Handys in Ostösterreich lahm*. Nov. 2004. URL: http://derstandard.at/1855222 (besucht am 10. 10. 2013).

[53] T. Takala, Mika Katara und J. Harty. “Experiences of System-Level Model-Based GUI Testing of an Android Application”. In: *Fourth IEEE International Conference on Software Testing, Verification and Validation* (2011), S. 377 –386.

[54] The Eclipse Foundation, Hrsg. *Eclipse*. URL: http://www.eclipse.org (besucht am 11. 12. 2013).

[55] *Tosca Testsuite*. URL: http://tricentis.com (besucht am 27. 06. 2013).

[56] M. Utting und B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Elsevier, 2007.

[57] X. Zhang, J. Windsor und R. Pavur. “Determinants of Software Volatility: A Field Study”. In: *Journal of Software Maintenance* 15.3 (2003), S. 191 –204.

# A Anhang

Dieser Abschnitt zeigt den Workflow der Applikation anhand eines Beispiels. Dabei werden alle für einen Benutzer relevanten Arbeitsschritte dargestellt und erläutert.

- Erstellung eines Modells

  Um mit der vorgestellten Applikation zu interagieren, werden Modelle einer Benutzeroberfläche in Form von Java-Klassen benötigt. Das nachfolgend abgebildete Modell stellt die Verbindung zu der in Abbildung A.1 gezeigten Webseite dar. Dabei stellt `SearchPage` die initiale Seite dar und `ResultPage` die Webseite nach betätigter Suche.

```
1  /**
2  * Model of the Search Page
3  */
4  @ModelDesc(name="SearchPage")
5  public class SearchPage extends WebPage<Driver>
6  {
7    @Field(id="searchField", locator="ByName||q")
8    private TextField searchField;
9
10   @Field(id="submit", locator="ByName||btnG",
11     navigateTo = ResultPage.class)
12   private Button searchButton;
13
14   public ResultPage search(String text) throws DriverException
15   {
16     searchField.setValue(text);
17     return (ResultPage) searchButton.click();
18   }
19
20   @Override
21   public void open(){
22     super.open();
23   }
24
25   @Override
26   public void close(){
27     super.close();
28   }
29 }
```

**Quellcode A.1:** Modell - Suchseite

  Das Modell enthält Objektrepräsentationen der Elemente der Webseite sowie Methoden, die deren Zugriff steuern. Zusätzlich enthält das Modell einige Methoden der Elternklasse `WebPage`, wie zum Beispiel die Methoden `open` und `close`, zum Öffnen und Schließen der Webseite.

  Das Modell der `ResultPage` weist zusätzlich zu den in Quellcodebeispiel A.1 dargestellten Methoden noch die in Quellcodebeispiel A.2 dargestellten Erweiterungen auf.


**Abbildung A.1:** Darstellung der Webseite, des in Quellcodebeispiel 6.1 abgebildeten Modells.

```
1  public String getSearchValue() throws DriverException{
2    return searchField.getValue();
3  }
```

**Quellcode A.2:** Unterschiede der ResultPage zur SearchPage

- Laden des Modells und Definieren eines Testfalls

  Die in der nachfolgenden Auflistung beschriebenen und in Abbildung A.2 dargestellten Arbeitsschritte sind notwendig, um aus einem Modell einen Testfall zu generieren.

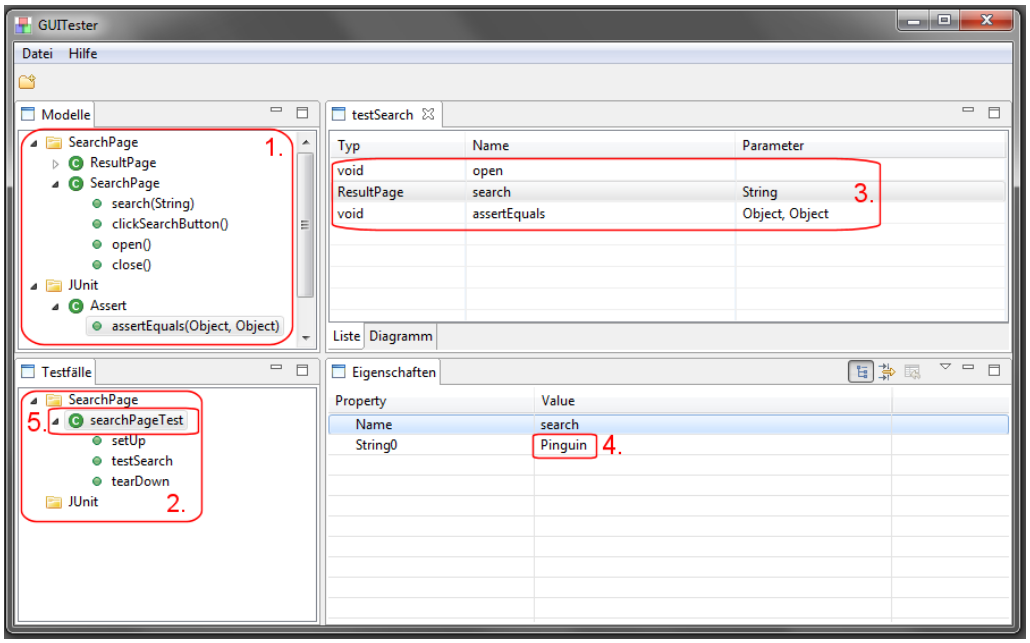



**Abbildung A.2:** Ablauf der Benutzung

1. Projekt erstellen und Modell laden
   Der erste Schritt besteht darin, durch Benutzung des Kontextmenüs ein *Projekt* zu erstellen und die benötigten *Modelle* zu laden. Dieser Arbeitsschritt findet ausschließlich im *Model-Explorer* statt. Im hier dargestellten Beispiel sind sowohl die Modelle der Webseite (`SearchPage` und `ResultPage`), als auch mehrere abhängige Modelle des Testframeworks JUnit (`Assert`, ...) geladen.

2. Testdatei und Testfälle erstellen
   Im nächsten Arbeitsschritt werden im *Testcase-Explorer* eine *Testdatei* sowie *Testfälle* angelegt.

   2.1. Eigenschaften der Testdatei und Testfälle anpassen
   Weiters werden im *Properties-View* die Eigenschaften bezüglich des verwendeten Testframeworks und der benötigten Annotation ausgewählt. Abbildung A.3 verdeutlicht dieses Vorgehen.

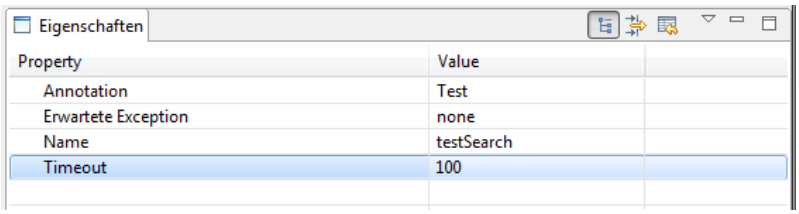



**Abbildung A.3:** Eigenschaften eines Testfalls

   Die Abbildung zeigt die Eigenschaften-Sicht für den Testfall `testSearch`. Hier wurde spezifiziert, dass dieser die Annotation `@Test` erhält und die maximale Dauer von 100 Millisekunden nicht überschreiten darf.

3. Testfall definieren
   Durch Drag&Drop-Aktionen werden ausgewählte Methoden aus dem im *Model-Explorer* ausgewählten *Modell* in den *Editor* gezogen und somit zum Testfall hinzugefügt. Dadurch wird die interne Abfolge des Testfalls festgelegt.

4. Eigenschaften des Testfalls anpassen
   Erneut werden im *Properties-View* die Eigenschaften der zum Testfall hinzugefügten Aktionen definiert. Abbildung A.2 zeigt den für die Methode `search` gesetzten Parameter *"Pinguin"*.

5. Testdatei exportieren
   Sind die Schritte 1-4 zur Zufriedenheit des Benutzers abgeschlossen, kann über das Kontextmenü des *Testcase-Explorers* die ausgewählte Testdatei als Java-Datei generiert und exportiert werden.

- Generieren und Exportieren des erstellten Testfalls

  Die in Quellcodebeispiel A.3 dargestellte Klasse ist das generierte Resultat der zuvor beschriebenen Arbeitsschritte unter Verwendung des JUnit-Frameworks. Die Methode `setUp` wurde in der GUI zwar erstellt, es wurden jedoch keine Anweisungen hinzugefügt, was in der Erstellung einer leeren Testmethode resultiert.

```
1  public class SearchPageTest{
2
3    private SearchPage searchPage;
4
5    @Before
6    public void setUp(){
7    }
8
9    @Test(timeout=100)
10   public void testSearch(){
11     searchPage.open();
12     ResultPage searchPage_returnValue =
13       searchPage.search("Pinguin");
14     Assert.assertEquals("Pinguin",
↪  searchPage.getSearchValue());
15   }
16
17   @After
18   public void tearDown(){
19     searchPage.close();
20   }
21 }
```

**Quellcode A.3:** Generierte Java-Testklasse