Otto-von-Guericke-Universität Magdeburg

Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme

Bachelorarbeit

Plattformunabhängigkeit grafischer Benutzeroberflächen - Analyse aktueller Frameworktechniken am Beispiel des Standard Widget Toolkit

Verfasser:

Gordon Damerau

28. April 2013

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake,
Dr.-Ing. Thomas Leich,
Dipl.-Inform. Matthias Ritter

Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik
Postfach 4120, D–39016 Magdeburg

Damerau, Gordon:
Plattformunabhängigkeit grafischer Benutzeroberflächen - Analyse aktueller Frameworktechniken am Beispiel des Standard Widget Toolkit
Bachelorarbeit, Otto-von-Guericke-Universität Magdeburg, 2013.

Danksagung

An dieser Stelle möchte ich mich bei meiner Familie und allen Personen bedanken, die mich bei meiner Bachelorarbeit und während meines Studiums unterstützt haben.

Ein besonderer Dank geht an meinen Prüfer Prof. Dr. rer. nat. habil. Gunter Saake und meine Betreuer Dr.-Ing. Thomas Leich und Dipl.-Inform. Matthias Ritter für die zahlreichen Anregungen und Tipps.

Weiterhin möchte ich mich bei den Mitarbeitern der Quinsol AG bedanken, die mich bei Entwurf und Umsetzung der Bachelorarbeit sowie der vorgestellten Komponenten unterstützt haben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

AWT . . . . . Abstract Window Toolkit

IFC . . . . . Internet Foundation Classes

JDK . . . . . Java Development Kit

JFC . . . . . Java Foundation Classes

JVM . . . . . Java Virtual Machine

MVC . . . . . Model-View-Controller

OLE . . . . . Object Linking and Embedding

PLAF . . . . . Pluggable Look and Feel

RCP . . . . . Rich-Client-Platform

SWT . . . . . Standard Widget Toolkit

Listings

Kapitel 1

1 Einleitung

Softwareprojekte für die breite Masse werden heutzutage fast ausschließlich mit einer grafischen Oberfläche (GUI: Graphical User Interface) ausgeliefert. Diese bestehen aus arrangierten Objekten wie Schaltflächen oder Registerreitern, die auf Analogien aus dem realen Leben zurückgreifen [Memon et al., 2001]. War es in den Anfängen nur Leuten vom Fach möglich, Computer mittels Eingabe von Kommandozeilencodes zu bedienen, werden jetzt optisch aufwändig gestaltete, aber dennoch funktionale und vor allem bedienerfreundliche Eingabemöglichkeiten erwartet. In modernen Anwendungen kann der Anteil des Quelltextes für eine grafische Oberfläche bis zu 60 Prozent des gesamten Quelltextes der kompletten Software betragen [Memon, 2002]. Bei den traditionellen Programmen ohne GUI bestimmte der Entwickler den eigentlichen Programmfluss und damit auch die Sequenz der Ein- und Ausgaben unter anderem mittels Tastatur und Maus. Bei grafischen Oberflächen ist dies anders. Diese sind nicht sequenziell festgeschrieben, sondern ereignisgesteuert aufgebaut. Klickt ein Benutzer zum Beispiel auf eine Schaltfläche, wird im Hintergrund ein vom Programmierer zuvor definierter Ablauf ausgelöst. Die Sequenz der Benutzereingaben durch den Anwender bestimmt demnach den eigentlichen Programmablauf. Die Umsetzung grafischer Oberflächen ist heutzutage ein wichtiger Schritt in der Erstellung von Softwareprodukten [Bishop, 2006]. Jan Borchers beschreibt die Sicht eines Benutzers auf ein Programm folgendermaßen:

> *As we know, users generally identify a software system with what its user interface shows them, without caring about its inner works.*
>
> *[Borchers, 2000]*

Demnach identifiziert ein Endanwender ein Software-System in erster Linie durch die Benutzeroberfläche, ohne sonderlich auf dessen Funktionalität zu achten.

Sun Microsystems, die Firma, die die Programmiersprache Java entwickelt hat und durch die Java die heutige Popularität erreichte, pries diese oft mit dem

Slogan „Write once, run anywhere!“ an [Kramer, 1996]. Damit wollte Sun Microsystems die Plattformunabhängigkeit, die Java von Haus aus bietet, betonen. Programme, die in Java geschrieben sind, haben den Vorteil, dass sie auf allen Plattformen lauffähig sind, auf denen auch eine Java Virtual Machine (JVM) lauffähig ist. Diese Grundidee wurde über die Jahre in viele andere IT-Bereiche übertragen (siehe zum Beispiel [Blom et al., 2008]).

Problemstellung

Um in Java mit relativ wenig Aufwand auch aufwändige grafische Oberflächen erstellen zu können, haben sich unter anderem Swing und SWT als potentielle Frameworks herauskristallisiert. Das Standard Widget Toolkit (SWT), welches sich die nativen Implementierungen der jeweiligen Plattform zu Nutze macht, deckt in Kombination mit JFace, einen sehr großen Teil dieses Entwicklungs-Stranges ab. SWT wird in einigen populären Softwareprojekten und Frameworks verwendet. So bindet zum Beispiel die Rich-Client-Platform (RCP) der Eclipse Foundation SWT zur Oberflächengestaltung ein [Geer, 2005].

Allerdings zeigt sich bereits bei kleineren Projekten, die auf unterschiedlichen Plattformen wie Microsoft Windows und Linux getestet werden, dass SWT dem Slogan der Java-Entwickler „Write once, run anywhere“ nicht immer folgt. Insbesondere bei Ereignissen und den grafischen Oberflächen ist eine Plattformunabhängigkeit nur sehr schwer zu realisieren. Um die einzelnen Missstände aufdecken und beheben zu können, benötigt es neben Tools zur Identifikation betroffener Bereiche auch Methoden, um derartige Diskrepanzen zu beheben.

Zielstellung und Gliederung der Arbeit

In dieser Arbeit soll auf die Umsetzung der Plattformunabhängigkeit von grafischen Oberflächen mittels Java in Kombination mit dem SWT eingegangen werden. Hierfür wird in Kapitel 2 der aktuelle Stand der eigentlichen Umsetzung erläutert, um gegebenenfalls existierende Abweichungen in den SWT-Bibliotheken zu identifizieren. Ein Vergleich der beiden am weitesten verbreiteten Frameworks, SWT als lightweight Umsetzung und dem gegenüber AWT/Swing als heavyweight Pendant, wird herausgearbeitet.

In Kapitel 3 wird genauer auf die Umsetzung des Frameworks SWT eingegangen. Dabei soll gezeigt werden, inwieweit die unterschiedlichen Bibliotheken die Plattformunabhängigkeit von Java unterstützen.

Anschließend werden in Kapitel 4 und Kapitel 5 geeignete Methoden vorgebracht, erläutert und in ihrer Umsetzbarkeit verglichen, mit denen die vorhandenen Differenzen in den SWT-Fassungen für die ausgewählten Plattformen behoben werden können. Die einzelnen Methoden werden in ihrem jeweiligen Anwendungsumfang evaluiert. Neben sprachinternen Mitteln wird auch auf zusätzliche Paradigmen, wie zum Beispiel der aspektorientierten Programmierung, Bezug genommen. An einem Fallbeispiel werden die Ideen und Konzepte evaluiert. Dieses

bezieht sich auf Softwareprodukte, die auf unterschiedlichen Plattformen (mindestens Microsoft Windows, Linux und Apple MacOS) umgesetzt werden und unter Umständen unterschiedliche Anforderungen an Komponenten aufweisen können. Als Basis zur Evaluierung wird ein Rich Client verwendet, der neben Paketen zur Datenverarbeitung auch Pakete für die grafische Oberfläche erfordert, beziehungsweise mitliefert. Ein konkretes Beispiel hierfür ist die RCP der Eclipse Foundation, welche SWT zur Darstellung voraussetzt.

In einer abschließenden Gegenüberstellung werden die vorgestellten Methoden in Kapitel 6 nochmals auf ihren spezifischen Anwendungsumfang untersucht.

Kapitel 2

2 Grundlagen grafischer Oberflächen in Java

In dem nachfolgenden Kapitel werden die Grundlagen für die Oberflächenentwicklung mit der Programmiersprache Java erläutert. Es wird zunächst in den Abschnitten 2.2, 2.3 und 2.4 darauf eingegangen, welche Bibliotheken sich im Laufe der Zeit für diesen Bereich der Entwicklung herauskristallisiert haben. In den einzelnen Passagen wird dabei genauer auf die jeweiligen Umsetzungen eingegangen, sowie Vor- und Nachteile herausgearbeitet. In einem abschließenden Vergleich in Abschnitt 2.5 der beiden populärsten Bibliotheken werden diese nochmals dargestellt.

2.1 Basisanforderungen

Eines der Hauptziele der Programmiersprache Java ist es, ein Framework für plattformunabhängige Software anzubieten. Da in der heutigen Zeit allerdings reine Konsolenprogramme in kommerziellen Produkten mehr oder weniger der Vergangenheit angehören, muss Java eine Möglichkeit bieten, grafische Oberflächen gestalten und verarbeiten zu können. Für eine sinnvolle Verwendung müssen diese Bibliotheken mindestens folgende Funktionen erfüllen:

- Zeichnen von Basiselementen
  Das Zeichnen von Elementen wie (Halb-) Kreise, Linien und Schriftzeichen und das jeweilige Ändern der Farben dieser Elemente ist die Grundlage zum Erstellen von höheren grafischen Benutzerschnittstellen.

- Widgets
  Das Erzeugen und Anzeigen von Widgets, also von grafischen Elementen zur Benutzereingabe und -ausgabe. Beispiele hierfür sind Schaltflächen, Textfelder oder Tabellen.

- Ereignisbehandlung
  Das Behandeln von Ereignissen wie zum Beispiel Maus und Tastatureingaben. Fährt der Benutzer mit der Maus über einen Menüeintrag oder klickt

auf eine Taste, so erwartet der Benutzer abhängig von der getätigten Aktion eine Reaktion der entsprechenden Komponente. Dafür muss die Bibliothek Modelle anbieten um derartige Ereignisse behandeln zu können.

Die Definition des Begriffs „Komponente“ in der Literatur variiert. So definiert Markus Völter diesen folgendermaßen:

> *Eine Komponente ist ein funktional abgeschlossener Softwarebaustein. Sie definiert die Dienste die sie anbietet sowie die Ressourcen die sie zur Erbringung der Dienste benötigt.*
>
> *[Völter, 2005]*

Eine andere Definition von Harro von Lavergne lautet:

> *Als Komponente wird [...] ein Kollektiv erzeugender Muster bezeichnet, das eine in sich geschlossene, im Rahmen eines entsprechenden Grundsystems funktionsfähige Einheit darstellt.*
>
> *[von Lavergne, 2004]*

Allen gemein ist der Grundbegriff der *Abgeschlossenheit*. In einer Javaumgebung sollten zudem sämtliche Komponenten auf allen relevanten Betriebssystemen vorhanden sein und eine vergleichbare Funktionalität bieten. Mit relevanten Betriebssystemen sollen hier Betriebssysteme gemeint sein, auf denen eine JVM lauffähig ist, darunter Microsoft Windows, Linux und Apple MacOS. Nachfolgend werden drei mögliche Frameworks vorgestellt, die die genannten grundlegenden Punkte zur Erstellung grafischer Benutzeroberflächen mehr oder weniger gut umsetzen.

2.2 Abstract Window Toolkit (AWT)

Bereits in der ersten Version aus dem Jahre 1995 enthielt Java das Abstract Window Toolkit (AWT), welches als rudimentäres Toolkit zur Oberflächengestaltung angesehen werden kann. In Hinblick darauf, dass Java von Beginn an als plattformunabhängige Programmiersprache konzipiert wurde, kristallisierten sich bezüglich der Oberflächenprogrammierung schnell mehrere Probleme heraus. Diese resultieren unter anderem aus den verschiedenen Hardwareansprüchen und Softwareunterstützungen. Insbesondere seien hier die grafischen Möglichkeiten unterschiedlicher Betriebssysteme zu nennen. Dennoch sollte auch bei der Entwicklung eines Frameworks eine möglichst allgemeingültige Lösung gefunden werden, die die in Abschnitt 2.1 genannten Punkte, nämlich dem Zeichnen von Grafikprimitiven, der Erzeugung von Widgets und der Ereignisbehandlung gerecht werden.

Die zunächst einfachste Lösung bediente sich den grafischen Routinen, die das Betriebssystem selbst bot. Dabei wurden die eigentlichen Befehle lediglich an

das Betriebssystem weitergeleitet. Jedes einzelne Widget erhielt dazu eine sogenannte Peer-Klasse. Diese Klassen interagierten dabei lediglich mittels einer dünnen Abstraktionsschicht mit den darunterliegenden nativen Elementen des Betriebssystems. Die Implementierung wurde dadurch relativ einfach gehalten. Da die nativen Betriebssystemelemente allerdings selbst viele weitere Routinen des Betriebssystems benötigen können, werden derartige Umsetzungen auch als *schwergewichtige Frameworks* (englisch heavyweight framework) bezeichnet. Eine detailierte Beschreibung der AWT API kann [Zukowski, 1997] entnommen werden. Die Abbildung 2.1 zeigt eine schematische Darstellung der beschriebenen Beziehung von AWT zum Betriebssystem.

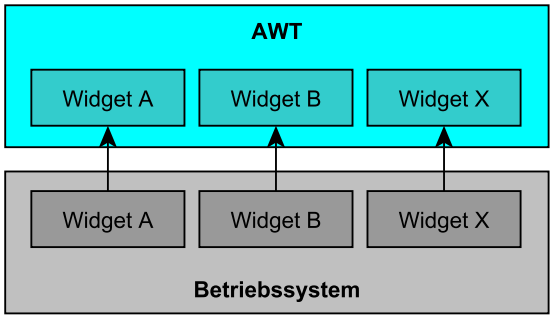

Abbildung 2.1: Schematische Darstellung von AWT

In Java umgesetzt, entstammen sämtliche Widgets der Klasse `Component` aus dem AWT Package. In Kombination mit der Klasse `Container`, die einen einfachen Weg anbietet, beliebige Widgets zu verschachteln, entstand eine komfortable Möglichkeit, Oberflächen oder auch eigene zusammengesetzte Widgets umzusetzen. In der Tabelle 2.1 sind die gängigsten, unter AWT verfügbaren Komponenten mit einer kurzen Beschreibung aufgelistet.

In einem Interview vom März 1998 mit James Gosling, einem der „Urväter“ der Programmiersprache Java, wird die eigentliche damalige Intention von AWT deutlicher:

> *The AWT was something we put together in six weeks to run on as many platforms as we could, and its goal was really just to work. So we came out with this very simple, lowest-common-denominator thing that actually worked quite well. But we knew at the time we were doing it that it was really limited. After that was out, we started doing the Swing thing, and that involved working with Netscape and IBM and folks from all over the place.*
>
> *[Ullenboom, 2008]* [1]

Es wurde demnach lediglich darauf Wert gelegt, AWT so schnell wie möglich auf vielen Plattformen lauffähig zu machen, ohne Wert auf einen umfangreichen

[1] Das Interview vom 24. März 1998 wurde von Oracle aus der Webseite gelöscht.

Bezeichnung	**Beschreibung**
Button	Schaltfläche
Canvas	Container-Element unter anderem für Zeichenoperationen
CheckBox	Beschreibungstext mit Markierungsfeld
CheckBoxMenuItem	Beschreibungstext mit Markierungsfeld in einem Menü
Choice	Drop-Down-Menü mit Texteinträgen
Component	Abstrakte Basisklasse für eigene Komponenten
FileDialog	Auswahldialog für Dateien
Label	Beschreibungstext
List	Auswahlliste von Texten
Menu	Menü, welches z.B. in einer MenuBar verwendet werden kann
MenuBar	Menüzeile in einem Fenster, kann mehrere Menüs beinhalten
MenuItem	Menüeintrag in einem Menü
PopUpMenu	Popup-Fenster mit Menüeinträgen
Scrollbar	Schieberegler zum Verschieben des sichtbaren Bereichs (View)
TextArea	Mehrzeiliges Eingabefeld für Texte
TextField	Einzeiliges Eingabefeld für Texte

Tabelle 2.1: Widgets des AWT-Packages

Funktionsumfang zu legen. Demnach war AWT auf der einen Seite einfach zu implementieren, was sich auch in der recht kurzen Entwicklungszeit von nur sechs Wochen widerspiegelte, auf der anderen Seite litt AWT unter starken Architekturproblemen.

- Speicherverbrauch und Leistungsprobleme
  Durch die Verwendung nativer Elemente des darunterliegenden Betriebssystems wurden für jedes weitere Element in einer grafischen Oberfläche neue Ressourcen belegt. Dies führte bereits bei durchschnittlich großen Formularen und Dialogen zu merklichen Leistungseinbußen.

- Einschränkung der Komponenten
  Als zweites Problem ist zu nennen, dass nicht alle Komponenten nativ auf einem System zur Verfügung standen. Daraus resultierend beinhaltete AWT nur den kleinsten gemeinsamen Nenner an Komponenten, der überall vorhanden und einsetzbar war. Auch die verbliebenen Funktionen der Komponenten waren diesem Umstand unterlegen. So ist es in AWT zum Beispiel zwar möglich, ein Button zu beschriften, jedoch nicht, ihn zusätzlich mit einem Icon zu versehen.

- Abhängige Nebeneffekte
  Es zeigten sich an einigen Stellen merkwürdige Nebeneffekte, die zu unvor-

hersehbarem Verhalten führen konnten, wenn der Programmierer diese nicht beachtet hatte beziehungsweise von diesen gar nicht wusste. So sind unter Windows Textfelder auf maximal 64Kib Zeichen beschränkt [Ullenboom, 2008], was unter anderen Systemen nicht der Fall ist.

- Abhängige Ereignisse
  Mitunter kommt es vor, dass Ereignisse in unterschiedlicher Reihenfolge erzeugt beziehungsweise anders verwendet werden. Hierzu soll das Ereignis „KeyPressed“ genannt werden, welches bei gedrückt gehaltener Taste unter Linux mehrfach ausgelöst wird, unter Windows jedoch nur einmalig [2].

- Abhängige Bedienung
  Weiterhin richtete sich ebenfalls das Handling nach dem jeweiligen Betriebssystem. Eine einheitliche Funktionsweise auf allen Systemen war damit kaum möglich.

Der zuletzt genannte Punkt könnte auch als Vorteil angesehen werden. Denn der spätere Benutzer der Software musste sich nicht erst an ein möglichweise neues Bedienkonzept gewöhnen. Die einzelnen Komponenten entsprachen optisch und haptisch exakt den gewohnten Komponenten aus anderen Anwendungen. Dennoch wurde kurz nach der Fertigstellung von AWT bereits mit der Planung von Swing begonnen, woran sich diverse Organisationen wie Netscape und IBM beteiligten.

2.3 Swing

Obwohl AWT das Problem einer einheitlichen Benutzeroberfläche lösen sollte, gelang dies kaum. Als Erweiterung von AWT 1.02 auf AWT 1.1 wurde ein neues Ereignismodell eingeführt. Das sogenannte Observer Pattern bietet hierzu eine einheitliche Schnittstelle an, bei dem sich „interessierte“ Objekte bei der Komponente „anmelden“ und informiert werden, sobald bestimmte Änderungen eintreffen [Gruntz, 2004].

Um weitere Komponenten anbieten zu können, wurden bereits im Jahre 1996 die Internet Foundation Classes (IFC) von der Netscape Corporation veröffentlicht. Dabei wurde bereits ein lightweight-Ansatz verfolgt. Die einzelnen Komponenten sollten nicht mehr nur auf die nativen Betriebssystemkomponenten gelinkt werden, sondern wurden komplett in Java geschrieben. Des Weiteren hatten einige Komponenten die Möglichkeit HTML Code zu interpretieren, was die Gestaltung von Beschriftungen vereinfachen sollte.

Im Folgejahr 1997 einigten sich Sun Microsystems, Netscape Corporation und IBM darauf, eine auf den IFC aufbauenden GUI-Bibliothek zu veröffentlichen. Diese Java Foundation Classes (JFC) genannte Bibliothek ist seit Java Version

[2] Java Bug Report „4153069: keyReleased behaviour incorrect“ `http://bugs.sun.com/view_bug.do?bug_id=4153069` (Stand März 2013)

1.2 standardgemäß im Java Development Kit (JDK) integriert. Eine sehr ausführliche Beschreibung findet sich unter anderem in [Sun Microsystems, 2001]. Die wesentlichen Bestandteile gliedern sich in die folgenden Pakete:

- Swing
  Zu Swing zählen alle grafischen Benutzerkomponenten, die zur Gestaltung von Oberflächen genutzt werden können. Hierbei wurde auf die Nutzung der Peer-Klassen, also auch der nativen Betriebssystemkomponenten, größtenteils verzichtet. Nur noch wenige Komponenten sind an die AWT-Peer-Klassen gebunden (zum Beispiel `javax.swing.JFrame`, welches von `java.awt.Frame` erbt). Sämtliche weiterführenden Komponenten werden mittels primitiven Zeichenoperationen erzeugt und sind somit nicht mehr vom System abhängig. Bei der Gestaltung muss nicht mehr auf Betriebssystemeigenheiten geachtet werden. So kann eine Schaltfläche halbtransparent und abgerundet sein, was zwar unter MacOS umsetzbar ist, unter Windows jedoch nicht nativ unterstützt wurde. Auch das Hinzufügen eines Icons zu einer Schaltfläche, was vorher bei AWT nicht der Fall war, wurde nun möglich. Abbildung 2.2 zeigt eine schematische Darstellung des Aufbaus von Swing mit den Abhängigkeiten zu AWT.

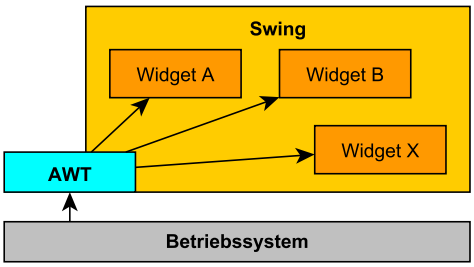

Abbildung 2.2: Schematische Darstellung von Swing

- Pluggable Look and Feel (PLaF)
  Durch die Nutzung der lightweight-Komponenten, muss auf jedem System das entsprechende Aussehen und die entsprechende Handhabung nachgebaut werden. Dazu dient das PLaF. Was in den IFC noch fest mit jeder Komponente verknüpft war, konnte nun sogar zur Laufzeit einer Anwendung ausgetauscht werden.

- Java2D API
  Sämtliche grundlegenden Basiszeichenoperationen sowie Deformationsfunktionen, wie zum Beispiel Translation und Scherung, wurden in die Java2D API ausgelagert. Die Java2D API wird unter anderem für die bereits erwähnte Swing Oberfläche genutzt und seit den frühen Versionen an diversen Stellen optimiert [3].

[3] „High Performance Graphics, Graphics Performance Improvements in the Java 2 SDK, version 1.4“
`http://www.oracle.com/technetwork/java/perf-graphics-135933.html` (Stand März 2013)

- Drag and Drop
  Die API für Drag and Drop sorgt dafür, dass Daten programmintern, unter verschiedenen Java-Programmen oder auch zu vollkommen anderen Programmen übertragen werden können.

- Accessibility
  Für Menschen mit körperlichen Beeinträchtigungen bietet die Accessibility-API eine Schnittstelle für unterschiedliche Ausgabegeräte beziehungsweise Ausgabemodifikationen an. Wie zum Beispiel Sprachausgabe oder Lesegeräte für Blinde. Die Accessibility-API ist in allen Swing-Komponenten integriert und automatisch verfügbar.

2.3.1 Komponenten

Bei der Implementierung der angebotenen Swing-Komponenten erfolgt die Realisierung nicht mehr auf Grund von Operationen des zugrunde liegenden Betriebsbeziehungsweise GUI-Systems. Das gesamte Swing-Package benötigt nur noch wenige maßgebliche native Methoden und stellt vielmehr alle nötigen Konzepte zur Verwaltung und Verarbeitung selbst zur Verfügung. Daraus resultiert, dass prinzipiell keine Abhängigkeiten mehr auf Betriebssystemseite existieren. Swing setzt durch diesen Ansatz sein eigenes Puggable Look and Feel um und ist damit vollkommen plattformunabhängig.

Die Klassenhierarchie baut auf AWT auf. Es setzt sich aus einigen schwergewichtigen Komponenten aus AWT und den neu implementierten leichtgewichtigen Swing-Komponenten zusammen. Aus AWT wurden die Klassen `java.awt.Window`, `java.awt.Frame` und `java.awt.Dialog` übernommen und erweitert. Diese so genannten Top-Level-Container stellen Bildschirmfenster dar, die die Basis einer jeden Java-GUI bilden. Auf AWT-Seite ist die Klasse `java.awt.Component` die Basisklasse aller zur Verfügung stehenden visuellen Komponenten. Auf der leichtgewichtigen Swing-Seite erbt `javax.swing.JComponent` nicht direkt von `java.awt.Component`, sondern erst von dessen Kindklasse `java.awt.Container`. Dies hat zur Folge, dass jede Komponente in Swing automatisch ein Container ist und somit weitere Komponenten aufnehmen kann. Aufbauend auf dieser Struktur existieren viele weitere Komponenten, die von einfachen Containerklassen (`javax.swing.JPanel`), über Basiselemente wie Beschriftungsfelder (`javax.swing.JLabel`) oder Schaltflächen (`javax.swing.JButton`), bis hin zu sehr komplexen Gebilden wie Tabellen (`javax.swing.JTable`) und Bäumen (`javax.swing.JTree`) führen. Alle Swing-Komponenten entstammen dem Package `javax.swing` und haben ein 'J' vorangestellt, um Namenskonflikte mit AWT-Komponenten vorzubeugen.

Die Tabelle 2.2 beinhaltet eine Auswahl der wichtigsten in Swing enthaltenen Komponenten in alphabetischer Reihenfolge.

Bezeichnung	**Beschreibung**
JButton	Schaltfläche
JCanvas	Container-Element unter anderem für Zeichenoperationen
JCheckBox	Markierungsfeld mit Beschreibungstext
JComboBox	Drop-Down-Menü mit Texteinträgen
JEditorPane	Mehrzeiliges Eingabefeld für Texte in diversen Formaten
JLabel	Beschreibungstext
JList	Auswahlliste von Texten
JMenu	Menü, welches z.B. in einer MenuBar verwendet werden kann
JMenuBar	Menüzeile in einem Fenster, kann mehrere Menüs beinhalten
JMenuItem	Menüeintrag in einem Menü
JPopUpMenu	Popup-Fenster mit Menüeinträgen
JPasswordField	Einzeiliges Eingabefeld für Passwörter
JProgressBar	Fortschrittsanzeige
JRadioButton	Markierungsfeld mit Beschreibungstext
JScrollBar	Schieberegler zum Verschieben des sichtbaren Bereichs (View)
JScrollPane	Container mit optionalen Schiebereglern
JSlider	Schieberegler
JSpinner	Inkrementelles Eingabefeld z.B. für Zahlen
JTabbedPane	Container mit Registerreitern
JTable	Tabelle
JTextField	Einzeiliges Eingabefeld für Texte
JTextPane	Mehrzeiliges Eingabefeld für Texte
JToolBar	Statusleiste in einem Fenster
JTree	Hierarchische Baumstruktur

Tabelle 2.2: Widgets des Swing-Packages

2.3.2 Model-View-Controller

Eine weitere Verbesserung gegenüber dem puren AWT ist die Trennung von Daten, Anzeige und Funktionsweise der Komponenten. Das wird durch eine konsequente Objektorientierung und dem sogenannten Model-View-Controller (MVC) Pattern ermöglicht (siehe [Burbeck, 1987] und [Gamma et al., 1993]). Das MVC Pattern besteht aus den folgenden drei Teilen:

- Model
  Das Model beinhaltet hauptsächlich die Daten, die später in einer Komponente angezeigt werden sollen. Zusätzlich können Eigenschaften definiert werden, wie zum Beispiel zu verwendende Zeichen für ein Passwortfeld oder der Inkrementwert für ein Zahlenfeld. Weiterhin bietet das Model Methoden, um die in ihm enthaltenen Daten abfragen beziehungsweise verändern zu können.

- View
  Dieser Teil ist für die reine Anzeige der ihm übergebenen Daten aus dem verknüpften Model verantwortlich. Mittels eines zuvor fest definierten Interfaces wird dabei sichergestellt, dass die gelieferten Daten korrekt verarbeitet und relevante Informationen aus dem Model korrekt im View dargestellt werden können.

- Controller
  Der Controller verknüpft Model und View miteinander. Er reagiert auf sämtliche Änderungen im Model und veranlasst den View diese darzustellen beziehungsweise reagiert auf alle Änderungen im View, wie zum Beispiel das Scrollen in einer Komponente, Eingabe neuer Daten oder Löschen von Zeilen in einer Tabelle und ändert die entsprechenden Daten im Model. Im Controller ist somit die eigentliche (Business-) Logik einer Komponente hinterlegt.

In einigen Implementierungen kann es vorkommen, dass View und Controller nicht voneinander getrennt sind. Dies ist zum Beispiel bei Swing mit den so genannten *UI Delegates* der Fall. Derartige Konstrukte, bei denen neben dem Model die Komponenten View und Controller zusammengefasst sind, werden auch als Model-Delegate-Prinzip bezeichnet [Madeyski und Stochmialek, 2005].

Der generelle Vorteil beim MVC Pattern besteht darin, dass Controller und View in der Regel nur einmal implementiert werden müssen. Eine Auswahlliste wird immer Elemente in Listenform untereinander oder nebeneinander darstellen und auf die gleichen Interaktionsmöglichkeiten zurückgreifen. Lediglich das Model ändert sich in den meisten Fällen. Bei AWT waren die Daten noch fest mit der Komponente verknüpft. Bei Swing kann ein Model auch in unterschiedlichen Komponenten dargestellt und verwendet werden. Als Beispiel soll hier eine Liste von Benutzern dienen, die in einem Model niedergeschrieben sind. Dieses Model kann nun als Datenquelle sowohl für eine Übersichtstabelle als auch gleichzeitig

für eine Auswahlliste in einem Suchdialog dienen. Es muss lediglich darauf geachtet werden, dass die Anzeige-Komponente (View) über entsprechende Interfaces auf die für sie relevante Daten zugreifen kann.

Zusätzlich dazu können durch die Trennung der Angelegenheiten Spezialisten für die einzelnen Bereiche herangezogen werden. So kann sich ein Datenbank-Spezialist um die Umsetzung des Modells kümmern und ein Designer um die des Views.

2.3.3 Look and Feel

Mit dem Pluggable Look and Feel wurde Swing mit einer herausragenden Möglichkeit ausgeliefert, das Aussehen und das Verhalten von Komponenten zu verändern. Dies war in der Hinsicht notwendig, da die lightweight-Komponenten von Swing nun nicht mehr mit nativen Betriebssystemkomponenten verknüpft waren und somit auch nicht mehr auf das Aussehen und die Funktionsweisen dieser zurückgreifen konnten. Mit dem Look and Feel von Swing sehen Anwendungen auf allen unterstützten Plattformen gleich aus. So sieht und verhält sich eine Schaltfläche mit dem *Windows Look and Feel* unter MacOS wie eine Windows-Schaltfläche. Diese Vorgehensweise bietet einer Firma zum Beispiel die Möglichkeit, ein eigenes Corporate Design für alle hausinternen Anwendungen zu designen und einfach auf alle Produkte anzuwenden. Das Java JDK enthält in der aktuellen Version 1.7 neben den bekannten Look and Feels (*MetalLookAndFeel* als Unix-Oberfläche, *MotifLookAndFeel* als Java Standard, *WindowsLookAndFeel* als Windows-Oberfläche) ein neues Crossplatform Look and Feel namens *Nimbus*. Mittels *Nimbus* [4] wird die Oberfläche als Vektorgrafik gerendert, was eine Ausgabe in jeder erdenklichen Auflösung erleichtert.

Für die Änderung des Look and Feel muss eine Anwendung nicht erst neu gestartet werden. Es kann mittels der Klasse `UIManager` mit nur einer Zeile Code on-the-fly ausgetauscht werden. Der UI-Manager dient dabei als Verwaltungseinheit. Für jedes Look and Fell und für jede Komponente existiert ein UI-Objekt, welches dafür zuständig ist, der verknüpften Komponente das entsprechende Aussehen zu geben. So hat ein `javax.swing.JButton` ein zugeteiltes ButtonUI-Objekt und ein `javax.swing.JTable` ein zugeteiltes TableUI-Objekt. An dieser Stelle können Entwickler das Aussehen von Komponenten den eigenen Wünschen und Bedürfnissen anpassen.

2.3.4 Vor- und Nachteile von Swing

Die hauptsächlichen Verbesserungen von Swing gegenüber AWT sind kurz zusammengefasst:

- Unabhängigkeit von nativen Komponenten des Wirtsbetriebssystems
- Erweiterter Funktionsumfang der Komponenten

[4] Weitere Informationen zum Nimbus Look and Feel unter `http://docs.oracle.com/javase/tutorial/uiswing/lookandfeel/nimbus.html`

- Freies Design von Komponenten (Transparenz, Formen frei wählbar, Look and Feel)
- Kein kleinster gemeinsamer Nenner
- Model-View-Controller Konzept

Die oben aufgeführten Neuerungen wurden zum Teil teuer erkauft. So ist die Swing-Bibliothek sehr ressourcenhungrig. Da in Swing jede Komponente einzeln gezeichnet wird, ist die CPU- und Speicherlast relativ hoch. Dies führte schnell dazu, dass Java generell als „träge“ und „speicherhungrig“ bezeichnet wurde. Dieser Effekt wurde noch verstärkt, wenn ohne des Zutuns des Entwicklers ungenutzte Speicherbereiche durch den Garbage Collector von Java freigegeben wurden [Scarpino et al., 2005].

Des weiteren hinkt die Entwicklung eines spezifischen Betriebssystem Look and Feel in der Regel der Entwicklung des Betriebssystems selbst hinterher und ist abhängig von der installierten JRE. So kann es sein, dass eine Anwendung, die unter Windows 7 ausgeführt wird, bedingt durch eine ältere JRE noch wie Windows XP aussieht. Außerdem ist es den Entwicklern nicht gänzlich gelungen, feinere Unterschiede im Aussehen zwischen Look and Feel und dem eigentlichen Betriebssystem zu entfernen. So sieht man zum Beispiel einem Öffnen-Dialog die Java-Herkunft noch an.

Mit jeder neu veröffentlichten Java Version werden mehr und mehr der vorhanden Inkompatibilitäten ausgebessert und Performanceprobleme durch optimierten Code behoben. Weiterhin kommen durch verbesserte Hardware und größere Speicher „träge“ Oberflächen nur noch in geringem Maße vor. Allerdings ist aus dem Grund die Swing-Bibliothek für Rechner mit begrenzten Hardwareressourcen (zum Beispiel Smart Phones) weniger geeignet.

2.4 SWT

Bereits im Jahr 1998, also ein Jahr nach der Veröffentlichung der Java Foundation Classes mit Swing, begann IBM mit der Entwicklung des Nachfolgers von „VisualAge for Java“, einer integrierten Entwicklungsumgebung für einige gängige Programmiersprachen wie C/C++, Smalltalk und Java. Das später in *Eclipse* umbenannte Produkt benötigte Komponenten zur Gestaltung von Oberflächen. Diese mussten auf Grund der geplanten Dimensionen performant und ressourcenschonend sein. Swing kam nicht in Frage, da die genannten Bedingungen zu der Zeit nicht erfüllt wurden.

So wurde das SWT ins Leben gerufen, was eine echte Alternative zu AWT und Swing darstellen sollte. Ebenso wie AWT greift SWT auf die nativen Betriebssystemelemente zurück. Dazu zählen einfache Komponenten wie Labels, Schaltflächen und Tabellen. Zusätzlich wurden Funktionen wie Drag and Drop, die Zwischenablage und ActiveX (Windows) unterstützt. Dabei ist SWT von AWT komplett unabhängig, da keinerlei Elemente aus Fremdpaketen (AWT, Swing,

Java 2D) genutzt werden. Durch die enge Bindung von SWT an das darunterliegende Wirtssystem müssen Anwendungen, die zur Oberflächengenerierung SWT nutzen, mit einer betriebssystemspezifischen Bibliothek ausgeliefert werden. SWT ist hierbei, wie AWT, als heavyweight-Framework anzusehen, da der Zugriff auf Bibliotheken des Betriebssystems Ladevorgänge weiterführender Bibliotheken nach sich ziehen kann. Die Abbildung 2.3 zeigt eine schematische Darstellung der Abhängigkeiten von SWT zum Betriebssystem.

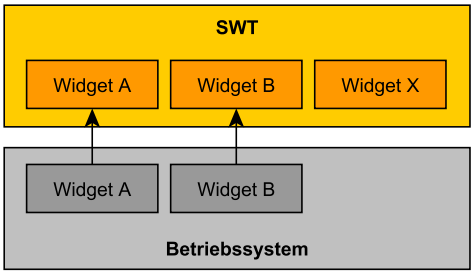

Abbildung 2.3: Schematische Darstellung von SWT

Anders als AWT stellt SWT nicht nur Widgets bereit, die auf allen Zielplattformen verfügbar sind, sondern es werden auch plattformspezifische Widgets angeboten. Auf der Seite der SWT-Bibliothek für Microsoft Windows ist dies beispielsweise der Zugriff auf Object Linking and Embedding (OLE) Dokumente über die Klasse `org.eclipse.swt.ole.win32.OleFrame`. Mittels OLE ist es SWT-Entwicklern dann zum Beispiel möglich, den Internet Explorer anzusprechen oder Office Dokumente zu öffnen und auszudrucken. Eine ausführliche Beschreibung des OLE Konzeptes liefert Kraig Brockschmidt in [Brockschmidt, 1995]. Komponenten und Funktionen, die auf einer Zielplattform nicht verfügbar sind, können im SWT nachträglich implementiert und dadurch auf dem System emuliert werden.

Ursprünglich war das SWT ausschließlich für die Oberflächengestaltung der Entwicklungsumgebung *Eclipse* gedacht. 2001 wurde das SWT allerdings von *Eclipse* entkoppelt und der Nutzergemeinde als eigenständiges Paket zur Verfügung gestellt. Mittlerweile erfreut sich die Alternative zu AWT und Swing großer Beliebtheit. Insbesondere für mobile Endgeräte, hier sei Windows CE genannt, ist das SWT sehr performant, da die Bibliothek auf einem kleinen nativen Kern aufbaut und recht leichtgewichtig ist. Die populärste Einsatzform bleibt allerdings die Entwicklungsumgebung *Eclipse*. Die Eclipse RCP bildet hierfür ein großes Rahmenwerk für komplexe grafische Oberflächen, basierend auf SWT, was Aufgaben wie Benutzerkonfigurationen oder Plugin- und Update-Verwaltung bereits vorimplementiert.

SWT steht in der Version 3.6.2 für die folgenden Betriebssysteme zur Verfügung:

- AIX (PPC/Motif)
- FreeBSD (x86/GTK 2)

- FreeBSD (AMD64/GTK 2)
- HP-UX (HP 9000/Motif)
- Linux (x86/GTK 2)
- Linux (AMD64/GTK 2)
- Linux (PPC/GTK 2)
- Linux (x86/Motif)
- Mac OS X (PPC/Carbon)
- Mac OS X (x86/Carbon)
- Mac OS X (x86 64/Carbon)
- Mac OS X (x86/Cocoa1)
- Mac OS X (x86 64/Cocoa1)
- QNX (x86/Photon)
- Solaris 8 (SPARC/GTK 2)
- Solaris 8 (SPARC/Motif)
- Windows (x86/Win32)
- Microsoft Windows CE (ARM PocketPC)
- Microsoft Windows CE (ARM PocketPC, J2ME profile)

Da die Widgets der einzelnen Betriebssysteme von Java entkoppelt sind, werden diese auch nicht vom Garbage-Collector verwaltet. Die Betriebssysteme unterhalb von SWT benötigten daher einen expliziten Belegungshinweis und eine explizite Freigabe von Ressourcen, die vom Entwickler selbst verwaltet werden muss. Die Freigabe wird in SWT durch die Methode `dispose()` auf den einzelnen Komponenten vorgenommen. Jedes generierte Widget muss mittels `dispose()` wieder aufgelöst werden. Dazu zählen auch Objekte wie Farben und Schriften. Eine Ausnahme besteht, wenn die dispose-Methode von einem Eltern-Widget aufgerufen wird. In dem Fall werden rekursiv auch alle registrierten Widgets unterhalb des Eltern-Widget freigegeben. Dieses Objekt existiert dann zwar noch in Java, das darunterliegende Objekt auf Betriebssystemebene ist allerdings zerstört. Der Zugriff auf ein derartiges Objekt beziehungsweise das mehrmalige Aufrufen der dispose-Methode hat dann unweigerlich eine „Widget is disposed“-Exception zur Folge. Im Vergleich mit dem Swing-Framework verhindert der Garbage-Collector von Java dort zwei Klassen von Fehlerquellen. Zum einen werden nicht mehr benötigte Ressourcen automatisch wieder freigegeben. Zum anderen werden Ressourcen, die an anderen Stellen im Programm noch genutzt werden, nicht zu früh freigegeben. Auf beide Fehlerquellen muss der Entwickler im SWT selbst achten.

2.4.1 JFace

SWT bietet lediglich einen Zugriff auf die nativen Betriebssystemkomponenten und stellt diese dar. Eine einheitliche Verwaltungsstruktur für Daten, wie dem MVC-Konzept in Swing oder Ereignisbehandlungen wurde mehr oder weniger außen vor gelassen. Das Toolkit JFace erweitert die Funktionen vom SWT und liefert eine Abstraktionsschicht (Viewer) für den Zugriff auf die Komponenten. Damit ist eine relativ saubere Trennung von Modell und Darstellung möglich. Weiterhin werden Actions implementiert, mit denen Ereignisse aus der GUI ausgekoppelt werden können. Außerdem bietet JFace ein reichhaltiges Set an zusammengesetzten Komponenten an, wie zum Beispiel Wizards und Dialoge. JFace wurde von Anfang an zur Verwendung mit der Eclipse RCP konzipiert und umgesetzt, so dass ein gutes Zusammenspiel zwischen beiden gewährleistet ist. Soll JFace ohne die RCP funktionieren, müssen trotzdem einige Bibliotheken aus dem Eclipse-Projekt installiert werden. Trotz dieses Umstandes erleichtert JFace die Entwicklung von SWT basierten Anwendungen teils erheblich.

2.4.2 Vor und Nachteile von SWT

Wie bereits von den ursprünglichen Entwicklern angedacht, ist eines der größten Vorteile des SWT die Nutzung der Peer Klassen und damit der betriebssystemeigenen Widgets. Dadurch ist das Aussehen und die Haptik einer grafischen Nutzeroberfläche immer mit einer nativen GUI vergleichbar. Ein Nutzer einer auf SWT basierenden Software muss sich nicht an neue Bedienkonzepte oder ein verändertes Aussehen gewöhnen. Es erfolgt eine nahtlose Einbettung der Anwendung in die jeweilige Betriebssystemumgebung. Da direkt auf die jeweilige Ereignisverarbeitung des Wirtssystems aufgesetzt wird, sind die Reaktionszeiten mit denen des Systems vergleichbar. Zudem wirken sich geänderte Grafik- und Hardware-Einstellungen des Wirtssystems ebenso auf die SWT-Widgets aus, wodurch eine gewisser Grad an Robustheit erreicht wird.

Die Bindung kann allerdings auch als große Schwäche von SWT angesehen werden. Denn durch den nötigen Zugriff auf native Ressourcen des Betriebssystems ist eine plattformunabhängige Programmierung, so wie Java es ermöglichen soll, nicht vorstellbar. Im Betriebssystem Windows von Microsoft werden grafische Benutzeroberflächen mittels GDI beziehungsweise seit Windows XP/Server 2003 mit GDI+ [5] umgesetzt. Mit jedem neuen Widget, jeder neuen Schriftart oder neuen Farballokation werden auf Betriebssystemseite Ressourcen reserviert, die nicht durch den Garbage Collector von Java verwaltet werden. Demnach muss der Entwickler selbst für die Freigabe reservierter Bereiche sorgen. Bei unsauberen Programmierungen kann ein Java-Program damit auch das System außerhalb der JVM verstopfen. Zudem sind unter GDI nur 1200 sogenannte Handels, in diesem Fall reservierte Objekte für die grafische Darstellung, zugelassen. Unter

[5] Microsoft Development Network „GDI+“ `http://msdn.microsoft.com/en-us/library/ms533798%28v=vs.85%29.aspx` (Stand März 2013)

GDI+ sind es insgesamt 2^{16}, wobei die Anzahl pro Anwendung auf voreingestellte 10.000 Objekte begrenzt ist. Dieser Wert lässt sich zwar in der Registry von Windows auf das Maximum einstellen, ist jedoch dennoch dadurch beschränkt. Von Seiten Microsofts heißt es dazu ferner:

> *When an application exceeds its GDI object quota, the application can no longer allocate GDI objects. However, in this scenario, the application continues to operate as if it can allocate objects and starts painting objects on the screen incorrectly. An additional effect of this behavior is that the operating system menus do not handle the situation correctly. Menus for other applications are also affected.*

Im Microsoft Knowledgebase Eintrag mit der Nummer 838283 [6] heißt es, dass das beschriebene Problem zusätzlich Menüs anderer Programme betreffen kann, sich also über die Grenzen der eigenen Anwendung hinaus ausbreiten kann. Dieses Beispiel soll verdeutlichen, dass die Bindung von SWT mit dem Betriebssystem zu unerwarteten Reaktionen in der Anwendung selbst oder sogar im kompletten System führen kann.

2.5 Vergleich Swing und SWT

Im Folgenden sollen die am weitesten verbreiteten Techniken Swing und SWT miteinander verglichen werden, um Vor- und Nachteile darzustellen. Dazu soll in den nächsten Abschnitten genauer auf diverse Gesichtspunkte hinsichtlich Verwendbarkeit, Performance und Programmierkomfort eingegangen werden.

2.5.1 Native Look and Feel

SWT ist ähnlich aufgebaut wie AWT, es nutzt eine eins zu eins Beziehung zwischen der Java-Komponente und dem nativen Gegenpart auf Betriebssystemseite. Durch diese recht simple Vorgehensweise erhält die spätere Anwendung automatisch das Aussehen und die Handhabung einer nativen Anwendung. Swing hingegen bildet das Aussehen mittels dem Pluggable Look and Feel lediglich nach. Java kommt mit der bisherigen Implementierung zwar nah an das eigentliche Aussehen heran, aber leider nicht komplett. Ein möglicher Lösungsansatz für einige Entwickler ist es daher, ein systemfremdes Look and Feel zu verwenden, wie zum Beispiel das seit Java 1.7 verfügbare Nimbus Look and Feel. Microsoft verfolgt bereits seit einigen Jahren einen ähnlichen Ansatz. Als Beispiel soll hier das Büropacket Office von Microsoft dienen. Die Oberfläche von Office XP entsprach damals noch weitestgehend der „normalen“ Windows XP Oberfläche. In jeder folgenden Office Version (2003, 2007, 2010) wurde an dem Aussehen der Oberfläche gefeilt und sich damit mehr und mehr vom Aussehen von Windows Vista und später Windows 7 entfernt.

[6] Microsoft Knowledgebase „Desktop application menus are improperly displayed if a process exceeds its GDI object quota in Windows XP or in Windows 2000“ `http://support.microsoft.com/kb/838283` (Stand März 2013)

Und auch andere Anwendungen setzen bereits seit vielen Jahren auf ein alternatives und veränderbares Aussehen. Hierfür ist Winamp mit seinem Skin-Konzept ein sehr gutes Beispiel.

2.5.2 Plattformunabhängigkeit

Java ist eine Programmiersprache, die es dem Entwickler ermöglichen soll, Anwendungen unabhängig von der späteren Plattform zu entwickeln. Als großes Ziel diente hierbei seit Beginn der Entwicklung von Java der Slogan „write once, run anywhere“. Diese Stärke von Java wird durch SWT zunichte gemacht, da eine enge Bindung mit dem Betriebssystem besteht. Des Weiteren muss jeder Entwickler Eigenheiten der Zielplattform kennen und weitestgehend selbst lösen beziehungsweise umgehen. Wie zum Beispiel die Beschränkung für Längen von Texteingaben in Textfeldern unter Windows, die auf 64 KiB beschränkt ist [Ullenboom, 2008]. Je größer eine Anwendung wird, desto schwieriger wird es auch sein, derartige Limitierungen zu umgehen. Die dadurch nötigen Anwendungstests müssen daher auf allen geplanten Plattformen durchgeführt werden und können bei großen Anwendungen schnell unüberschaubar sein.

Da Swing auf allen Plattformen nur auf einen sehr begrenzten Teil des nativen Window-Toolkit zugreift und alle Komponenten selbst zeichnet und verwaltet, entstehen hier keine derartigen Probleme. Eine Swing-Oberfläche wird auf allen Plattformen gleich aussehen, gleich agieren und hat die gleichen Freiheiten beziehungsweise Einschränkungen.

2.5.3 Automatische Speicherverwaltung

So wie bereits die Plattformunabhängigkeit mehr oder weniger aus dem Java SWT Konzept übergangen wurde, verhält es sich auch mit der automatischen Speicherverwaltung. Der Garbage Collector war und ist das Aushängeschild für Java-Anwendungen. Speicherbereiche, die unter C/C++ noch mit `allocate` reserviert und mit `free` wieder freigegeben wurden, werden mit Java automatisch verwaltet. Der Entwickler muss sich nicht mehr darum kümmern. Egal ob in kleinen eigenen Klassen, großen Anwendungen oder auch der Swing-Bibliothek.

Mit der Einführung von SWT muss der Entwickler allerdings wieder darauf achten, belegte Systemressourcen durch die Methode `dispose()` wieder freizugeben [Daum, 2004]. Frei nach dem Motto „If you created it, you dispose it.“ [MacLeod und Northover, 2001]. Wenn zum Beispiel ein Anwendungsstrang ein Textfeld erzeugt, ein anderer diesen mittels `dispose()` freigibt, generiert jeder weitere Zugriff auf dieses Textfeld eine `DisposeException`. Da selbst Schriften und Farben Ressourcen des eigentlichen Betriebssystems belegen, müssen auch diese nach der Benutzung wieder freigegeben werden.

Auch hier verhält sich Swing Java-typisch und nutzt die Funktionen der automatischen Speicherverwaltung.

2.5.4 Flexibilität und Anpassbarkeit

Swing bietet als Toolkit sowohl Komponenten- als auch Anwendungsentwicklern reichhaltige Anpassungsmöglichkeiten. unter anderem ist es sehr einfach die Oberflächengestaltung eines Widgets, die Ereignisbehandlung oder die vordefinierte Tastaturbelegungen anzupassen. Da SWT lediglich eine Abstraktionsschicht über einem betriebssystemspezifischen Toolkit ist, kann nur der kleinste gemeinsame Nenner der auf allen Systemen verfügbaren Komponenten angeboten werden. Aus diesem Grund ist eine optische und haptische Anpassung der Komponenten unter SWT sehr schwierig, wenn nicht sogar unmöglich. Es ist dabei nicht ausgeschlossen, dass sogar Änderungen im nativen Code vorgenommen werden müssten.

Eine der Stärken von Swing ist das Nutzen eines Custom Look and Feel. Dadurch können Firmen ihr Corporate Design auf allen Produkten ausliefern, ohne hunderte Zeilen Code anpassen zu müssen. Heute existieren viele sowohl Javainterne als auch Look and Feels von Drittanbietern, die in den eigenen Anwendungen verwendet werden können. Da SWT dahingehend ausgelegt ist, das native Look and Feel des Wirtsbetriebssystems zu nutzen, ist hier eine Umsetzung eines Corporate Designs nur mit viel Aufwand möglich.

2.5.5 Programmierfreiheiten

SWT Komponenten sind immer mit einer nativen Komponente des Betriebssystems verknüpft. Sobald also eine derartige Komponente in Java erzeugt wird, werden auch Ressourcen im GUI-Toolkit des Betriebssystems belegt. Unter SWT ist es erforderlich, dass jede neue Komponente in einem Elterncontainer initiiert wird. Diese kann dann innerhalb dessen frei positioniert oder per Layoutmanager ausgerichtet werden. Eine spätere Änderung dieses Elterncontainers ist allerdings nicht mehr möglich. Daher hat jedes SWT GUI-Element einen Konstruktor, der mindestens die Angabe des Elternelements benötigt. Die Programmierweise kann hierbei als top-down bezeichnet werden, da zuerst die Container-Elemente erzeugt werden (müssen), bevor die einzelnen Komponenten hinzugefügt werden können. Der Komponentenbaum, der sich durch das Verschachteln der einzelnen Container und Widgets aufbaut, ist starr. Lediglich ein nachträgliches Hinzufügen oder Löschen von Komponenten ist möglich, nicht jedoch das Umsortieren bestehender Knoten. Das Listing 2.1 zeigt eine simple Oberfläche in Java, umgesetzt mit SWT.

Listing 2.1: Beispielcode einer einfachen SWT-Oberfläche

```
1  Display display = new Display();
2
3  Shell shell = new Shell(display);
4  shell.setSize(220, 120);
5  shell.open();
6
7  Label label = new Label(shell, SWT.NONE);
8  label.setText("Email:");
9  label.setBounds(10, 10, 40, 20);
10
11 Text text = new Text(shell, SWT.BORDER);
12 text.setTextLimit(20);
13 text.setBounds(50, 10, 150, 20);
14
15 while (!shell.isDisposed())
16   if (!display.readAndDispatch())
17     display.sleep();
18
19 display.dispose();
```

In Zeile 01 wird ein Objekt vom Typ `Display` erzeugt, welches Zugriff auf das native Grafiksystem realisiert. In den Zeilen 03 bis 05 wird ein Fenster erzeugt und konfiguriert, in welches alle späteren Widgets eingefügt werden. Diese werden in den Zeilen 07 bis 13 erzeugt. An dieser Stelle sei auf die beiden Konstruktoren der Widgets (Zeile 07 und 11) und den direkten Bezug zum Elterncontainer (in diesem Fall das Hauptfenster) hingewiesen. Die abschließende Schleife wird so lange ausgeführt, bis der Benutzer das Fenster schließt. Sie ist ebenfalls für das Auslesen der Systemereignisse zuständig. Abschließend werden alle belegten Ressourcen des Betriebssystems in Zeile 19 wieder freigegeben.

An diesem Punkt lässt Swing dem Entwickler ebenfalls mehr Freiheiten. Einerseits kann ähnlich wie bei SWT eine top-down Strategie genutzt werden. Dazu wird erst ein Container erzeugt und diesem anschließend per `add()` eine Komponente oder auch ein weiterer Container hinzugefügt. Die Positionierung erfolgt ebenfalls frei mittels direkter Positions- und Größenangabe oder mittels eines gewählten Layoutmanagers. Andererseits können auch zuerst die hinzuzufügenden Komponenten erzeugt werden und im Anschluss das zusammenfassende Elternelement. Der Komponentenbaum lässt sich bei Swing leicht erweitern, verkleinern und auch nachträglich umsortieren. Das Listing 2.2 zeigt die Swing-Umsetzung der in Listing 2.1 vorgestellten SWT-Oberfläche.

Listing 2.2: Beispielcode einer einfachen Swing-Oberfläche

```
1  JFrame frame = new JFrame();
2  frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
3  frame.setSize(220, 120);
4  frame.setLayout(null);
5  frame.setVisible(true);
6
7  JLabel label = new JLabel("Email:");
8  label.setBounds(10, 10, 40, 20);
9  frame.add(label);
10
11 JTextField text = new JTextField();
12 text.setBounds(50, 10, 150, 20);
13 frame.add(text);
```

In den Zeilen 01 bis 05 wird das Hauptfenster erzeugt und mit diversen Parametern, wie der Fenstergröße, vorkonfiguriert. Die Zeilen 07 bis 13 erzeugen insgesamt zwei Widgets und fügen diese in den Zeilen 09 und 13 dem Hauptfenster hinzu. Diese Reihenfolge könnte problemlos umgedreht werden, so dass die Widgets (Zeile 07 und 11) vor dem Hauptfenster (Zeile 01) generiert werden.

Zudem ist in nahezu allen Widget Klassen von SWT (abgesehen von `Canvas` und `Composite`) folgende Bemerkung zu lesen:

```
IMPORTANT: This class is not intended to be subclassed.
```

Das bedeutet, dass eigene Klassen nicht von SWT-Widgets erben können. Dies erschwert eine Anpassung bestehender Widgets.

2.5.6 Performance und Speicherverbrauch

SWT speichert die Zustände in den nativen Elementen des Betriebssystems. Diese werden nicht redundant in Java abgelegt. Dadurch ist SWT recht schlank und zum Beispiel gut auf mobilen Endgeräten mit wenig Speicher einzusetzen. Des Weiteren verwenden die verschiedenen Komponenten automatisch die integrierte Grafikbeschleunigung, soweit dies vom Betriebssystem unterstützt wird.

Swing hat seit Anbeginn den schlechten Ruf, viele Systemressourcen zu verbrauchen und träge zu reagieren. Dies sei darin begründet, da Swing sämtliche Komponenten selbst erzeugt, verwaltet und zeichnet. Oracle, beziehungsweise zuvor die Firma Sun, arbeiten allerdings daran und verbessern die Performance mit jeder neu erscheinenden Java Version. Tests aus dem Jahr 2005 belegen, dass Swing SWT in den meisten performancerelevanten Disziplinen in nichts nachsteht [7].

[7] SWT Vs. Swing Performance Comparison, Revision 1.4
`http://pub.cosylab.com/CSS/DOC-SWT_Vs._Swing_Performance_Comparison.pdf`
(Stand März 2013)

Kapitel 3

3 Umsetzung von SWT

Durch die in den vorherigen Kapiteln mehrfach beschriebenen Abhängigkeiten zum Wirtsbetriebssystem ergeben sich, besonders im Umfeld der Plattformunabhängigkeit, mehrere Probleme. Da Java als Programmiersprache eine gegenüber systemspezifischen Eigenheiten tolerante Plattform bietet, kann es besonders für Entwickler sehr frustrierend sein, sich mit dem Wesen eines Betriebssystems auseinandersetzen zu müssen. In diesem Kapitel soll darauf eingegangen werden, wie sich SWT auf unterschiedlichen Plattformen (hauptsächlich Windows, Linux und MacOS) verhält. Hierbei soll das Hauptaugenmerk auf die Unterschiede in den Klassen gelegt werden, die es dem Entwickler erschweren, plattformunabhängig zu entwickeln. Dieser Abschnitt unterteilt sich in insgesamt drei Teile. Als erstes sollen die Klassen selbst untersucht werden, ob die angebotenen Methoden auch in allen SWT-Bibliotheken existieren. Als zweites werden exemplarisch die Quelltexte zweier Widgets näher betrachtet und gezeigt, dass teils nützliche Funktionen nicht überall abgebildet beziehungsweise nachimplementiert wurden. Im letzten Abschnitt wird abschließend an einem Beispiel gezeigt, dass sich selbst die erwarteten Resultate einzelner Methoden stark unterscheiden. Auf Swing muss an dieser Stelle unter den gegebenen Untersuchungspunkten nicht eingegangen werden, da in der Regel identische Bibliotheken für alle Plattformen genutzt werden und so keine Unterschiede zu erwarten sind.

3.1 Untersuchung der vorhandenen Klassen und Methoden

Im Folgenden soll eine Vergleichsmethode gezeigt werden, mit der die existierenden SWT-Bibliotheken für unterschiedliche Plattformen verglichen werden können. Dabei wird die Version 3.6.2 als Referenz herangezogen. Die Gründe dafür sind einerseits die weite Verbreitung dieser SWT Version. Des Weiteren existieren bisher (Stand Februar 2013) keine 3.7.2 Bibliotheken für Linux (PPC/GTK 2), Linux (x86/Motif), Solaris 10 (SPARC/Motif) und QNX (x86/Photon). Mac OSX (Mac/Carbon) wird weiterhin vom Hersteller auf der Webseite als „unsupported“ ausgewiesen. Aus den genannten Gründen würden also insgesamt fünf

Plattformvarianten aus dem Vergleich entfallen.

Auf der Herstellerwebseite von SWT sind für die folgenden Betriebssysteme insgesamt 19 SWT-Bibliotheken in der Version 3.6.2 gelistet:

- Windows
- Windows (x86_64)
- Windows CE (ARM PocketPC)
- Windows CE (ARM PocketPC, J2ME profile)
- Linux (x86/GTK 2)
- Linux (x86_64/GTK 2)
- Linux (PPC/GTK 2) [nicht als 3.7.1]
- Linux (PPC64/GTK 2)
- Linux (x86/Motif) [nicht als 3.7.1]
- Solaris 10 (SPARC/GTK 2)
- Solaris 10 (x86/GTK 2)
- Solaris 10 (SPARC/Motif) [nicht als 3.7.1]
- HPUX (IA64_32/Motif)
- QNX (x86/Photon) [nicht als 3.7.1]
- AIX (PPC/Motif)
- AIX (PPC64/GTK 2)
- Mac OSX (Mac/Cocoa)
- Mac OSX (Mac/Cocoa/x86_64)
- Mac OSX (Mac/Carbon) [als 3.7.1 als unsupported gekennzeichnet]

Dies erlaubt einen umfangreichen Vergleich zwischen allen unterstützten Plattformen. Wie eingangs beschrieben, sollen die einzelnen Klassen dahingehend überprüft werden, ob alle Klassen identische Methoden auf allen Plattformen besitzen. Das Resultat beziehungsweise der Rückgabewert jeder einzelnen Methode kann in einem automatischen Test nur sehr schwer überprüft werden und soll nicht Bestandteil dieser Arbeit sein. Exemplarisch wird darauf jedoch in den nächsten Kapiteln kurz eingegangen werden. In dem beschriebenen Vergleich sollen

Methoden als identisch gelten, wenn sie in derselben Klasse einer anderen SWT-Bibliothek eine identische Anzahl an Parametern vorweisen und diese den gleichen Typ besitzen. Dieselbe Klasse einer anderen SWT-Bibliothek kann über den Klassennamen und das Package, in dem die Klasse liegt, identifiziert werden.

Für den Verglich werden zunächst alle vorhandenen SWT-Bibliotheken der gleichen Version heruntergeladen und in einem gemeinsamen Stammverzeichnis entpackt. Dadurch wird eine gewisse Übersichtlichkeit gewahrt. Unter den extrahierten Dateien befinden sich unter anderem die Dateien *swt.zip*, die die Source Codes enthalten, und *swt.jar*, die als vorkompilierte Bibliothek in Java-Projekte eingebunden werden kann.

Um alle vorhandenen Methoden auflisten und vergleichen zu können, bieten sich zwei Vorgehensweisen an. Zum einen könnte mit speziellen Suchmustern (reguläre Ausdrücke) in den *.java Dateien der *swt.zip* nach vorhanden Methoden gesucht werden. Da die Source-Dateien allerdings nicht zwangsläufig gleich formatiert sind, ist diese Methode recht fehleranfällig und aufwändig. Des Weiteren müssten im Anschluss noch die Typen der Parameter interpretiert werden, um wirklich nur identische Methoden zu vergleichen. Außerdem würden Modifikationen wie **public** und **private** Einschränkungen bei Klassen und Methoden nur mit großem Aufwand korrekt zu interpretieren seien (Stichwort Vererbungshierarchie). Daher ist diese Vorgehensweise recht aufwändig und daher weniger geeignet.

Java bietet allerdings mittels dem Classloader-Konzept und dem integrierten Reflection-Model eine sichere und einfach zu verwendende Alternative an.

- Classloader
  Wie der Name es bereits andeutet, ist ein Classloader dafür verantwortlich, Klassen zu laden. Dabei wird aus einer Datenquelle, in den meisten Fällen einer Datei, eine Klasse eingelesen und die Informationen in das Laufzeitsystem eingebracht. Dies geschieht im Allgemeinen vollkommen automatisch und ohne weiteres Zutun des Entwicklers. Java liefert bereits unterschiedliche Classloader, die auch dazu dienen können, eigene Classloader zu schreiben, um zum Beispiel auf verschlüsselte Java Archive zugreifen zu können.

- Reflection
  Mittels der Nutzung von Reflections ist es unter Java möglich, Klassen und Objekte, die von der Java Virtual Machine im Speicher gehalten werden, zu analysieren. Dabei kann auf die Metadaten beziehungsweise Struktur von Klassen (zum Beispiel verwendete Felder, genutzte Klassenmodifikatoren, vorhandene Methoden) zur Laufzeit zugegriffen werden. In der Entwicklungsumgebung *Eclipse* werden Reflections zum Beispiel verwendet, um eine automatische Codevervollständigung anbieten zu können. Dazu wird die Klasse des aktuell selektierten Objekts analysiert und in einem Popupfenster alle verfügbaren Methoden samt Parameterübersicht angezeigt.

Für die Analyse ist es zunächst wichtig eine vollständige Übersicht über alle vorhandenen Klassen in allen vorhandenen SWT-Bibliotheken zu erhalten. Dazu werden mittels der Klasse `JarInputStream` alle Klassen einer SWT-Bibliothek

(*swt.jar*) ausgelesen und in einem Set gespeichert. Das Set beinhaltet dabei keine Duplikate und ermöglicht so schnell eine zuverlässige Übersicht. In das Set werden nur Klassen aufgenommen, die die folgenden Punkte erfüllen:

1. Die Klasse muss öffentlich sein.
   Private Klassen sind für den Entwickler nicht sichtbar und können daher ignoriert werden.

2. Die Klasse muss aus dem Package `org.eclipse.swt` stammen.
   Klassen aus anderen Packages sind für diesen Vergleich irrelevant, da diese nicht für grafische Oberflächen genutzt werden.

3. Die Klassen dürfen nicht aus dem Package `org.eclipse.swt.internal` stammen.
   Diese beinhalten lediglich plattformspezifische Implementierungen, die für den Entwickler unsichtbar und daher nicht relevant sind.

4. Die Klasse darf keine geschachtelte Klasse sein.
   Geschachtelte Klassen sind Klassen, die in anderen Klassen anonym erzeugt werden. Diese sind für den Entwickler nicht sichtbar und werden daher ignoriert. Derartige Klassen entsprechen dem Schema *[Klassenname]$[Zähler].class* und können dadurch leicht herausgefiltert werden.

Im nächsten Schritt wird diese Liste von Klassen sequenziell für alle Betriebssysteme durchlaufen. Dazu wird für jede Plattform mittels eines Classloader die entsprechende *swt.jar* geladen. Durch die Methode `Class.forName()` kann zum einen überprüft werden, ob die Klasse überhaupt existiert und im Erfolgsfall ein Liste von vorhandenen Methoden dieser speziellen Klasse zurückgeliefert werden. Der Parameter `initialize` der Methode gibt dabei an, ob eine gefundene Klasse nur geladen (**false**) oder auch erzeugt werden soll (**true**). Da die Analyse unter einem einzigen Betriebssystem durchgeführt wird, muss dieser Parameter auf **false** belassen werden. Andernfalls würde Java unter Umständen versuchen, plattformfremde, nicht vorhandene Betriebssystemkomponenten zu laden. Dies würde zum Beispiel der Fall sein, wenn die Analyse für die Linux SWT-Bibliothek (x86/GTK 2) unter Windows gestartet wurde und eine Klasse daraus auf das GTK 2 Toolkit zuzugreifen versucht, welches unter Windows nicht unbedingt existieren muss. Das Resultat dieses Schrittes ist ein Set an Klassen mit der Information, auf welchen Plattformen diese verfügbar ist.

Aufbauend auf diesem Set, können nun alle Methoden verglichen werden, die für jeweils eine Klasse in den unterschiedlichen SWT-Bibliotheken existieren oder auch nicht existieren. Für diesen Zweck bietet sich die Methode **class**`.getMethods()` an, die eine Liste von den existierenden Methoden mit entsprechender Parameterliste der Klasse zurückgibt. Aus dieser müssen im Anschluss noch alle Methoden herausgefiltert werden, die nicht mit dem Modifikator **public** versehen sind. Da diese für den Entwickler nicht sichtbar sind, müssen sie aus dem Vergleich herausgezogen werden.

Auswertung der Analyse

Eine auf dem oben beschriebenen Verfahren durchgeführte Analyse wurde auf die eingangs aufgelisteten 19 SWT-Bibliotheken angewendet. Die daraus generierte Übersicht im CSV Format wurde mittels Microsoft Excel eingesehen. In der Übersicht befinden sich insgesamt 9.933 Methoden aus 294 unterschiedlichen Klassen. Diese sollten auf Grund des Ziels einer Plattformunabhängigkeit bereits identisch sein. Ein genauerer Blick zeigt allerdings ein anderes Bild. In der nachfolgenden Auflistung sind vier ausgewählte Beispiele für das Vorhandensein auf unterschiedlichen Plattformen gelistet.

- `org.eclipse.swt.awt.SWT_AWT`
  Diese Klasse ist auf allen getesteten Plattformen, bis auf Python QNX und Windows CE, vorhanden. Dies lässt darauf schließen, dass auf den beiden zuletzt genannten Systemen keine Nutzung von AWT unter SWT und somit auch keine Swing-Einbindung möglich ist.

- `org.eclipse.swt.browser.Browser`
  Das Browser-Objekt ist unter Windows CE nicht verfügbar.

- `org.eclipse.swt.dnd.Clipboard`
  Ein Zugriff auf die Zwischenablage ist zwar unter allen Systemen möglich, allerdings fehlt unter Windows CE die Methode `clearContents()`.

- `org.eclipse.swt.widgets.DateTime`
  Python QNX scheint diese Klasse nicht implementiert zu haben. Dadurch steht dort anscheinend kein natives Eingabefeld für Datumsangaben zur Verfügung.

Die Tabelle 3.1 zeigt eine komplette Auswertung anhand der vorliegenden Zahlen zu gefundenen beziehungsweise nicht gefundenen Klassen und Methoden pro SWT-Bibliothek. Bei einer Auswertung der Übersicht fällt besonders auf, dass die Umsetzung von SWT für Windows CE anscheinend nicht komplett durchgeführt wurde. SWT bietet demnach nicht auf allen Plattformen denselben Funktionsumfang an.

3.2 Untersuchung spezieller Methoden

Die vorherige Betrachtung zeigt bereits einige Diskrepanzen unter den einzelnen Varianten der SWT-Bibliothek, bezugnehmend auf das Vorhandensein von Klassen und Methoden. Dies kann unter anderem auf den speziellen Einsatzort der einzelnen SWT-Bibliotheken zurückzuführen sein. So wurde zum Beispiel bei der SWT-Bibliothek für Windows CE bewusst auf die Drag and Drop Klassen verzichtet. Daraus resultierend konnten hier dringend benötigte Ressourcen freigegeben werden.

Eine genauere Analyse der einzelnen Methoden von SWT würde den Rahmen der Arbeit sprengen und soll auch nicht zentraler Bestandteil der Arbeit sein.

SWT-Bibliothek	Klassen			Methoden		
	gesamt	fehlen	%	gesamt	fehlen	%
carbon-macosx	294	12	4,1%	9933	612	6,2%
cocoa-macosx	294	12	4,1%	9933	612	6,2%
cocoa-macosx-x86_64	294	12	4,1%	9933	612	6,2%
gtk-aix-ppc64	294	11	3,7%	9933	564	5,7%
gtk-linux-ppc	294	11	3,7%	9933	564	5,7%
gtk-linux-ppc64	294	11	3,7%	9933	564	5,7%
gtk-linux-x86	294	11	3,7%	9933	564	5,7%
gtk-linux-x86_64	294	11	3,7%	9933	564	5,7%
gtk-solaris-sparc	294	11	3,7%	9933	564	5,7%
gtk-solaris-x86	294	11	3,7%	9933	564	5,7%
motif-aix-ppc	294	12	4,1%	9933	612	6,2%
motif-hpux-ia64_32	294	12	4,1%	9933	612	6,2%
motif-linux-x86	294	12	4,1%	9933	612	6,2%
motif-solaris-sparc	294	12	4,1%	9933	612	6,2%
photon-qnx-x86	294	14	4,8%	9933	776	7,8%
win32-wce_ppc-arm-j2me	294	85	28,9%	9933	3371	33,9%
win32-wce_ppc-arm-j2se	294	85	28,9%	9933	3371	33,9%
win32-win32-x86	294	1	0,3%	9933	54	0,5%
win32-win32-x86_64	294	1	0,3%	9933	54	0,5%

Tabelle 3.1: Absolute und relative Anteile von vorhandenen Differenzen in den verschiedenen SWT Umsetzungen

Dennoch soll kurz darauf bezugnehmend auf einen weiteren Unterschied eingegangen werden. Es lassen sich an mehreren Stellen im SWT Source Code Kommentare der Entwickler finden, die darauf schließen, dass einige Funktionen für bestimmte Plattformen nicht implementiert sind oder auf dem Wirtssystem nicht zur Verfügung stehen.

Beispiel: Hintergrundfarbe setzen

Mit der Image-Klasse soll es möglich sein, primitive Zeichenoperationen durchzuführen. Dazu zählt unter anderem das Zeichnen von Linien und Kreisen, das Setzen der Linienstärke und das Einfärben von Elementen. Die Image Klasse kann zum Beispiel dazu verwendet werden, Spezial-Widgets zu realisieren, Diagramme zu zeichnen oder Bilder und Icons zu erzeugen.

Das folgende Listing 3.1 stammt aus der Klasse `org.eclipse.swt.graphics.Image.java` der SWT-Bibliothek für Windows (x86_64).

Listing 3.1: Beispiel einer vom Betriebssystem abhängigen nicht implementierten SWT-Methode

```
1 public void setBackground(Color color) {
2   /*
3    * Note. Not implemented on WinCE.
4    */
5   if (OS.IsWinCE) return;
6     ...
```

Das Setzen der Hintergrundfarbe beim Zeichnen mit einem Image-Objekt wird unter Windows CE also offensichtlich nicht unterstützt. Dies ist ein Punkt, den ein GUI-Designer beachten muss, wenn die spätere Oberfläche auf Windows CE portiert werden soll.

Beispiel: Status einer Fortschrittsanzeige setzen

Seit SWT Version 3.4 gibt es die Möglichkeit, einer Fortschrittsanzeige einen Status zu setzen. Das folgende Listing 3.2 stammt aus der Klasse `ProgressBar` der SWT-Bibliothek für Linux (x86/Motif).

Listing 3.2: Beispiel einer nicht implementierten SWT-Methode

```
1 public void setState (int state) {
2   checkWidget ();
3   //NOT IMPLEMENTED
4 }
```

Dieser kann die Werte `SWT.NORMAL`, `SWT.ERROR` und `SWT.PAUSED` annehmen. Zum Vergleich soll in der Tabelle 3.2 die Auswirkung unter Linux und unter Windows gezeigt werden.

	SWT.NORMAL	SWT.ERROR	SWT.PAUSED
Linux			
Windows			

Tabelle 3.2: Unterschiedliche Darstellungen des Status einer Fortschrittsanzeige unter Ubuntu 12.04 LTS und Windows Vista

Auch hier muss ein GUI Designer von den unterschiedlichen Umsetzungen auf den Systemen achten und dies in seine Implementierung mit einfließen lassen. Dieser Umstand ist auf die historische Entwicklung der einzelnen Betriebssysteme zurückzuführen. Auf den Entwicklerseiten von Microsoft, dem Microsoft Development Network, ist nachzulesen, dass die Minimalvoraussetzung für das Setzen des Status einer Fortschrittsanzeige Windows Vista ist [1]. Dies lässt darauf schließen, dass diese Funktionalität erst mit Windows Vista, also am Anfang

[1] Microsoft Development Network „PBM_SETSTATE message“ `http://msdn.microsoft.com/en-us/library/windows/desktop/bb760850%28v=vs.85%29.aspx` (Stand März 2013)

2007, eingeführt wurde. Andere Betriebssysteme unterstützen derartige Statusangaben nicht. Durch die direkte Bindung von SWT an das Wirtsbetriebssystem kann hier also keine hundertprozentige Übereinstimmung erreicht werden.

3.3 Funktionsumfang spezieller Methoden

Eine andere Art der Differenz zeigt sich erst beim praktischen Einsatz von SWT auf unterschiedlichen Plattformen. Diese ist von den vorgestellten Punkten am schwersten zu erkennen. Hier bedarf es viel Hintergrundwissen des Entwicklers, da sich die Auswirkungen erst im laufenden System unter unterschiedlichen Bedingungen zeigen können. Die Rede ist von Methoden, bei denen das erwartete Resultat unter unterschiedlichen Betriebssystemen teilweise oder ganz und gar verschieden ist.

Beispiel: Widget als Grafik kopieren

Ein Beispiel hierfür ist die Methode `print(GC gc)` der Klasse `Control`, der zentralen Klasse, von der alle SWT-Widgets erben. Diese wird dafür benutzt, das Widget in dem übergebenen grafischen Kontext rekursiv zu zeichnen. Es wird also ein komplettes Abbild in eine Grafik übertragen.

Die Abbildung 3.1 zeigt das Resultat einer Widget-Kopie mittels der vorgestellten Methode unter Windows und Linux. Dabei wurde eine Schaltfläche samt Beschriftung in eine rot umrandete Zeichenfläche kopiert. Hier ist zu sehen, dass das Resultat unter Linux nicht das erwartete Ergebnis liefert.

Abbildung 3.1: Unterschiedliche Resultate der Methode print(GC gc) unter SWT für Windows und Linux

Kapitel 4

4 Lösungsansätze für einheitliche SWT Varianten

In den nächsten Kapiteln sollen unterschiedliche Lösungsansätze vorgestellt werden, wie die vorhandenen Differenzen in den SWT-Bibliotheken umgangen oder sogar behoben werden können. Die einzelnen Methoden bedienen sich unterschiedlicher Konzepte, die zur Lösung dieses Problems beitragen können. Zur Verdeutlichung werden die jeweiligen Methoden vorgestellt, ein Lösungsansatz anhand eines Beispiels veranschaulicht dargestellt und im Anschluss Vor- und Nachteile benannt. Die Vor- und Nachteile werden sich auf allgemeine Programmiertechniken beziehen.

Zunächst werden zwei Methoden beschrieben, die nativ von der Programmiersprache Java unterstützt werden. Dazu zählt zum einen ein Precompiler-Konstrukt, welches mit wenigen Codezeilen umgesetzt werden kann und zum anderen die Verwendung von Interfaces, um vorhandene Lösungen programmierfreundlich auszulagern. Im weiteren Verlauf werden zwei Ansätze vorgestellt, die die Nutzung von Drittprodukten erfordert. Darunter fallen Präprozessoren und Build-Scripte, die beide einen vergleichbaren Ansatz verfolgen. Abschließend wird in diesem Kapitel verdeutlicht, wie eine aspektorientierte Lösungsmethode zur Bereinigung des Problems dienen kann.

4.1 Precompiler in Java

Entwickler, die Software in C oder C++ schreibt, kennt automatisch auch die Precompiler-Befehle. Diese können sich zwar von Compiler zu Compiler unterscheiden, ein Basissatz ist allerdings immer gleich. Dazu zählen unter anderem die #ifdef Befehle. Mit #ifdef kann ein Entwickler den Compiler dazu veranlassen, bestimmte Codeabschnitte bei der Kompilierung zu vernachlässigen. Dies kann zum Beispiel nützlich sein, um Debug-Informationen aus dem endgültigen Code herauszufiltern beziehungsweise diese nur bei Bedarf zu aktivieren.

Java bietet von Haus aus keinen Precompiler an. Es gibt allerdings die Möglichkeit, die Funktionsweise von #ifdef mit einfachen Mitteln, wie der Ausnutzung der sogenannten *Unreachable-Code Elimination* [Debray et al., 2000], nachzubau-

en. Durch diesen Nachbau wird der Java-Compiler dazu gezwungen, bestimmte Codeabschnitte nicht in den Bytecode zu übersetzen. Zur Veranschaulichung soll das Listing 4.1 dienen.

Listing 4.1: Beispiel einer Precompiler-Umsetzung vor der Kompilierung in Java

```
1 public static final boolean DEBUG_MODE = true;
2
3 public static void ifdef() {
4   if (DEBUG_MODE) {
5     System.out.println("Running␣in␣debug␣mode.");
6   } else {
7     System.out.println("Running␣in␣normal␣mode.");
8   }
9 }
```

Durch die Deklarierung der Variable `DEBUG_MODE` als final, weiß der Compiler, dass sich der Wert dieser Variablen nicht mehr ändern kann. Daher überspringt er den entsprechenden Teil der if-Bedingung automatisch und kompiliert diesen nicht mit. Dies ist ein Teil der automatischen Codeoptimierung des Java-Compilers [Gosling et al., 2013]. Der kompilierte Code aus dem vorherigen Listing 4.1 würde dem Listing 4.2 entsprechen.

Listing 4.2: Beispiel einer Precompiler-Umsetzung nach der Kompilierung in Java

```
1 public static void ifdef() {
2   System.out.println("Running␣in␣debug␣mode.");
3 }
```

Wäre die Variable nicht final, würde die Abfrage ganz normal übernommen und bei jedem Aufruf ausgewertet werden.

SWT Beispiel

Dieses Konstrukt kann nun dabei helfen, zum Beispiel nicht implementierte Funktionen in einzelnen SWT-Bibliotheken nachzuliefern. Zur Veranschaulichung soll der bereits beschriebenen Missstand einbezogen werden, dass SWT unter Linux keine funktionierende Methode `setState` anbietet (siehe Listing 3.2). Diese liegt lediglich mit dem Kommentar `NOT IMPLEMENTED` vor. Eine mögliche Lösung ist in Listing 4.3 zu sehen ist.

Listing 4.3: Umsetzung der Fortschrittsanzeige mit Precompiler-Konstrukten

```
01 public static final boolean IS_IMPLEMENTED = false;
02
03 public static void setStateOfProgressBar(ProgressBar pB, int state) {
04   if (IS_IMPLEMENTED) {
05     pB.setState(state);
06   } else {
07     Display d = pB.getDisplay();
08     switch (state) {
09       case SWT.ERROR:
10         pB.setForeground(new Color(d, 255, 0, 0));
11         break;
12
13       case SWT.PAUSED:
14         pB.setForeground(new Color(d, 255, 255, 0));
15         break;
16
17       default:
18         pB.setForeground(new Color(d, 0, 0, 0));
19     }
20   }
21 }
```

Hierzu wird zunächst mittels der finalen Variable `IS_IMPLEMENTED` definiert, ob die Funktionalität unter dem gewünschten Zielbetriebssystem zur Verfügung steht oder nicht (Zeile 01). In der Methode `setStateOfProgressBar` wird diese anschließend ausgewertet und abhängig des Wertes die native Variante (Zeile 05) oder ein Workaround (Zeilen 07 bis 19) ausgeführt. Der Workaround simuliert in diesem Fall nur das gewünschte Verhalten, indem es die Vordergrundfarbe abhängig vom zu setzenden Status übernimmt. Für den Fall, dass das Betriebssystem das Setzten des Status nicht unterstützt, muss allerdings auch auf Folgendes geachtet werden; Da der Status im Workaround lediglich die Farben ändert, das Widget sonst jedoch komplett unangetastet bleibt, wird die Methode `getStatus` keinen brauchbaren Wert (unter Linux immer `SWT.NORMAL`) zurückliefern. Das bedeutet, dass Anpassungen mit Precompilern unter Umständen noch weitere Stellen im Quelltext betreffen können, die auf den ersten Blick nicht unbedingt ersichtlich sein müssen.

Vor- und Nachteile von Precompiler in Java

Der oben beschriebene Nachbau von Precompilern in Java mittels finalen Variablen bedient sich der Ausnutzung von Compiler Optimierungen. Dabei werden Codefragmente, die unter keinen Umständen ausgeführt werden können, von vornherein vom Java-Compiler ignoriert. Dadurch entstehen im späteren ByteCode keine „Codeleichen“, die die eigentliche Ausführung der Anwendung verlangsamen würden.

Zwar können derartige Workarounds in statische Klassen ausgelagert und wo nötig angewendet werden. Allerdings muss der Entwickler selbst darauf achten, diese an jeder Stelle im eigenen Quelltext anzuwenden. Die finalen Variablen müssen vor jeder Kompilierung für ein neues Zielsystem umgestellt werden, so dass auch hier ein gewisser, wenn auch geringer Mehraufwand zu verzeichnen ist.

4.2 Interfaces

Die nächste hauseigene Methode könnte sein, Codeanpassungen für unterschiedliche Bereiche der SWT-Bibliotheken in plattformspezifische Klassen auszulagern und mittels Interface an betroffenen Stellen zu importieren. Ein Interface spezifiziert eine Schnittstelle im eigentlichen Sinne, dessen in ihr beschriebene abstrakte Methoden von jeder Klasse, die diese verwendet, implementiert werden muss. Interfaces können somit nicht instanziiert, jedoch implementiert werden. Sie bilden also eine Abstraktionsschicht, die die Verwendung durch den Entwickler und eine konkrete Implementierung voneinander trennen.

Die einzelnen zur Verfügung gestellten SWT-Bibliotheken arbeiten selbst mit Interfaces. So braucht der Entwickler nicht zu wissen, wie zum Beispiel ein SWT-Textfeld unter Windows erzeugt wird, sondern nur, dass es ein SWT-Textfeld gibt, das auf eine bestimmte Art und Weise erzeugt werden kann. Die konkrete Vorgehensweise regelt später die eingebundene SWT-Bibliothek für Windows.

Vor- und Nachteile von Interfaces

Die Verwendung von Interfaces bietet sich daher an, weil eine Korrektur nur einmalig implementiert werden muss und an einer zentralen Stelle abgelegt werden kann. Bei der Kompilierung beziehungsweise Auslieferung des finalen Produktes muss nur noch die entsprechende Bibliothek für die Zielplattform beigepackt werden. Da die Vorgehensweise bei den SWT-Bibliotheken äquivalent ist, würde es sich anbieten, die Korrekturen gleich zu den Bibliotheken zu packen. Es muss hier bei der finalen Kompilierung nicht mehr darauf geachtet werden, für welches Zielsystem das entsprechende Produkt generiert wird.

Ein Nachteil bei dieser Vorgehensweise bleibt aber, wie bei dem zuvor beschriebenen Precompiler-Ansatz, bestehen. Der Entwickler muss die Eigenheiten der einzelnen SWT-Bibliotheken kennen und an den entsprechenden Stellen im eigenen Quelltext Bausteine aus den Korrekturbibliotheken anwenden. Dieser Umstand kann durch ein Interface, welches sich strukturell an SWT anlehnt, weitestgehend kompensiert werden. Der Artikel [Bartolomei et al., 2010] greift dieses Thema im Rahmen eines API-Mappings auf und zeigt dazu Anforderungen und Probleme, die bei einer Umsetzung auftreten können.

4.3 Präprozessoren

Eine ähnliche Herangehensweise zur automatischen Codeoptimierung bei finalen Variablen, nutzen externe Präprozessoren oder „Versions-Konfigurierer“. Sie bedienen sich einer speziellen Syntax, die in den Kommentaren der einzelnen Quelltext-Dateien hinterlegt sind. Präprozessoren können prinzipiell auf jede Art von Textdatei angewendet werden, eignen sich jedoch besonders für Quelltexte. Preprozessoren werden hauptsächlich in C/C++ verwendet und sind dort schon lange etabliert [Ernst et al., 2002]. Zwei Präprozessoren für die Programmiersprache Java sind *Preprocessor-Java* und *Prebop*. Mit ihnen wird eine einfache

Bedingung geschaffen, mit der unterschiedliche Softwareversionen aus nur einer Codequelle entstehen können.

Preprocessor-Java bedient sich dabei der bekannten Syntax aus C/C++ Zeiten. Die Befehle werden in den Kommentaren des Java-Codes hinterlegt. Einzige Voraussetzung ist, dass die Programmiersprache eine Zeichensequenz besitzt, um eine einzelne Zeile Code auszukommentieren. In Java ist dies `//`. Zur Erzeugung und Initialisierung von Variablen muss `#define <Variablenname> <Wert>` aufgerufen werden. Unterstützung finden nach Herstellerangaben die Datentypen `Boolean`, `Integer`, `Float` und `String`. Anschließend können im späteren Verlauf nun Abfragen mittels `#ifdef <Ausdruck>` ausgeführt werden. Das Listing 4.4 soll die Vorgehensweise verdeutlichen.

Listing 4.4: Einfaches Beispiel einer Codeumsetzung mit *Preprocessor-Java*

```
1 public static void preprocessorJava() {
2   // #define BOOL_VALUE True
3   // #ifdef BOOL_VALUE
4   System.out.println("value is true");
5   // #else
6   System.out.println("value is false");
7   // #endif
8 }
```

Die Initialisierung der Variablen `BOOL_VALUE` erfolgt in Zeile 02. Gleich im Anschluss wird dieser Wert abgefragt und abhängig davon ein entsprechender Text auf der Konsole ausgegeben (Zeilen 03 bis 07). Würde dieser Quelltext unverändert von Java kompiliert und ausgeführt werden, würden beide Ausgaben erscheinen. Wie eingangs beschrieben, muss vor dem eigentlichen Kompilierungsvorgang *Preprocessor-Java* die Quelldatei einlesen, interpretieren und umwandeln. Dies geschieht mit dem Konsolenbefehl:

```
java -jar javapc_0.1.jar -s "src/PreprocessorJava.java"-d "src_processed"
```

Das sorgt nun dafür, dass eine umgewandelte Version der Datei im Ordner *src_processed* abgelegt wird. Im Resultat wird lediglich die Zeile 06 auskommentiert. Sobald Java nun diese Datei kompiliert, erhalten wir auch das gewünschte Resultat.

Prebop geht einen vergleichbaren Weg, jedoch mit einer modifizierten Syntax. *Prebop* steht nicht wie *Preprocessor-Java* als Standalone-Version zur Verfügung. *Prebop* wird als Ant-Task einem bestehenden Build-Prozess vorangestellt. Das Buildwerkzeug *Ant* wird im Abschnitt 4.4 näher betrachtet. Abweichend zum vorherigen Präprozessor werden die verwendeten Variablen zentral im Ant-Task definiert. Ein Beispiel, welches die Syntax demonstrieren soll, ist in Listing 4.5 abgebildet.

Listing 4.5: Einfaches Beispiel einer Codeumsetzung mit *Prebop*

```
1 public static void prebop() {
2   /* $if version < 0.1 $ */
3   System.out.println("alpha version");
4   /* $elseif version < 1.0 $ */
5   System.out.println("beta version");
6   /* $else $ */
7   System.out.println("released version");
8   /* $endif $ */
9 }
```

Ziel des Codeausschnitts ist es, eine definierte Konsolenausgabe zu erzeugen, die abhängig von einer zuvor definierten Versionsnummer ist. Die entsprechenden Kontrollstrukturen sind dafür in den Kommentaren in den Zeilen 02, 04, 06 und 08 hinterlegt.

Vor- und Nachteile von Präprozessoren

Da Präprozessoren in der Regel lediglich nicht benötigte Bereiche auskommentieren, bleiben die Relationen und Zeilenangaben beim Debuggen erhalten. Zeilenangaben in Exceptions beziehen sich dementsprechend noch auf die Zeilen im Ursprungsquelltext.

Ein erster Nachteil zeigt sich allerdings bei größeren Codeblöcken mit konkurrierenden Codezeilen. Als konkurrierende Codezeilen seien hier Passagen im Quelltext zu sehen, die in der Ursprungsfassung vom Compiler als Fehler interpretiert werden würde. Entwicklungsumgebungen wie *Eclipse* würden den Entwickler fälschlicherweise auf diesen Missstand hinweisen. Das Listing 4.6 soll dies verdeutlichen.

Listing 4.6: Konkurrierende Codezeilen im Umfeld eines Präprozessors

```
1 public static void preprocesor_disadvantage() {
2   // #define BOOL_VALUE True
3   // #ifdef BOOL_VALUE
4   int x = 0;
5   // #else
6   int x = 1;
7   // #endif
8 }
```

Hier wird in Zeile 04 eine Variable mit dem Namen `x` und dem Wert 0 deklariert. In Zeile 06 wird versucht eine weitere Variable mit identischem Namen und abweichenden Wert zu deklarieren. An dieser Stelle würde eine Entwicklungsumgebung wie *Eclipse* Alarm schlagen, da dieser Variablenname bereits vergeben wurde. Nach der Interpretation des eingesetzten Präprozessors würde die Zeile 06 auskommentiert werden, so dass im Anschluss ein gültiger Quelltext entsteht. Für diesen Umstand können Workarounds eingesetzt werden. So könnte die Variable `x` einmalig deklariert und im relevanten Teil des Präprozessors initialisiert werden. Das Listing 4.7 zeigt dafür eine mögliche Umsetzung.

Listing 4.7: Workaround für das Problem der konkurrierenden Codezeilen

```
1 public static void preprocesor_disadvantage() {
2   int x;
3   // #define BOOL_VALUE True
4   // #ifdef BOOL_VALUE
5   x = 0;
6   // #else
7   x = 1;
8   // #endif
9 }
```

Allerdings ist zu beachten, dass Workarounds einerseits den Umfang von nötigen Quellcodezeilen erhöhen und zudem die Les- und Wartbarkeit des Quelltextes negativ beeinflussen können.

Ein weiterer Nachteil ergibt sich aus automatischen Codeformatierungen, wie es von vielen Entwicklungsumgebungen angeboten wird. Dabei könnten mehrere einzeilige Kommentare zu einem mehrzeiligen Kommentar zusammengefasst werden. *Preprocessor-Java* benötigt allerdings die einzeiligen Kommentare, in dem relevante Befehle hinterlegt sein können. Bei den automatischen Codeformatierungen kann es also unter Umständen passieren, dass externe Präprozessoren den Code nicht mehr fehlerfrei interpretieren können.

4.4 Buildscripte

Um zunächst einen groben Überblick über das Build-Management-Tool *Ant* zu erhalten, ist es notwendig, den Begriff des „Build-Management“ näher zu erläutern. Darunter wird ein Werkzeug verstanden, das den zuvor geschriebenen Quelltext mit möglichst geringem Aufwand in eine Zielsprache übersetzt. Gerade bei komplexerem Sourcecode, welcher sich nicht nur auf einige wenige Dateien und Klassen beschränkt, ist das effiziente Kompilieren eine große Herausforderung. So können zwischen unterschiedlichen Dateien Abhängigkeiten bestehen. Zur Veranschaulichung soll eine Klasse dienen, die an mehreren Stellen im Projekt eingebunden ist. Sobald ein Entwickler eine enthaltene Methode ändert, müssen sämtliche damit verbundenen Klassen ebenfalls neu übersetzt, beziehungsweise die Korrektheit der Einbindung überprüft werden. Das Projekt könnte alternativ auch komplette neu kompilieren werden. Dieses Vorgehen mag bei kleinen Projekten noch praktikabel sein, stößt aber an seine Grenzen, sobald diese eine gewisse Größe überschreiten, Stichwort Kernel Kompilierung. Aus diesem Grund sollte ein gutes Build-Management-Tool in der Lage sein, Abhängigkeiten zu erkennen und nur diese neu zu erstellen.

Bereits in den 70er Jahren wurde *make* von Stuart Feldman entwickelt [Feldman, 1979], welches als Vorreiter der automatischen Build-Management-Tools angesehen werden kann. *Make* ist dabei so mächtig, dass es nicht nur für die Unterstützung bei der Softwareentwicklung, sondern auch bei anderen untereinander abhängigen Prozessen eingesetzt wird, bei denen Dateien erzeugt, kopiert, modifiziert oder gelöscht werden. Eine Alternative zu *make* stellt Apache *Ant* dar, was ausgeschrieben für „Another Neat Tool“ steht. *Ant* wurde 1998 von James

Duncan Davidson zunächst als Build-Management-Tool für den Tomcat Server entwickelt. Als man erkannte, dass das Tool viele weitere Probleme bei der automatischen Erzeugung von Projekten beheben würde, wurde es aus dem Tomcat Server entkoppelt und als eigenständiges Open Source Projekt weitergeführt.

Um verschiedene Ziele für ein Projekt verfolgen zu können, wie zum Beispiel das Übersetzten der Quelltexte für verschiedene Zielplattformen, werden in der Ant Konfiguration (in der Regel *build.xml*) verschiedene Targets definiert. Zunächst wird das *default target* aufgerufen, dem, je nach Konfiguration, weitere *Targets* folgen können. Jedes *Target* beinhaltet dabei verschieden viele Arbeitsschritte, sogenannte *Tasks*, wie zum Beispiel das Erzeugen von Ordnern, Löschen, Modifizieren oder Kompilieren von Dateien. Bei jedem Vorgang werden die konfigurierten Abhängigkeiten überprüft. So wird zum Beispiel vor dem Kompilierungsvorgang überprüft, ob die Dateien bereits kompiliert wurden. Existieren diese bereits und sind jünger als die abhängigen Dateien, wird der Kompilierungsvorgang übersprungen, da in diesem Fall die gewünschten Dateien bereits in der neuesten Version vorliegen.

Im Gegensatz zu *make* bietet *Ant* hier den großen Vorteil, dass grundlegende Tasks bereits plattformunabhängig vorimplementiert sind und mit dem Tool mitgeliefert werden. Zu den mehr als 150 build-in Tasks zählen unter anderem die in Tabelle 4.1 aufgezählten Befehle. Diese können ohne größeren Aufwand durch selbst geschriebene Tasks erweitert werden.

Befehl	**Kurzbeschreibung**
javac	Kompilieren von Java Quellcode
copy	Kopieren von Dateien aus einem Quell- in einen Zielordner
delete	Löschen von Dateien oder Verzeichnissen
mkdir	Erstellen von Verzeichnissen
junit	Automatisierte JUnit-Tests
replace	Ersetzen von Text in Dateien

Tabelle 4.1: Auswahl häufig benutzter Befehle in Apache Ant Build-Scripten

SWT Beispiel

Zur Veranschaulichung soll dieses Mal das nicht vorhandene Widget `DateTime` unter Python QNX herangezogen werden. Hierbei ist zu beachten, dass im Gegensatz zu dem vorhergegangenen Beispiel nicht nur einzelne Methoden fehlen beziehungsweise nicht ausimplementiert sind, sondern das gesamte Widget fehlt. Als Workaround kann ein normales Textfeld so modifiziert werden, dass es lediglich Eingaben in einem vorher festgelegten Datumsformat zulässt. Dazu wird dem Textfeld ein `VerifyListener` hinzugefügt, der die Eingaben des Nutzers nach jeder Änderung prüft. Da die Entwicklung eines DateTime-Widgets unter Python nicht Bestandteil dieser Arbeit sein soll, wird an dieser Stelle auf detailliertere Angaben verzichtet. Das Listing 4.8 zeigt dazu eine mögliche Java-Umsetzung.

Listing 4.8: DateTime-Widget unter Python QNX unter Zuhilfenahme von Apache Ant für Java

```
1  public static void createDateTimeWidget(Composite parent, int style) {
2    // @SNP@
3    DateTime dateTime = new DateTime(parent, style);
4    // @ENP@
5
6    // @SP@
7    Text text = new Text(parent, style);
8    text.addVerifyListener(...);
9    // @EP@
10 }
```

Wie bei der Präprozessor-Methode zuvor werden die betriebssystemrelevanten Abschnitte mit speziellen „Codewörtern“, in diesem Fall speziell formatierte Kommentare (Zeile 02, 04, 06 und 08), umschlossen. Auf diese kann das später laufende Ant-Script dann entsprechend reagieren.

Der relevante Teil der Build-Konfiguration für Ant kann die gesetzten Codewörter durch simple einzeilige oder mehrzeilige Kommentare ersetzen, um damit nur noch die plattformrelevanten Abschnitte zu belassen.

Listing 4.9: Buildscript für DateTime-Widget unter Python QNX

```
1  <target name="build_python">
2    <copy todir="../out" includeEmptyDirs="false">
3      <filterchain>
4        <tokenfilter>
5          <replacestring from="//␣@SP@" to="//" />
6          <replacestring from="//␣@EP@" to="//" />
7          <replacestring from="//␣@SNP@" to="/*" />
8          <replacestring from="//␣@ENP@" to="*/" />
9        </tokenfilter>
10     </filterchain>
11     <fileset dir=".">
12       <include name="**/*.java" />
13     </fileset>
14   </copy>
15
16   <javac srcdir="../out" />
17 </target>
```

In dem benannten Target `build_python` im Listing 4.9, welches ein Target für die Plattform Python darstellen soll, werden zunächst alle Java-Dateien aus dem Projekt in den Ordner *../out* kopiert. Dabei werden simple Zeichenkettenersetzungen verwendet, um *// @SP@* und *// @EP@* mit einzeiligen Kommentaren (*//*) und *// @SNP@* und *// @ENP@* mit mehrzeiligen Kommentaren (*/** und **/*) zu ersetzen. In der Zeile 16 wird der eigentliche Kompilierungsprozess auf den modifizierten Dateien angestoßen.

Nach diesem Ant Task sind die Zeilen 02 bis 04 des Listings 4.8 also mit mehrzeiligen Kommentaren umgeben, wobei die Zeilen 06 und 09 durch einfache leere Kommentare ersetzt wurden.

Vor- und Nachteile von Build-Management Tools

Build-Management Tools wie Ant werden besonders häufig bei größeren Projekten eingesetzt. Da in diesen Fällen bereits Ant Konfigurationen existieren, sollte es

keine großen Umstände bereiten, die entsprechenden Passagen anzupassen und zu erweitern.

Da allerdings bei Build-Management Tools wie bei den Präprozessoren eine simple Ersetzung von Zeichenketten durchgeführt wird, gelten hier dieselben Nachteile, wie zum Beispiel das Problem bei konkurrierenden Codezeilen und automatischen Codeformatierungen in Entwicklungsumgebungen.

Aufgrund der Tatsache, dass die Ersetzungen nicht auf Kommentarbereiche beschränkt sind, können durch eine unsaubere Handhabung zusätzliche Fehlerquellen entstehen. Spezielle Codewörter, die eigentlich nur für das Buildscript vorgesehen sind, können ebenso im Quelltext auftauchen. Diese würden bei einer Ausführung des Buildscripts nicht vorgesehene Bereiche im Quelltext modifizieren und so (im günstigsten Fall) einen Kompilierungsfehler erzeugen.

4.5 Der aspektorientierte Ansatz

Ein entscheidender Nachteil bei allen zuvor vorgestellten Verfahren ist, dass alle Stellen in den Quelltexten lokalisiert werden müssen, bei denen ein fehlendes Widget oder eine nicht implementierte Methode verwendet wird. Diese muss anschließend an allen Fundstellen behoben, korrigiert oder mittels Workaround umgangen werden. Daraus resultiert ein nicht abzuschätzender Mehraufwand für den Entwickler. Unter Umständen müssten sogar fremde Quelltexte und Bibliotheken angepasst werden, was abhängig von der verwendeten Sichtbarkeit nicht immer möglich ist.

Ein komplett anderer Weg kann mit der aspektorientierten Programmierung beschritten werden.

Einführung

Bei der Entwicklung von komplexen Systemen bedient man sich heutzutage dem Prinzip „Teile und Herrsche“. Dabei wird versucht, ein komplexes Produkt in viele einzeln lösbare, von anderen Bestandteilen unabhängige Teilaufgaben zu zerlegen, die zusammengesetzt im Nachhinein wieder das komplette Produkt abbilden [Allen, 1997]. Diese Teilaufgaben können als Module bezeichnet werden. Mit den heutzutage oft eingesetzten objektorientierten Programmierverfahren bieten sich diesbezüglich viele Möglichkeiten an. Auf Grund der dynamischen Polymorphie können Abhängigkeiten so verallgemeinert werden, dass diese nur noch von abstrakten Schnittstellen abhängig sind. Ein oft und an mehreren Stellen verwendeter Code kann so in wiederverwendbare Module ausgelagert werden. Im Idealfall ist ein Modul lediglich über eine abstrakte Schnittstelle ansprechbar und nur für die ihm anvertraute Aufgabe zuständig. Bei der praktischen Anwendung der reinen Objektorientierung ist dies bei näherer Betrachtung allerdings nicht möglich.

Code Tangling und Code Scattering

Um dies etwas deutlicher veranschaulichen zu können, soll an dieser Stelle das Beispiel herangezogen werden, welches gern in Einsteiger-Tutorials verwendet wird, wenn es um das Erlernen von Objektorientierung geht, die Klasse `Auto`. Das Relevante Exemplar der Klasse soll allerdings nur zwei primäre Funktionen haben, nämlich das Setzen der Farbe und der Anzahl der Türen. In einem idealen System würden diese Funktionen den einfachen Zweck erfüllen, die Lackfarbe und die Türanzahl des Autos zu setzen. In der Praxis sieht das meist anders aus. Hier müssen sich die Funktionen selbst neben ihrer eigentlicher Primäraufgabe noch um andere Dinge kümmern. In sicherheitsrelevanten Systemen müsste zum Beispiel überprüft werden, ob derjenige, der die Funktionen nutzen will, auch die Berechtigung hat, dies zu tun. Zum Beispiel kann nur ein Fahrzeugdesigner die Anzahl der Türen setzen, nicht jedoch der spätere Kunde. Hier kann zwar die Überprüfung in ein Modul ausgelagert werden, der eigentliche Methodenaufruf muss jedoch in den einzelnen Funktionen geschehen. Die Anforderung lässt sich bei der Objektorientierung nicht zentral einer Stelle zuordnen.

Eine spätere Erweiterung der Anforderung könnte lauten, dass alle Änderungen mit Namen und Zeitstempel protokolliert werden müssen. Im Idealfall muss hierfür nur das eingebundene Modul, welches die Sicherheitsüberprüfung durchführt, angepasst werden. Ungünstigenfalls betrifft dies auch jeden Methodenaufruf, der sich auf diese Überprüfung bezieht. Was ist jedoch, wenn Teile der Autoklasse von einem anderen Entwickler stammen, auf dessen Sourcecode kein Zugriff möglich ist? Diese Vermischung von unterschiedlichen Anforderungen in einem Modul wird als *Code Tangling* bezeichnet. *Code Tangling* vermindert dabei die Wiederverwendbarkeit von Code beziehungsweise Modulen, wie auch die Lesbarkeit, da die elementare Anforderung eines Moduls mit anderen Anliegen vermischt wird.

Die erwähnte Sicherheitsüberprüfung liegt dabei über mehrere Klassen verstreut. Abgesehen von der Klasse `Auto` kann sie noch an vielen anderen Stellen Verwendung finden, wie zum Beispiel in Klassen, die von der Klasse `Auto` erben. Diese Redundanz von Code wird auch als *Code Scattering* bezeichnet. Hier gibt es mehrere Module, die Verantwortung für das gleiche Anliegen tragen. Herauszufinden, an welcher Stelle die eigentliche Funktionalität umgesetzt wird, ist dadurch erschwert.

Lösung durch Aspektorientierung

Ein zentrales Prinzip des Software-Engineerings ist das von Dijkstra 1976 eingeführte „Separation of Concerns“ [Dijkstra, 1976] was allgemein eine Trennung der Anliegen bedeutet. Um dieses in der objektorientierten Programmierung umzusetzen, können aspektorientierte Mechanismen verwendet werden [Kiczales et al., 1997]. Sie erweitern die Sprache um definierte Verfahren, mit denen im gewissen Maße in die Struktur von Programmen eingegriffen werden kann. Die Struktur von Programmen beschreiben in diesem Kontext Dinge wie, welche Klassen existieren, welche Methoden mit welchen Parametern diese haben oder mit welchen Modifikatoren diese belegt sind. Abgrenzbare Anliegen, wie das oben angespro-

chene Sicherheitskonzept, können dabei kompakt und zentral implementiert werden. Auch wenn sie sich auf mehrere Objekte unterschiedlicher Klassen beziehen. In diesem Fall werden die Anliegen *Crosscutting Concerns* genannt. Hierbei werden die entsprechenden Codeabschnitt und -aufrufe automatisiert mittels der geschriebenen Aspekte in die Struktur des Programms eingewoben. Im Beispiel der Klasse `Auto` ist das Resultat nach dem Einweben identisch. Vor den Aufrufen der Methoden, um die Farbe oder Anzahl der Türen zu setzen, wird die anliegenfremde Sicherheitsprüfung durchgeführt. Allerdings ist der Code nun einmalig zentral definiert und von der Klasse entbunden, was im Endeffekt der Übersichtlichkeit und Wartbarkeit des Codes dient.

In insgesamt fünf Kategorien unterteilt werden in [Völter und Lippert, 2004] weitere Anwendungsfälle aufgezeigt und beschrieben. Eine ausführliche Beschreibung, wie sicherheitsrelevante Aspekte in *Eclipse* RCP Anwendungen verwendet werden können, ist in [Friese et al., 2007] zu finden.

Crosscutting Concerns

Es werden zwei Arten von Crosscutting Concerns unterschieden, die statischen und die dynamischen Crosscuttings [Lahres und Rayman, 2009].

Beim statischen Crosscutting wird die Struktur der Klasse an sich angepasst. Dabei können zum Beispiel neue Felder oder ganze Methoden hinzugefügt werden. Im benannten Beispiel könnte diese das Hinzufügen einer neuen Methode zum Setzen des Markennamens sein.

Im Gegensatz dazu ist das dynamische Crosscutting ein Verfahren, welches das Verhalten von Objekten modifiziert. Dabei werden zum Beispiel vor oder nach einem bestimmten Methodenaufruf weitere Aktionen durchgeführt. Auch das komplette Austauschen einer Methode mit einem anderen Codeblock fällt unter diese Art des Crosscuttings. Dafür kommen sogenannte Interzeptoren (`before`, `after`, `around`) zum Einsatz, die den Programmfluss an vorher definierten Stellen unterbrechen und Codefragmente ausführen.

Wie diese Anpassungen umgesetzt werden, hängt von der eigentlichen Implementierung des aspektorientierten Frameworks ab, welches eingesetzt wird. Die üblichsten Methoden sollen nachfolgend benannt und kurz erklärt werden.

- Umsetzung bei der Kompilierung
  Der Code wird an den entsprechenden Stellen beim Kompilieren so eingesetzt, als wäre er ein Teil des ursprünglichen Codes.

- Umsetzung zur Laufzeit
  Die Klassen werden ohne Aspekte kompiliert und zur Laufzeit so modifiziert, dass sie die Interzeptoren an den vordefinierten Stellen ausführen.

- Proxyklassen
  Es werden dynamisch Proxyklassen erzeugt, die neben den Originalklassen existieren. Diese Proxyklassen können dann auf die Methoden der Originalklasse zugreifen, bevor oder nachdem die definierten Interzeptoren behandelt wurden.

Um explizit festlegen zu können, wo ein Interzeptor in den Programmcode eingreifen soll, werden so genannte *Join Points* benötigt. *Join Points* sind, wie der Name bereits andeutet, Punkte im Code an denen eingegriffen werden kann. Dies kann zum Beispiel das Erzeugen eines Objektes, das Aufrufen einer Methode, das Erzeugen einer Exception oder der Zugriff auf Felder einer Klasse sein. *Join Points* stellen dabei lediglich eine Menge von möglichen Einhängepunkten dar. Unter *AspectJ*, welches eine konkrete Umsetzung der Aspektorientierung für Java darstellt [Kiczales et al., 2001], kann unter anderem auf die folgenden *Join Points* zurückgegriffen werden.

- Feld Zugriffe (field get, field set)
  Diese treten ein, wenn auf ein Feld einer Klasse, also zum Beispiel eine öffentliche Variable, zugegriffen oder diese verändert wird.

- Methodenausführung (execution)
  Ein *Join Point* für die Ausführung einer Methode tritt ein, wenn eine konkrete Implementierung einer Methode aufgerufen wird. Dies ist zum Beispiel nicht mehr der Fall, sobald die Methode in einer Unterklasse überschrieben wird.

- Operationsaufruf (call)
  Der *Join Point* bei Operationsaufrufen wird wie der *Join Point* bei der Ausführung von Methoden verwendet. Allerdings ist dieser nicht abhängig von einer konkreten Implementierung, sondern wird immer beim Aufruf einer Operation angewendet. Dadurch kann sich dieser *Join Point* auch auf abstrakte Methoden beziehen, ein *Join Point* für Methodenausführungen jedoch nicht.

- Konstruktorausführung und -aufruf (contructor execution, contructor call)
  Hier muss zwischen Aufruf und Ausführung eines Konstruktors unterschieden werden. Bei dem Aufruf wird die gesamte Vererbungshierarchie der Klasse betrachtet, also auch alle Konstruktoren der Elternklassen. Die Ausführung hingegen ist lediglich auf den Konstruktor der Klasse selbst bezogen.

Konkret werden bestimmte *Join Points* nun bei der Definition von den sogenannten *Point Cuts* verwendet. Diese stellen nun die expliziten Punkte dar, an denen Interzeptoren aufgerufen werden sollen. Mehrere *Join Points* können dabei an einem *Point Cut* verknüpft werden. Zum Beispiel sind alle Methoden der Klasse `Auto` *Join Points*. Ein *Point Cut* könnte nun alle *Join Points* der Klasse verwenden, die sich auf Methoden beziehen, dessen Namen mit `set*` anfangen. Außerdem kann ein *Join Point* auch in mehreren *Point Cuts* vorkommen. Zusätzlich zu dem soeben definierten *Point Cut* könnte ein weiterer auf alle *Join Points* der Methoden der Klasse `Auto` zugreifen. So wäre ein *Join Point* der Methode `setColor` Teil beider *Point Cuts*.

Was genau an einem definierten *Point Cut* geschehen soll, wird mit dem *Advice* definiert. Ein *Advice* ist eine konkrete Implementierung, die ausgeführt wird,

sobald ein *Join Point* aufgerufen wird, der mittels eines *Point Cuts* mit dem entsprechenden *Advice* verknüpft ist.

Die soeben beschriebenen Punkte sollen anhand der im Listing 4.10 dargestellten Klasse `Auto` veranschaulicht werden. Da hier lediglich auf die Verwendung der aspektorientierten Programmierung eingegangen werden soll, besteht diese Klasse nur aus einer Methode zum Setzen der Farbe.

Listing 4.10: Rudimentäre Auto-Klasse als Basis zur Veranschaulichung der aspektorientierten Programmierung

```
1 public class Auto {
2   private String color;
3   public void setColor(String color) {
4     this.color = color;
5   }
6 }
```

Das folgende Listing 4.11 stellt eine Implementierung in *AspectJ* dar. Hierbei soll vor jeder Ausführung der Methode `setColor` aus der Klasse `Auto` eine Nachricht auf der Konsole ausgegeben werden. Dies kann nützlich sein, um unter anderem Logausgaben zum Überprüfen der Funktionalität in eine Datei zu schreiben.

Listing 4.11: Modifikation der Auto-Klasse durch AspectJ

```
1 public aspect Auto_AspectJ {
2   pointcut color():
3     execution(void Auto.setColor(..));
4
5   before(): color() {
6     System.out.println("setze␣Farbe");
7   }
8 }
```

Die Erläuterung erfolgt nun in der Reihenfolge der Definition der Begriffe. In Zeile 03 wird auf ein *Join Point* zugegriffen. Dieser greift bei jeder Ausführung (execution) einer Methode `setColor` aus einer Klasse `Auto`, die keinen Rückgabewert (`void`) und eine beliebige Liste von Parametern (..) hat. Dieser *Join Point* wird in der Zeile 02 direkt als *Point Cut* mit der Bezeichnung `color` verwendet. Bis zu dieser Stelle wurde lediglich definiert, dass bei der beschriebenen Aktion etwas im Programmcode passieren soll. Was genau passiert wird, im *Advice* in den Zeilen 05 bis 07 definiert. Hier wird eine Zeile in der Konsole ausgegeben, bevor (`before`) der *Point Cut* mit der Bezeichnung `color` ausgeführt wird. Zusammengefasst wird jedes Mal bevor die Methode `setColor` in der Klasse `Auto` ausgeführt wird, die Zeile „setze Farbe“ in der Konsole ausgegeben.

Dabei sei nochmals drauf hingewiesen, dass für diese erweiterte Funktionalität keine einzige Zeile Code der Auto-Klasse angefasst werden muss.

Beispiel

Wie kann die aspektorientierte Programmierung nun dabei helfen, SWT plattformunabhängig zu gestalten? Sobald derartige Teile in den SWT-Bibliotheken identifiziert sind, können die betroffenen Stellen einfach umgangen werden. Dazu könnte vor deren Ausführung ein entsprechender Code dafür sorgen, dass bestimmte Umgebungsparameter verändert werden. Oder die „fehlerhafte“ Methode

könnte auch komplett ausgetauscht werden.

Um dies zu veranschaulichen, soll das Beispiel des Setzens eines Status einer Fortschrittsanzeige unter Linux herangezogen werden. Bereits in den vorherigen Kapiteln wurde in unterschiedlichen Methoden gezeigt, wie die farbliche Gestaltung abhängig eines gesetzten Status geändert werden kann, wie zum Beispiel in Listing 4.3. Daher soll nachfolgend nur auf den relevanten Teil Bezug genommen werden, der sich mit der Aspektorientierung beschäftigt. Die Grundidee ist, nach jedem Aufruf der Methode `setState()`, abhängig vom verwendeten Betriebssystem, die Vordergrundfarbe zu setzen, wenn das Betriebssystem die Methode nicht oder nicht korrekt unterstützt. Der entsprechende Code ist in Listing 4.12 dargestellt.

Listing 4.12: Beispielumsetzung von Progressbar.setState in AspectJ

```
1  pointcut ProgressBarSetState(int state):
2    call(void ProgressBar.setState(int)) && args(state);
3
4  after(int state): ProgressBarSetState(state) {
5    if (isLinux) {
6      System.out.println("is linux");
7      setColorDependantOnState(thisJoinPoint.getTarget(), state);
8    }
9
10   if (isWindows) {
11     // do nothing
12   }
13 }
```

In Zeile 01 wird ein neuer *Point Cut* mit dem Namen `ProgressBarSetState` definiert. Dieser wird jedes Mal aktiv, wenn in der Klasse `ProgressBar` die Methode `setState` mit einem Parameter des Typs Integer (`int`) aufgerufen wird (Zeile 02). Es handelt sich hierbei nicht um einen Methodenaufruf in einer selbst geschriebenen Klasse, sondern um einen Operationsaufruf einer fremden Klasse, daher wird der call-Operator verwendet. Da später der übergebene Parameter noch benötigt wird, wird dieser mit dem Konstrukt `args(state)` gespeichert, um ihn später weiterverwenden zu können. Ab Zeile 04 wird das eigentliche *Advice* definiert, also was passieren soll, wenn ein bestimmter *Point Cut* ausgeführt wird. Der Interzeptor `after` sorgt hierbei dafür, dass der definierte *Point Cut* `ProgressBarSetState` nach der Ausführung der Methode `setState` aufgerufen wird. In den Zeilen 05 bis 12 findet die eigentliche Behandlung statt.

Der aspektorientierte Code für das oben beschriebene Beispiel befindet sich separiert vom Code der `ProgressBar` in einer eigenen Source-Datei. Dieser kann sich auch von Projekt zu Projekt unterscheiden, um zum Beispiel unterschiedliche Färbungen für Unterschiedliche Kunden oder Produkte zu realisieren.

Vor- und Nachteile des aspektorientierten Ansatzes

Mit Hilfe der aspektorientierten Programmierung ist es, wie an dem Beispiel zu sehen, sehr einfach, in bereits existierende Quelltexte einzugreifen und diese den eigenen Bedürfnissen anzupassen und zu erweitern. Dabei ist es nicht erforderlich, den Quelltext direkt zu verändern. Dies ermöglicht demnach auch die Modifikation von Bibliotheken, auf dessen Inhalt selbst kein Zugriff gewährt wird.

Allerdings wird die Einsetzbarkeit durch die Zielklassen beschränkt, in die mittels der *Point Cuts* eingegriffen werden soll. Es ist zwar gängige Praxis, dass bei der objektorientierten Programmierung in Java große Module in kleinere Arbeitspakete, und damit auch Methoden, unterteilt werden, es aber nicht zwingend nötig ist. Das bedeutet, in je mehr Methoden ein ursprüngliches Problem zerlegt wurde, desto feingranularer kann mit der aspektorientierten Programmierung gearbeitet werden. Betrachtet man allerdings den umgekehrten Fall, nämlich wenn eine Klasse im worst case Szenario lediglich aus einer großen Methode besteht, kann der aspektorientierte Ansatz kaum zu einer Lösung beitragen. In derartigen Fällen sollte allerdings generell eine Neuimplementierung betroffener Codeabschnitte in Betracht gezogen werden, um im Zuge dessen auch die Les- und Wartbarkeit entscheidend zu verbessern.

Kapitel 5

5 Konzeption einer neuen SWT Komponente

Wie beim Vergleich der einzelnen Lösungsansätze im vorherigen Kapitel zu sehen ist, ist keine der vorgestellten Methoden im vollen Maße geeignet, SWT Differenzen in den diversen Betriebssystem-Varianten zu beheben. Aus diesem Grund können die Lösungsansätze nur für kleinere Projekte oder bedingt auch für größere Projekte verwendet werden. Dies geht aber immer mit einem erhöhten Aufwand bei der Entwicklung oder zumindest einer erhöhten Fehleranfälligkeit einher.

In den nächsten Kapiteln soll daher eine weitere Methode vorgestellt werden, die die meisten aufgezeigten Beschränkungen umgehen soll. Für diesen Zweck wird eine neue Komponente auf Basis von SWT kreiert, die, zur Veranschaulichung des Prinzips, das vorhandene Tabellenkonzept ersetzen kann. Eine Tabelle ist eines der komplexesten GUI-Elemente, wenn es um Oberflächenentwicklung geht, so dass anhand dieser auf viele verschiedene Probleme bei der Konzeption und Programmierung einer Komponente eingegangen werden kann. Diese Komponente soll, ähnlich wie beim Konkurrenzframework Swing, auf einem Basiscontainer aufbauen. Da dabei nicht auf vorhandene native Implementierungen zurückgegriffen wird, werden damit von vornherein unterschiedliche Umsetzungen auf den diversen Plattformen, für die SWT angeboten wird, umgangen. Da es sich hierbei allerdings um eine nahezu vollständige Neuentwicklung handelt, müssen selbst grundlegende Basisfunktionen und -eigenschaften mit bedacht und in die Planung einbezogen werden.

Die Umsetzung soll in der ersten Fassung mindestens die nachfolgend aufgeführten Punkte erfüllen. Bei der Auflistung wird teilweise Bezug auf die vorhandene Umsetzung von SWT genommen, ob diese Vorgabe erfüllbar wäre oder nicht und ein Vergleich mit Swing angestrebt.

- Nutzung von MVC Pattern
  Das MVC Pattern gilt mittlerweile als De-facto-Standard für den Grobentwurf vieler komplexer Komponenten und ganzer Softwaresysteme. Bei diesem Muster ist eine Trennung von Daten (Model), Anzeige (View) und Geschäftslogik (Controller) vorgesehen. Besonders die Auslagerung der Anzeige ist für die Umsetzung von unterschiedlich gestalteten Oberflächen,

also auch Corporate Designs, sehr hilfreich.

Da SWT auf unterschiedlichen Plattformen portiert wurde und im Grunde genommen lediglich eine Schnittstelle zu den nativen Widgets darstellt konnte weitestgehend nur auf einen kleinsten gemeinsamen Nenner zurückgegriffen werden. Ein MVC Pattern wird daher nicht von Haus aus verwendet. SWT wurde später um das UI-Toolkit JFace ergänzt, welches unter anderem auch Viewers zur Verbindung von GUI-Elementen zu einem Datenmodell ermöglichen sollen. Das Datenmodell besteht allerdings in den meisten Fällen aus einem einfachen Array oder einer Liste, wobei sich der Entwickler selbst um die Aktualisierungen im View kümmern muss.

- Einheitliche Basisfunktionen
  Zu den Basisfunktionen zählen unter anderem das Setzen von Spaltenüberschriften, das Verändern von Zellinhalten programmatisch und per Eingabe des Benutzers, das Modifizieren des ausgewählten Views (sprich das Scrollen mit der Maus oder der Tastatur) oder das Hinzufügen und Löschen von Daten, seien es ganze Zeilen oder Spalten. Auch das automatische oder manuelle Aktualisieren von Bereichen im View, sobald sich Daten im Model verändern, muss bedacht werden. Weiterhin muss in der Planung betrachtet werden, wie Selektionen von Zellen, Zeilen und Spalten vorgenommen werden können.

- Einheitliches Bedienkonzept
  Die Bedienung soll auf den Hauptplattformen, die von SWT unterstützt werden, einheitlich sein. Von diesen sollen zunächst Windows, Linux und Mac näher betrachtet werden. Hier ist vor allem auf ein konsistentes Event-Handling zu achten.
  Unter SWT kommt es an einigen Stellen vor, dass Events in unterschiedlicher Reihenfolge ausgelöst werden. Auch inkonsistente Eventnutzung kommt vor (Selection für Checked bei Checkboxen etc.)

- Erweiterter Funktionsumfang gegenüber den nativen Tabellen
  Um das erweiterte Spektrum und die weitreichenden Möglichkeiten dieser Methode zu veranschaulichen, soll die endgültige Fassung der Tabelle um eine Funktionalität erweitert werden, die in den nativen Implementierungen bisher nicht oder nur in seltenen Fällen vorkommt. In diesen Umsetzungen ist es nicht vorgesehen, Spalten zu fixieren. Fixierte Spalten sind Spalten, die am Anfang einer Tabelle angezeigt werden. Diese sind unabhängig von der horizontalen Scrollbar. Sobald nachfolgende nicht-fixierte Spalten gescrollt werden, verschwinden diese hinter den fixierten Spalten beziehungsweise kommen hinter diesen wieder vor.
  Diesem Faktor kann mit dem bestehenden SWT (und auch Swing) ohne Neuimplementierung nur sehr schwer nachgegangen werden. Es ist möglich, eigene Widgets durch geschickte Kombination aus mehreren vorhandenen Widgets zusammenzusetzen. Existierende Grenzen zeigen sich allerdings schnell, sobald die interne Funktionsweise einer Komponente geändert

werden muss.

- Performanz
  Eine Tabelle sollte hunderte, tausende oder mehr Zeilen (Datensätze) und Spalten verwalten können, ohne dass der Benutzer unangenehm lange auf Aktualisierungen des Views warten muss. Daher soll bereits in einer frühen Phase der Planung eine hohe Performanz angestrebt werden.

- Ressourcenschonend
  Der Ressourcenverbrauch sollte sich, angelehnt an den Punkt Performanz, ebenfalls moderat verhalten. Besonders bei Tabellen ist nicht abzuschätzen, wie viele Daten später durch den Benutzer verwaltet und dargestellt werden. Im Falle von großen Datenmengen sollte sich der Speicherverbrauch, der zusätzlich durch die Tabelle erzeugt wird, daher in Grenzen halten.

- Corporate Design
  Bei einem Corporate Design sollen sich grafische Elemente wie Farben, Formen oder Bilder einer Organisation im späteren Produkt wiederfinden. Dazu ist es nötig, auch native GUI-Elemente wie Schaltflächen oder Tabellen grafisch anpassen zu können. Um dies zu bewerkstelligen, muss die Möglichkeit geschaffen werden, die Elemente nach eigenen Wünschen und Vorgaben zu modifizieren.
  SWT bietet hier im Gegenzug zu Swing keinerlei Möglichkeiten für grafische Anpassungen. Es wäre zwar anzudenken, mittels des SWT paint-Events Bereiche selbst zu gestalten, allerdings steigt der Aufwand bei komplexen Komponenten erheblich. In Swing wird die Oberfläche von vornherein selbst gezeichnet. Dazu wird auf die UI-Klassen des aktuell gesetzten Look and Feels zurückgegriffen. Daher müssten hier nur die entsprechenden UI-Klassen (soweit möglich) konfiguriert oder modifiziert werden.

Nachfolgend soll nun Schritt für Schritt ein Lösungsansatz erarbeitet und die jeweiligen Vor- und Nachteile herausgestellt werden. Dieser Ansatz dient vorwiegend zur Veranschaulichung des Aufwands und des Potenzials, den die Kombination von SWT mit Swing bietet.

5.1 Initialer Lösungsansatz

Der initale Ansatz besteht darin, die Grundidee von Swing auf SWT zu übertragen und so die Basis für eine besser anpassbare und plattformunabhängige Komponente zu schaffen. Swing nutzt die Klasse `Container` des Toolkits AWT, um auf plattformspezifische Eigenschaften zuzugreifen und simple Grafikoperationen auszuführen. Sämtliche höheren Widgets basieren auf diesem Konstrukt beziehungsweise der Swing-Klasse `JComponent`, die die AWT-Klasse `Container` weiter abstrahiert.

Unter SWT werden durch Peer Klassen nahezu alle Widgets auf die nativen Betriebssystemelemente abgebildet. Als Basis dient hier die abstrakte Klasse `Widget`.

Jede Komponente, die in grafischen Oberflächen verwendet wird, erbt von dieser Klasse. Allerdings können hier noch keine Kindelemente hinzugefügt und mittels Layoutmanager positioniert werden. Dies geschieht erst auf Ebene der Klasse `Composite`. Wie der Name bereits vermuten lässt, kann ein Composite beliebig viele weitere Widgets beinhalten. Eine grundlegende Unterstützung von Scrollbars wird ebenfalls bereits angeboten. Aus den genannten Features kann, aufbauend auf einem Composite, neue und vor allem komplexere Komponenten entwickelt werden.

Als Basis für die nachfolgende Konzeption soll das MVC Pattern verwendet werden, welches bereits in Unterabschnitt 2.3.2 beschrieben wurde. Dadurch soll eine strikte Aufgabentrennung erfolgen, um unter anderem in späteren Produkten den Austausch einzelner Bereiche zu ermöglichen und die einzelnen Teile der Komponente getrennt voneinander entwickeln zu können. Designer oder Softwareergonomen kümmern sich dabei um alle Aspekte des Komponenten-Designs (siehe zum Beispiel [Wood, 1997]) und Softwarearchitekten beziehungsweise Entwickler um die eigentliche Implementierung.

In den nachfolgenden Abschnitten werden die einzelnen Teilbereiche des Patterns in Bezug auf die geplante Umsetzung näher betrachtet. Zur Veranschaulichung soll, wie bereits in Unterabschnitt 2.3.2 angesprochen, eine einfache Benutzerübersicht dienen, auf die an passenden Stellen Bezug genommen wird. Diese wird auf einem Datenpool von diversen Benutzern aufbauen. Wo die Daten persistent gespeichert werden, ist beim MVC irrelevant. So kann als Datenquelle zum Beispiel eine XML Datei oder eine rationale Datenbank verwendet werden. Jeder Benutzer aus diesem Pool besitzt die Eigenschaften Name, Passwort und Postleitzahl, die mit entsprechenden Methoden ausgelesen und gesetzt werden können. Eine abstrakte Klasse Person würde wie in Listing 5.1 aussehen.

Listing 5.1: Abstrakte Klasse „Benutzer“

```
1  public abstract class Benutzer {
2    public String getName();
3    public void setName(String s);
4
5    public String getPassword();
6    public void setPassword(String s);
7
8    public int getZipCode();
9    public void setZipCode(int i);
10 }
```

Auf Vorschläge für Umsetzungen soll an dieser Stelle nur im Rahmen der getätigten Beispiele und Erläuterungen eingegangen werden.

5.2 Model

Das Modell bildet die eigentliche Datenbasis, die in einer elementaren Fassung sowohl Operationen zum Lesen (Datenausgabe) als auch zur Modifikation der vorhandenen Daten (Dateneingabe) unterstützen muss. Dieses Model muss im MVC Verbund für eine Tabelle über ein Interface einheitlich angesprochen werden können. Demnach werden lediglich angepasste Model-Klassen benötigt, die

auf bereits vorhandene Datenpools zugreifen können. Die Verwendung und Umstellung der neuen Tabelle kann somit in bereits existierende Projekte meist recht einfach erfolgen.

Eine Tabelle gliedert die Daten in Spalten und Zeilen. Der Zugriff auf ein spezielles Datum erfolgt somit über die Angabe der Spalte, was ein Schlüssel (zum Beispiel „Spalte A“) oder Index (Spalte 1) sein kann, und der Zeile, was in der Regel die Zeilennummer ist. Zu Gunsten der Übersichtlichkeit soll in der konkreten Implementierung lediglich auf indexbasierte Identifikatoren zurückgegriffen werden. Eine Erweiterung auf ein schlüsselbasiertes Schema ist allerdings ohne Probleme durch einfache Anpassungen des Modells möglich. Für die korrekte Darstellung der Daten besitzt das Datenmodell eine Methode, um die Anzahl der Daten zu erhalten und eine Methode, um ein bestimmtes Datum zu liefern. Zudem muss sich die Hauptkomponente der Tabelle bei dem Model als Listener registrieren können. Damit hat das Model die Möglichkeit, die Komponente bei Veränderung des Datenbestandes zu benachrichtigen. Der Controller kann daraufhin entscheiden, ob bestimmte Bereiche der Anzeige neu gezeichnet werden müssen oder ob darauf verzichtet werden kann. Die Vorgehensweise wird im Abschnitt 5.4 näher betrachtet.

Bereits an dieser Stelle sei darauf hingewiesen, dass derartige Änderungsnachrichten nur generiert werden sollten, wenn sich wirklich Daten „verändert“ haben. Im produktiven Einsatz kann es unter Umständen vorkommen, dass das Modell mehrfach in der Sekunde mit den gleichen Daten befüllt wird. Zum Beispiel, wenn Sensordaten automatisiert zeitlich gesteuert eingelesen werden, die sich aber nicht immer ändern. In diesem Fall würde jedes Mal ein Neuzeichnen der Tabelle veranlasst werden, was zu Performanceproblemen führen kann.

Da das Model lediglich über Schnittstellen mit dem Controller kommuniziert und die View-Komponente ebenfalls über Schnittstellen über Änderungen im Model informiert wird, ist das Model weitestgehend unabhängig von den restlichen Komponenten. Die Daten des Models werden in java-eigenen Konstrukten gehalten, verarbeitet und ausgelesen, wodurch weiterhin keine Abhängigkeiten zum SWT entstehen.

In Bezug auf das eingangs erwähnte Beispiel der Benutzerübersicht soll das Modell aus einer Liste von Benutzer-Objekten bestehen. Dem Modell kann darüber ein einfacher Zugang auf Informationen, wie die Anzahl der Benutzer (Länge der Liste) und die Benutzer-Objekte selbst, ermöglicht werden. Der Zugriff auf einzelne Zeilen der Tabelle entspricht dabei dem Zugriff auf die jeweiligen Speicherplätze der Liste. Die Zuordnung der Spalten erfolgt über die Felder des Objekts. Somit entspricht die erste Spalte dem Feld Name, die zweite Spalte dem Feld Passwort und die dritte Spalte dem Feld Postleitzahl.

5.3 View

Der Viewer ist für die eigentliche Anzeige der darzustellenden Daten zuständig. Im konkreten Fall der Tabelle also, wie die Informationen aus dem Modell in Spalten und Zeilen im Sichtbereich des Nutzers übertragen werden. Da, wie eingangs

beschrieben, der Umfang der einzusetzenden SWT Elemente auf elementare Widgets, wie das `Composite`, reduziert werden soll, müssen alle Teile selbst verwaltet und gezeichnet werden.

Ein erster trivialer Ansatz führt daher dazu, den gesamten Inhalt der Tabelle auf ein Composite zu zeichnen und dieses als Kindelement einem `ScrolledComposite` hinzuzufügen. Eine „simple“ Tabelle könnte so bereits erstellt werden. Diese Vorgehensweise hat allerdings mehrere Nachteile:

- Der Speicherverbrauch wächst mit der Anzahl der Spalten und Zeilen im Modell. Die Daten werden bei diesem Verfahren sogar doppelt vorgehalten, nämlich einmal gezeichnet im View und in dem Modell selbst, aus dem die Daten ursprünglich stammen. Bei großen Tabellen kann dies zu Speicher- und damit Performanceproblemen führen.

- Durch das Modell ist eine Generierung von virtuellen Daten möglich, die in der Tabelle ebenfalls zu Speicher- und Performanceproblemen führen können. Ein Modell kann zum Beispiel so aufgebaut sein, dass für jedes abgefragte Datum eine Kombination aus Spalten- und Zeilenindex zurückgeliefert wird. Ein derartiges Modell ist nur durch die vorher festgelegte Anzahl der Spalten und Zeilen begrenzt und benötigt selbst nur sehr wenig Speicherplatz. Das erzeugte Tabellen-Widget wäre aber um ein Vielfaches größer.

- Die Spaltenköpfe verschwinden beim horizontalen Scrollen. Da die gezeichnete Tabelle lediglich ein Kindelement eines `ScrolledComposite` ist, welches sich um die Scroll-Bewegungen kümmert, werden alle Informationen, darunter auch die Spaltenköpfe, durch das Elternelement verdeckt. Eine Lösung dafür wäre die Trennung von Spaltenköpfen und Inhalt. Dies führt allerdings dazu, dass zwei Komponenten beim horizontalen Scrollen synchronisiert werden müssen, was besonders beim schnellen Scrollen zu asynchronen „Hinterherzieh“-Effekten führen kann.

Besonders aus dem genannten möglichen sehr hohen Ressourcenverbrauch ist diese Herangehensweise für produktive Einsätze eines Widgets nicht geeignet. Dieser kann allerdings signifikant verringert werden, wenn lediglich der aktuell sichtbare Bereich überwacht und dargestellt wird. Dazu muss der Viewer genaue Angaben wie Zeilenanzahl, Spaltenanzahl, Zellhöhe oder Zellbreite aus den gelieferten Modelldaten extrahieren beziehungsweise berechnen. Daraus ist es möglich, mittels geeignetem Clipping und Neuzeichnen nur den aktuellen View darzustellen. Ein entsprechende schematische Darstellung ist in Abbildung 5.1 zu sehen.

Die Abbildung zeigt eine Tabelle mit insgesamt sechs Spalten sowie acht Zeilen. Die Zelleninhalte sind nach dem Schema `Spaltenindex+"x"+Zeilenindex` generiert. Der aktuelle Sichtbereich ist rot umrandet. Sobald Änderungen im Model verzeichnet werden, die die Zellen von `1x2` bis `5x7` betreffen, muss lediglich dieser relevante Teil neu generiert werden. Die Spalte mit dem Index 6 sowie die Zeilen 1 und 8 können aus der Betrachtung vollständig herausgenommen werden, da diese den sichtbaren

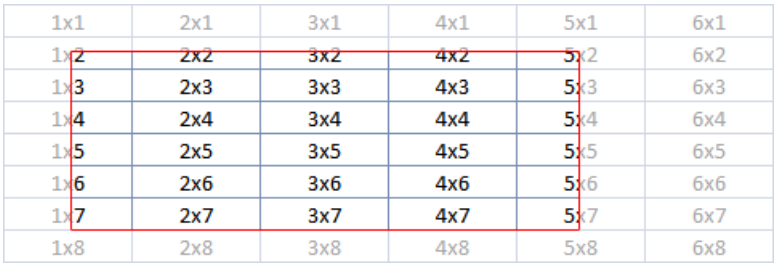

Abbildung 5.1: View- und Draw-Schema einer Tabelle

Bereich nicht berühren. Im Speicher muss hierbei nur der rot umrandete Bereich gehalten werden, was den Ressourcenverbrauch, auch bei sehr großen Modellen, auf ein Mindestmaß reduziert. Damit steigt allerdings die Anzahl der nötigen Überprüfungsschritte an, da bei jeder Scrollbewegung und Modell-Aktualisierung zunächst der betroffene Abschnitt extrahiert werden muss.

Für die technische Umsetzung muss an dieser Stelle auf das SWT zurückgegriffen werden. Bei diesem werden über das Ereignis vom Typ `SWT.Paint` Neuzeichnungen der Komponente angefordert, die auf dem sogenannten „Grafik Kontext“ (`gc`) umgesetzt werden. Diese Generierung findet für jede Komponente getrennt statt, bei denen sich Inhalte, zum Beispiel durch Interaktionen des Benutzers oder Entfernen eines überlagernden Fensters, geändert haben könnten. Das Ereignis lässt sich separat für jede Komponente mittels Listener abfangen. Das Listing 5.2 zeigt die grundlegende Behandlung eines Paint-Events mit SWT.

Listing 5.2: Paint-Event unter SWT

```
1 Composite container = new Composite(parent, SWT.NONE);
2 container.addListener(SWT.Paint, new Listener() {
3   public void handleEvent(Event e) {
4     if (e.gc == null)
5       return;
6     ...
7   }
8 });
```

Dem `Composite` wird dabei ein neuer Event-Handler vom Typ `SWT.paint` in Zeile 02 hinzugefügt. Dieser wird immer dann aufgerufen, wenn Zeichenoperationen abgearbeitet werden müssen, die das `Composite` selbst betreffen. An dieser Stelle können alle relevanten Zeichenoperationen, die für die Darstellung der Tabelle nötig sind, ausgeführt werden. Die Zeilen 04 und 05 beinhalten eine Sicherheitsabfrage für den Fall, dass der grafische Kontext, auf dem gezeichnet werden soll, nicht mehr vorhanden ist. Das `Event` Objekt beinhaltet diverse Informationen, von denen nachfolgend die hierfür relevanten aufgelistet sind:

- `gc`
  Der Kontext, auf dem gezeichnet werden soll. Auf diesem werden die vom System unterstützen Grafikoperationen, wie zum Beispiel geometrische Objekte zeichnen, Farben setzen oder Clipping, ausgeführt.

- `width` und `height`
  Die Breite und Höhe des Bereichs, welcher in der Komponente neu gezeich-

net werden soll.

- `x` und `y`
  Der obere linke Punkt des Bereichs, welcher neu gezeichnet werden soll.

Die Angaben `x`, `y`, `width` und `height` beschreiben dabei das zu zeichnende Rechteck. Dieses beschreibt eine Teilmenge des sichtbaren Bereichs der Komponente. Wird zum Beispiel ein Bereich der Komponente von einem Fenster überdeckt und dieses Fenster wieder entfernt, so muss auch nur dieser Bereich neu generiert werden.

5.4 Controller

Der Controller ist für die Interaktion des Benutzers mit der Komponente verantwortlich. Dieser Teil bildet also das spezifische Verhalten der Komponente unter verschiedenen Aktionen ab. Wird zum Beispiel in der angezeigten Tabelle ein Wert programmatisch oder durch den Benutzer verändert, so teilt der Controller dies dem Model mit. Das Model speichert den entsprechenden Wert ab und teilt wiederum dem View die Änderung mit, welches dann eine Aktualisierung der Anzeige veranlasst. Der Controller stellt demnach die Logik der Komponente bereit und kümmert sich um die Verwaltung der eingehenden Ereignisse (Events).

Ebenso wie die View-Komponente ist der Controller direkt von SWT abhängig. Eingehende Ereignisse müssen vom Controller registriert und an entsprechenden Stellen umgesetzt werden. Die Herangehensweise ist ähnlich wie bei der View-Komponente. Ein Beispiel für ein Doppelklick-Event ist in Listing 5.3 zu sehen.

Listing 5.3: Doppelklick-Event unter SWT

```
1 Composite container = new Composite(parent, SWT.NONE);
2 container.addListener(SWT.MouseDoubleClick, new Listener() {
3   public void handleEvent(Event e) {
4     ...
5   }
6 });
```

Über die Methode `addListener()` des `Composite`, also der eigentlichen Komponente, wird ein Listener für ein bestimmtes Ereignis registriert. An dieser Stelle kann nun die spezifische Abarbeitung erfolgen. Bezogen auf das Listing könnte die Komponente einen Editor über der selektierten Zelle für Benutzereingaben öffnen, sofern mit dem Doppelklick eine Zelle selektiert wurde und diese auch veränderbar ist.

So kann ein Texteingabefeld angezeigt werden, wenn im Beispiel der Benutzerübersicht auf ein Element in der Spalte „Passwort“ geklickt wird. Dieses Eingabefeld würde, im Gegensatz zum Eingabefeld für den Namen, die Daten hinter „*“ verschleiern.

Zu beachten sei an dieser Stelle allerdings, dass die Ereignisse vom Betriebssystem erzeugt, an SWT weitergeleitet und dort prozessiert werden, bevor sie durch die Listener dem Entwickler zur Verfügung stehen. So kann ein einzelner Tastendruck mehrere Low-Level Events unter Windows erzeugen, die erst in SWT zu einem KeyDown oder KeyUp-Event zusammengesetzt werden [Northover und

Wilson, 2004]. Bei einer plattformunabhängigen Entwicklung muss ebenso auf plattformspezifische Ereignisse geachtet werden. So wird unter MacOS von einem Benutzer erwartet, einen selektierten Text mittels *Command+C* kopieren zu können. Unter Windows wäre das Äquivalent hingegen *Control+C*. Um derartige betriebssystemspezifische Ereignisbehandlungen handhaben zu können, müssen die entsprechenden Kontrollmechanismen von der Komponente getrennt und in separate Klassen ausgelagert werden, ähnlich der UI-Delegates unter Swing. Diese ermöglichen es dadurch, für jedes Betriebssystem andere Steuerungsabläufe zu implementieren und so eine Komponente für Windows, Linux oder MacOS auf einer einheitlichen Basis zu schaffen.

5.5 Erweiterte Basis-Funktionalität

Mit den beschriebenen Schritten kann bisher eine zwar performante aber dennoch funktional sehr eingeschränkte Variante einer Tabelle erschaffen werden. Zunächst fehlen noch grundlegende Eigenschaften, von denen eine Auswahl in der nachfolgende Liste aufgeführt sind.

- Größenveränderung von Spalten
- Automatische Sortierung von Zeilen
- Filtern von Zeilen und Spalten
- Inhaltsabhängige Zell-Editoren
- Anpassung unterschiedlicher Attribute (Schrift, Größe, Farbe) der Tabelle

Der vorgestellte Ansatz, Swing auf Basis von SWT nachzubauen, bietet allerdings auch die Möglichkeit den Funktionsumfang einer Komponente signifikant zu steigern. Zur Veranschaulichung sollen die beiden nachfolgend aufgelisteten Zusatzfunktionen dienen.

- Fixierte Spalten
  Fixierte Spalten sind aus Sicht des Nutzers Spalten, die je nach Konfiguration am linken oder rechten Rand der Tabelle fixiert sind. Alle anderen Spalten werden bei Scrollbewegungen durch diese verdeckt. Aus Sicht des Entwicklers sind diese Spaltentypen von horizontalen Scrollbewegungen ausgeschlossen und befinden sich nicht im scrollbaren Viewport.

- Gruppierung von Zeilen und Spalten
  Die Gruppierung von Zeilen oder Spalten gehört in Office-Anwendungen wie Microsoft Excel seit längerem zur Standardumsetzung. Native Tabellen des Betriebssystems unterstützen diese Funktionalität in der Regel nicht.

5.6 Auswertung des Verfahrens

Das bisher vorgestellte Verfahren stellt einen groben Leitfaden zur Umsetzung von Komponenten mit einer Swing-ähnlichen Architektur auf Basis von SWT dar. Dabei wird das MVC Pattern bereits in den frühen Konzeptionsphasen eingebunden und als Grundlage für höhergradige Funktionen genutzt. Dadurch ist ein flexibler Programmentwurf möglich, der spätere Änderungen oder Erweiterungen erleichtert. Bei geschickter Umsetzung ist es sogar denkbar, ein bestehendes Swing-Model für die SWT-Umsetzung zu verwenden.

Ein einheitliches Bedienkonzept und ein einheitlicher Funktionsumfang über alle unterstützten Plattformen ist durch die Art der Umsetzung zwangsweise gegeben. Da nur auf einen kleinen Teil der SWT-Bibliotheken zurückgegriffen wird, sind unterschiedliche Verhaltensweisen, zum Beispiel unter Microsoft Windows, Apple MacOS und Linux, weitestgehend ausgeschlossen. Allerdings ist zu beachten, dass die Ereignisse der Eingabegeräte, also in der Regel Tastatur und Maus, zunächst durch SWT verarbeitet werden. Sollten an dieser Stelle plattformspezifische Einschränkungen gelten, müssen diese in der Komponente selbst behandelt werden.

Der Funktionsumfang einer einzelnen Komponente kann durch geeignete Methoden variabel gestaltet werden. Prinzipiell ist dieser allerdings nicht begrenzt und nur von vorhandenen Ressourcen wie Speicherplatz, Entwicklerstärke, Zeit oder Budget abhängig. Der mögliche Umfang wird nur durch die plattformspezifischen Gegebenheiten beschränkt, die die Basisklasse der Widgets, das SWT Composite, selbst hat.

Der Ressourcenverbrauch und die Performanz eines Widgets liegen in der Regel in der Hand des Entwicklers. Diese Aspekte können allerdings in sehr frühen Phasen der Konzeption in die Planung mit aufgenommen werden. So beschränkt sich zum Beispiel der Speicherverbrauch auch bei großen Datenbasen durch die beschriebene Betrachtungsbegrenzung des aktuellen Viewports in Abschnitt 5.3. Da grafische Elemente in separate Klassen des Views ausgelagert wurden, ist die Umsetzung eines speziellen Designs des betroffenen Widgets ohne größere Probleme möglich. Auch hier ist der Umfang der Anpassungen lediglich durch externe Ressourcen begrenzt.

Kapitel 6

6 Evaluierung der vorgestellten Methoden

Nachfolgend sollen die im Kapitel 4 vorgestellten Ansätze mit dem in Kapitel 5 vorgestellten Verfahren zur Behebung der vorhandenen Differenzen in den SWT-Bibliotheken verglichen und der spezifische Anwendungsumfang herausgestellt werden. Die einzelnen Methoden sind in den Kapiteln 4.1 (Precompiler in Java), 4.2 (Interfaces), 4.3 (Präprozessoren), 4.4 (Buildscripte) und 4.5 (Der aspektorientierte Ansatz) beschrieben.

6.1 Robustheit

Nach Andreas Wieland und Carl Marcus Wallenburg wird der Begriff „Robustheit“ wie folgt definiert:

> *Der Begriff Robustheit bezeichnet die Fähigkeit eines Systems, Veränderungen ohne Anpassung seiner anfänglich stabilen Struktur standzuhalten.*
>
> *[Wieland und Wallenburg, 2012]*

Nachfolgend sollen die vorgestellten Verfahren gegenüber (automatischen) Veränderungen im Quelltext untersucht werden. Zu diesen Veränderungen zählen unter anderem Codeformatierungen nach vordefinierten Regeln durch Entwicklungsumgebungen oder manuelle Codeanpassungen.

Precompiler sowie Interfaces sind von diesen Änderungen nicht betroffen, da sie ausschließlich Konstrukte verwenden, die durch die Java-Spezifikation definiert wurden.

Präprozessoren nutzen hingegen spezielle Sequenzen, die in den Kommentaren der Quelltexte hinterlegt sind. Durch Codeformatierungen können diese speziellen Kommentarsektionen allerdings so verändert werden, dass der entsprechende Präprozessor an diesen Stellen nicht mehr greift.

Das vorgestellte Verfahren, welches auf Buildscripte aufbaut, verwendet normale Zeichenkettenersetzungen. Diese funktionieren in der Regel zwar noch bei

automatischen Codeformatierungen, versagen allerdings, wenn verwendete Ersetzungssequenzen auch in anderen Teilbereichen des Quelltextes vorkommen.

Ähnlich verhält es sich bei der aspektorientierten Programmierung. Die definierten *Point Cuts* können nach Änderungen im Quelltext auf unterschiedliche Weise ihre Funktion verlieren.

- Der Punkt im Quelltext, auf den sich der *Point Cut* bezieht, wurde gelöscht oder umbenannt. Dadurch wird die entsprechende Funktionalität nicht mehr umgesetzt.
- Eine Methode wurde hinzugefügt, auf die die Definition des *Point Cuts* ebenfalls zutrifft. Dadurch wird die entsprechende Funktionalität zu oft ausgeführt.

Eine Neukonzeption nutzt wie die Precompiler und Interfaces lediglich spracheigene Konstrukte. Zudem werden Standard-SWT-Widgets komplett ersetzt, so dass diese auch von weiteren Quelltextänderungen unabhängig sind.

6.2 Entwicklungs- und Bedienkomfort

In diesem Abschnitt soll vor allem der Aufwand betrachtet werden, der bei der Anwendung der einzelnen Verfahren nötig ist.

Vor allem Precompiler, Präprozessoren und Buildscripte bedürfen einem erhöhten manuellen Aufwand. Diese Verfahren müssen vom Entwickler selbst im Quelltext an Stellen angewendet werden, an denen eine Differenz in den SWT-Bibliotheken existiert und behoben wurde. Demnach muss der Entwickler ebenfalls davon Kenntnis haben. Die Variante der Interfaces ist davon weniger betroffen, da diese auch ohne gesonderte Kenntnisse an Stelle des normalen Funktionsaufrufs angewendet werden kann.

Die aspektorientierte Programmierung bietet den größten Komfort, da die vorhandenen Codeabschnitte vollautomatisch bei der Kompilierung in den Bytecode von Java eingewoben werden. Der Entwickler kann nach wie vor alle Konstrukte und Widgets von SWT ohne gesonderte Anpassungen verwenden.

Die Neukonzeption ersetzt vorhandene Widgets vollständig. Dadurch muss der Entwickler diese nur anstelle der normalen SWT Widgets verwenden. Auch hier ist der Aufwand relativ gering, da nicht auf SWT Eigenheiten geachtet werden muss.

6.3 Aufwand und Kosten

Für die in Kapitel 4 beschriebenen Methoden müssen zunächst alle für ein Produkt relevanten SWT Missstände identifiziert werden. Anschließend ist die Behebung mittels eines Workarounds und die entsprechende Umsetzung notwendig. Insbesondere bei den Precompilern, Präprozessoren und Buildscripten muss die Umsetzung immer manuell an den betroffenen Stellen im Quelltext durchgeführt

werden. Diese Verfahren bieten daher keine Sicherheit, dass ein Bereich einfach übersehen wird. Allerdings muss ein SWT-Fehler nur einmalig behoben und an entsprechenden Abschnitten kopiert werden.

Darauf kann allerdings bei der Einbindung per Interface oder mittels des aspektorientierten Ansatzes verzichtet werden. Insbesondere durch Aspektorientierung kann die Entwicklung erleichtert werden, da die relevanten Teile automatisch beim Kompilieren ersetzt werden und sich der Entwickler dadurch darüber keine Gedanken mehr machen muss. Allerdings ist hierfür die Verwendung eines entsprechenden Compilers notwendig, was bei bestehenden Projektarchitekturen mitunter zu Problemen führen könnte, wenn diese zum Beispiel einen bestimmten Compiler benötigen.

Die Erzeugung einer neuen Komponente erzeugt hingegen den größten Aufwand. Wie in Kapitel 5 zu sehen ist, müssen selbst grundlegende Dinge, wie Basisfunktionen, Ereignisbehandlungen oder die Anwendung von Zeichenoperationen von grundauf geplant und umgesetzt werden. Insbesondere bei komplexeren Widgets wie Tabellen oder Bäumen kann daher der eigentliche Nutzen den Aufwand für Konzeption und Implementierung schnell übersteigen. Die Vielfalt der Anforderungen an derartig umgesetzte Komponenten steigt weiter, wenn die Leitlinien der ISO Norm 9241 (siehe DIN EN ISO 9241) beachtet werden müssen. Unter dem Titel „Ergonomie der Mensch-System-Interaktion“ beschreibt diese unterschiedliche Anforderungen an die Arbeitsumgebung sowie Hardware und Software, um gesundheitliche Schäden beim Arbeiten am Bildschirm zu vermeiden und dem Benutzer die Ausführung seiner Aufgaben zu erleichtern.

6.4 Integration in GUI-Builder

GUI-Builder sind Tools, mit denen grafische Oberflächen für eine Programmiersprache vom Entwickler mittels dem WYSIWYG-Ansatz („What you see is what you get“) relativ einfach zusammengestellt werden können. Spezielle Tools für Java bieten dabei eine Widget-Palette von AWT, Swing oder SWT an, die per Drag and Drop arrangiert werden. Aus diesem, in früheren Fassungen meist proprietärem Konstrukt, erstellt das Tool nach Beendigung den eigentlichen Java-Quelltext. Ein Beispiel dafür wäre der „WindowBuilder“ der das bisherige „Visual Editor Project“ weitestgehend abgelöst hat. Mit diesem sind auch bidirektionale Übersetzungen, also vom Java-Quelltext zur Design-Ansicht und zurück, möglich. Derartige Anwendungen werden zum Beispiel für GUI-Prototyping, dem schnellen Entwerfen und Testen von Prototypen grafischer Oberflächen, verwendet [Bäumer et al., 1994].

Die Ansätze mittels Precompiler, Präprozessoren und Buildscripten bauen auf handgeschriebenen Befehlselementen und Codemodifikationen auf. Daher ist diese Nutzung in einer breiten Masse von GUI-Buildern, wenn überhaupt, nur sehr schwer denkbar.

Die Umsetzung mittels Interfaces könnte bedingt mit GUI-Buildern umgesetzt werden. Dazu wird allerdings vorausgesetzt, dass das Tool die Modifikation des erzeugten Quelltextes zulässt. In diesem Fall könnten die relevanten Widgets durch

die Interfaces ausgetauscht werden. Allerdings könnten hierbei Probleme beim Rückübersetzen von Quelltext zur Design-Ansicht bei bidirektionalen Tools auftreten, da diese in der Regel die selbsterstellten Interfaces nicht interpretieren.

Eine Neukonzeption einer Komponente könnte in modernen Design-Tools umgesetzt werden, da diese meist auch benutzerdefinierte Widgets erzeugen können. Sollte dies nicht der Fall sein, bestünde noch die Möglichkeit, die neue Komponente über ein Standard-Interface von Swing oder SWT ansprechbar zu gestalten und damit dem GUI-Builder „vorzugaukeln“, eine Swing oder SWT Komponente vor sich zu haben. Hierbei wären allerdings bei den Kompilierungsvorgängen manuelle Schritte nötig, da die Package-Definitionen der GUI-Builder durch die eigenen ersetzt werden müssen.

Kapitel 7

7 Zusammenfassung und Ausblick

In dieser Arbeit wurde auf die Plattformunabhängigkeit spezieller Frameworks zu Erstellung von grafischen Oberflächen in Java eingegangen. Dazu wurden zunächst in Kapitel 2 unter anderem auf die beiden zur Zeit geläufigsten Frameworks Swing und SWT eingegangen.

7.1 Zusammenfassung der Arbeit

Swing baut dabei auf einem kleinen Teil des AWT Frameworks auf, welches die Basis für die Ereignisbehandlung legt und den Zugriff auf grundlegende Zeichenoperationen des jeweiligen Systems bietet. Höhergradige Widgets, wie Schaltflächen, Texteingabefelder oder Tabellen, werden durch geeignete Operationen selbst gezeichnet. Die sogenannten UI-Delegates sorgen dabei dafür, dass sich ein nachgebautes Widget optisch nicht sonderlich von dem Pendant des Betriebssystems unterscheidet und ebenso funktioniert. Durch die nötigen Zeichenoperationen wurde eine Swing-Oberfläche in den Anfangsjahren allerdings als sehr träge empfunden. Diverse Updates brachten hier aber einen signifikanten Performanceschub. Swing baut zudem auf einer vereinfachten Form des MVC Pattern auf. Das Model-Delegate Pattern nutzt ebenfalls ein Modell für die Datenverwaltung, verbindet allerdings die grafische Ansicht (View) und Logik (Controller) einer Komponente miteinander. Dadurch wird zum einen eine strikte Trennung von Datenverwaltung, Ausgabe und Logik geschaffen, zum anderen ist die technische Umsetzung einfacher zu realisieren als beim eigentlichen MVC Pattern.

Als zweites Framework wurde das SWT untersucht. SWT nutzt im Gegensatz zu Swing sogenannte Peer-Klassen und bedient sich damit den Widgets, die das Betriebssystem selbst liefert. Ist ein Widget nativ auf Seiten des Betriebssystems nicht vorhanden, kann es in der nativen SWT-Bibliothek emuliert werden. Durch die Auslagerung der Widgets auf die Betriebssystemseite und dem damit einhergehenden Verzicht der Speicherverwaltung von Java wird SWT auch als heavyweight Framework bezeichnet. Eine einheitliche Struktur zur Verwaltung der Daten, wie es das MVC Pattern bei Swing ist, wurde erst mit der Erweiterung JFace

nachgeliefert. JFace bietet den Entwicklern eine einheitliche Abstraktionsschicht mit zusätzlichen Funktionen, wie Eingabe-Validation oder Widget-Dekorationen, die zum Beispiel eine fehlerhafte Eingabe mittels entsprechendem Icon hervorheben.

Durch die enge Bindung von SWT an das zugrunde liegende Betriebssystem wurde an verschiedenen Stellen wissentlich auf die durch Java gegebene Plattformunabhängigkeit verzichtet. In Kapitel 3 wurden drei verschiedene Arten untersucht, in denen sich SWT-Bibliotheken auf verschiedenen Plattformen unterscheiden. Mit geeigneten Methoden wurden diese exemplarisch herausgestellt. SWT unterscheidet sich grob in den Klassen selbst, so dass bestimmte Widgets auf einigen Plattformen nicht vorhanden sind. Weiterhin existieren Unterschiede in den Klassen selbst, die sich zum Beispiel in nicht implementierten Methoden zeigen. Der letzte Punkt betrifft die Funktionsweise einiger Methoden selbst, die in verschiedenen Umgebungen unterschiedliche Resultate liefern. Dies führt dazu, dass Entwickler, die Produkte zum Beispiel für Windows und Linux schreiben, zusätzlich die Funktionsweise der GUI unter den verschiedenen Systemen testen müssen.

Um diese Differenzen in eigenen Produkten zu beseitigen, wurden im Kapitel 4 verschiedene Methoden vorgestellt, Anwendungsbeispiele sowie Vor- und Nachteile benannt. Alle Ansätze basieren darauf, dass für eine SWT-Differenz ein entsprechender Workaround existiert. So kann zum Beispiel die fehlende Status-Angabe für Fortschrittsanzeigen unter Nicht-Windows-Systemen durch ein einfaches Setzen der Vordergrundfarbe erfolgen.

Zunächst bediente man sich bei sprachinternen Konstrukten. Dazu zählen Precompiler und Interfaces. In der Sprache Java existieren keine Precompiler, wie sie zum Beispiel aus C/C++ bekannt sind. Allerdings kann sich der Umstand zunutze gemacht werden, dass der Compiler Codeblöcke entfernt, wenn diese unter keinen Umständen erreicht werden können. Dadurch können betriebssystemspezifische Behandlungen von SWT-Missständen durchgeführt werden. Ähnlich dazu können diese Codeabschnitte mittels Interface an den geeigneten Stellen eingebunden werden. Die plattformspezifische Lösung würde in einem plattformabhängigen Paket mitgeliefert werden, ähnlich also der SWT-Bibliothek an sich.

Des weiteren wurde der Anwendungsumfang von Präprozessoren und Buildscripten untersucht. Diese bedienen sich spezieller Codewörter im Quelltext, die im einfachsten Fall ersetzt oder mit anderen Konstrukten überschrieben werden. Die Codewörter sind meist in speziellen Kommentaren versteckt, um Probleme mit dem eigentlichen Java-Compiler zu vermeiden.

Der letzte vorgestellte Ansatz bedient sich der aspektorientierten Programmierung. Dabei wird der Java-Bytecode an vordefinierten Stellen modifiziert, so dass zum Beispiel vor einem Methodenaufruf eine Überprüfung stattfindet. Der Vorteil hierbei ist, dass an dem ursprünglichen Quelltext keinerlei Änderungen vorgenommen werden müssen und der Entwickler kein Wissen um die SWT-Differenzen besitzen muss.

Allen bisher vorgestellten Methoden ist allerdings gemein, dass sie noch immer auf die Widgets des Betriebssystems zurückgreifen. Eine Anpassung des Ausse-

hens oder der verwendeten Ereignisse ist damit nur sehr schwer möglich. Daher wurde in Kapitel 5 eine weitere Möglichkeit vorgestellt. Diese bedient sich dem Konzept von Swing und überträgt es auf SWT. Gemeint ist, dass von SWT lediglich der kleinste gemeinsame Nenner verwendet wird und darauf aufbauend alle höhergradigen Widgets selbst gezeichnet und verwaltet werden. Dazu wurde beschrieben, wie auf Basis des MVC Pattern eine Tabelle konzipiert wurde, die auf keine nativen Widgets mehr zugreift. Diese Vorgehensweise hat nun den Vorteil, dass Features, die aus Swing bekannt sind, wie zum Beispiel unterschiedliche Look and Feels, auch unter SWT verwendet werden können. Grundlegende Probleme, wie Differenzen in der Ereignisbehandlung konnten damit weitestgehend behoben werden. Diese haben sich unter anderem in Ereignissen gezeigt, die unter Windows und Linux in unterschiedlicher Art und Weise generiert wurden. Ein entscheidender Nachteil des Verfahrens sind allerdings die sehr hohen Kosten (finanziell und zeitlich), die bei einer Erstkonzeption und -umsetzung auf Betroffene zukommen können.

7.2 Verwendung und Ausblick

Die in Kapitel 5 in Auszügen vorgestellte Komponente wird in einem Set von weiteren Widgets, die alle auf dem gleichen Konzept basieren, bereits bei diversen Projekten eingesetzt. Zu den weiteren Widgets zählen unter anderem Labels, DatePicker und Bäume. Die Tabelle existiert zur Zeit in der zweiten Version und wurde um diverse Features gegenüber den Standard-Widgets von Windows und Linux erweitert. So können Editoren für Spaltenköpfe definiert werden, die für benutzerdefinierte Aktionen der ganzen Spalte verwendet werden können. Die Tabelle kann in einen Editiermodus versetzt werden, bei dem verschiedene Attribute der Tabelle, Spalten und Zeilen verändert werden können. Ein erweitertes Konzept ist in Abbildung 7.1 dargestellt.

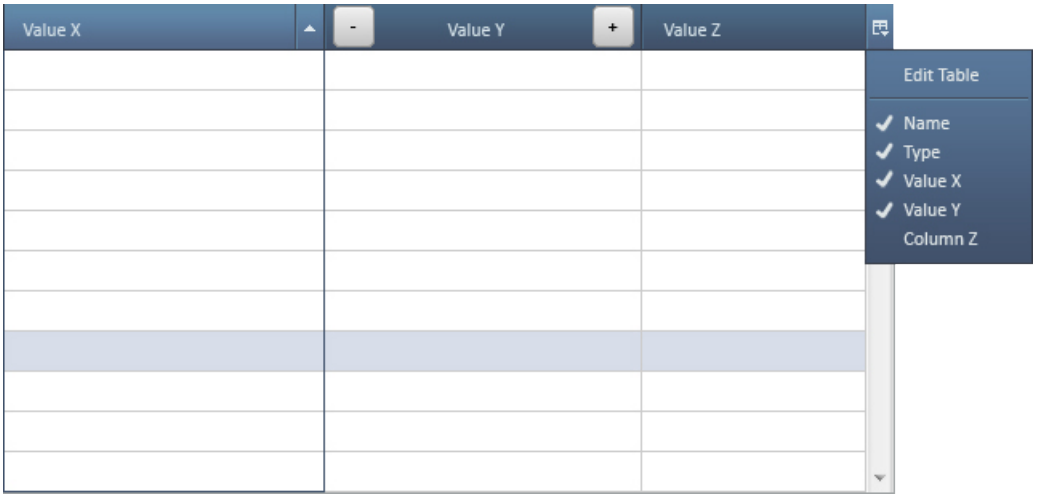

Abbildung 7.1: Tabellen-Konzept mit erweitertem Funktionsumfang

Zu sehen ist eine leere Tabelle mit drei Spalten. Die zweite Spalte „Value Y“

enthält den bereits erwähnten Editor im Spaltenkopf. Damit können mit einem einmaligen Klick alle Werte der Spalte dekrementiert beziehungsweise inkrementiert werden. Ferner ist ein Kontextmenü im oberen rechten Feld zu sehen, über das Spalten ein und ausgeblendet und die Tabelle in den Editiermodus versetzt wird.

Der Neuentwurf der Widgets hat in diesem Fall eine positive Entwicklung vollzogen und bietet noch Potenzial für Erweiterungen der Widgets. Die Art der Umsetzung bietet den Vorteil, auf spezielle Kundenbedürfnisse angepasst werden zu können und dennoch nicht auf Rich-Client-Plattformen verzichten zu müssen, die das SWT zwingend voraussetzen.

Literaturverzeichnis

[Allen 1997] Allen, Robert J.: *A Formal Approach to Software Architecture.* 1997

[Bartolomei et al. 2010] Bartolomei, Thiago T. ; Czarnecki, Krzysztof ; Lämmel, Ralf: Swing to SWT and back: Patterns for API migration by wrapping. In: *2012 28th IEEE International Conference on Software Maintenance (ICSM)* 0 (2010)

[Bishop 2006] Bishop, Judith: Multi-platform user interface construction: a challenge for software engineering-in-the-small. In: *Proceedings of the 28th international conference on Software engineering.* New York, NY, USA : ACM, 2006 (ICSE '06)

[Blom et al. 2008] Blom, Soren ; Book, Matthias ; Gruhn, Volker ; Hruschchak, Ruslan ; Köhler, Andre: Write Once, Run Anywhere - A Survey of Mobile Runtime Environments. In: *Grid and Pervasive Computing Workshops, 2008. GPC Workshops '08. The 3rd International Conference on*, 2008

[Borchers 2000] Borchers, Jan O.: Interaction Design Patterns: Twelve Theses. In: *Workshop on Pattern Languages for Interaction Design, CHI 2000 Conference on Human Factors in Computing Systems* Bd. 2, 2000

[Brockschmidt 1995] Brockschmidt, Kraig: *INSIDE OLE Second Edition.* Microsoft Corporation, 1995 (Microsoft programming series)

[Bäumer et al. 1994] Bäumer, Dirk ; Bischofberger, Walter R. ; Lichter, Horst ; Schneider-Hufschmidt, Matthias ; Sedlmeier-Scholz, Veronika ; Züllighoven, Heinz: *Prototyping von Benutzungsoberflächen.* Hamburg : Univ. Bibliothek des Fachbereichs Informatik, 1994

[Burbeck 1987] Burbeck, Steve: *Applications Programming in Smalltalk-80: How to Use Model-View-Controller (MVC).* Softsmarts, Incorporated, 1987

[Daum 2004] Daum, Berthold: *Java-Entwicklung mit Eclipse 3: Anwendungen, Plugins und Rich Clients.* Dpunkt.Verlag GmbH, 2004 (Programmer to programmer)

[Debray et al. 2000] Debray, Saumya K. ; Evans, William ; Muth, Robert ; Sutter, Bjorn D.: Compiler techniques for code compaction. In: *ACM Trans. Program. Lang. Syst.* 22 (2000), März, Nr. 2

[Dijkstra 1976] Dijkstra, Edsger W.: *A Discipline of Programming.* Prentice Hall, Inc., 1976

[DIN EN ISO 9241 2006] *Ergonomie der Mensch-System-Interaktion.* 2006

[Ernst et al. 2002] Ernst, Michael D. ; Badros, Greg J. ; Notkin, David: An Empirical Analysis of C Preprocessor Use. In: *IEEE Transactions on Software Engineering* 28 (2002), Nr. 12

[Feldman 1979] Feldman, Stuart I.: Make — a program for maintaining computer programs. In: *Software: Practice and Experience* 9 (1979), Nr. 4

[Friese et al. 2007] Friese, Peter ; Lippert, Martin ; Seeberger, Heiko: Eclipse und Security Aspekt-orientiert - Security Does Matter. In: *Eclipse Magazin* 12 (2007), Nr. 12

[Gamma et al. 1993] Gamma, Erich ; Helm, Richard ; Johnson, Ralph E. ; Vlissides, John M.: Design Patterns: Abstraction and Reuse of Object-Oriented Design. In: *ECOOP*, 1993

[Geer 2005] Geer, David: Eclipse becomes the dominant Java IDE. In: *Computer* 38 (2005), Nr. 7

[Gosling et al. 2013] Gosling, James ; Joy, Bill ; Steele, Guy ; Bracha, Gilad ; Buckl, Alex: *The Java Language Specification*
Java SE 7 Edition. 500 Oracle Parkway, Redwood Shores, CA 94065 : Spezifikation, Februar 2013

[Gruntz 2004] Gruntz, Dominik: Java Design: On the Observer Pattern / University of Applied Sciences, Aargau. 2004. – Forschungsbericht

[Kiczales et al. 2001] Kiczales, Gregor ; Hilsdale, Erik ; Hugunin, Jim ; Kersten, Mik ; Palm, Jeffrey ; Griswold, William G.: An Overview of AspectJ. In: *Proceedings of the 15th European Conference on Object-Oriented Programming.* London, UK, UK : Springer-Verlag, 2001 (ECOOP '01)

[Kiczales et al. 1997] Kiczales, Gregor ; Lamping, John ; Mendhekar, Anurag ; Maeda, Chris ; Lopes, Cristina ; Loingtier, Jean marc ; Irwin, John: Aspect-Oriented Programming. In: Aksit, Mehmet (Hrsg.) ; Matsuoka, Satoshi (Hrsg.): *ECOOP '97 - Object-Oriented Programming: 11th European Conference* Bd. 1241. 1997

[Kramer 1996] Kramer, Douglas: *The Java Platform - A White Paper.* Mai 1996

[Lahres und Rayman 2009] Lahres, Bernhard ; Rayman, Gregor: *Objektorientierte Programmierung: das umfassende Handbuch.* Galileo Press, 2009

[von Lavergne 2004] Lavergne, Harro von: Objekte, Klassen, Module, Kontrakte und Komponenten. In: *LOG IN* 24 (2004), Nr. 131/132

[MacLeod und Northover 2001] MacLeod, Carolyn ; Northover, Steve: *SWT: The Standard Widget Toolkit*
PART 2: Managing Operating System Resources. November 2001

[Madeyski und Stochmialek 2005] Madeyski, Lech ; Stochmialek, Michal: Architectural design of modern web applications. In: *Foundations of Computing and Decision Sciences* 30 (2005), Nr. 1

[Memon 2002] Memon, Atif M.: GUI Testing: Pitfalls and Process. In: *Computer* 35 (2002), August, Nr. 8

[Memon et al. 2001] Memon, Atif M. ; Soffa, Mary L. ; Pollack, Martha E.: Coverage criteria for GUI testing. In: *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*. New York, NY, USA : ACM, 2001 (ESEC/FSE-9)

[Northover und Wilson 2004] Northover, Steve ; Wilson, Mike: *SWT - The Standard Widget Toolkit*. Volume 1. ADDISON WESLEY Publishing Company Incorporated, 2004

[Scarpino et al. 2005] Scarpino, Matthew ; Holder, Stephen ; Ng, Stanford ; Mihalkovic, Laurent: *SWT/JFace In Action*. Dreamtech Press, 2005

[Sun Microsystems 2001] Sun Microsystems: *Java Look and Feel Design Guidelines: Advanced Topics*. Boston, MA : Addison-Wesley, 2001

[Ullenboom 2008] Ullenboom, Christian: *Java ist auch eine Insel - Programmieren mit der Java Platform, Standard-Edition 6*. 7. Auflage. Bonn : Galileo Press, 2008

[Völter 2005] Völter, Markus: Modellgetriebene, Komponentbasierte Softwareentwicklung - Teil 1. In: *JavaMagazin* (2005), Nr. 10

[Völter und Lippert 2004] Völter, Markus ; Lippert, Martin: Die 5 Leben des AspectJ. In: *JavaSpektrum* (2004), Nr. 3

[Wieland und Wallenburg 2012] Wieland, Andreas ; Wallenburg, Carl M.: Dealing with supply chain risks: Linking risk management practices and strategies to performance. In: *International Journal of Physical Distribution and Logistics Management* 42 (2012), Nr. 10

[Wood 1997] Wood, Larry E.: *User Interface Design: Bridging the Gap from User Requirements to Design*. CRC, 1997

[Zukowski 1997] Zukowski, John: *Java AWT reference*. Sebastopol, CA, USA : O'Reilly & Associates, Inc., 1997

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, den 28. April 2013

Gordon Damerau