Entwicklung plattformunabhängiger GUI-Anwendungen mit DSLs

Karsten Pietrzyk
Matrikelnummer: 1014617

25.10.15

Masterarbeit an der Christian-Albrechts-Universität zu Kiel

Institut für Informatik

Fachbereich Programmiersprachen und Übersetzerkonstruktion

Erstgutachter: Prof. Dr. Michael Hanus
Zweitgutachter: Dipl.-Inf. Thomas Stahl

Zusammenfassung

Die Programmierung von grafischen Benutzeroberflächen (GUIs) ist industriell relevant, aber softwaretechnisch nicht einfach, da eine Vielzahl an Frameworks und Plattformen zur Verfügung stehen. Diese erfordern Einarbeitungszeit und haben konzeptuelle und implementierungsspezifische Unterschiede. Durch Architekturmuster wie Model-View-ViewModel (MVVM) können diese Unterschiede vereinheitlicht und zusätzlich GUI- und Logik-Zuständigkeiten im Quellcode getrennt werden. So lassen sich wartbare und erweiterbare Anwendungen entwickeln.

Zur Vereinfachung der Entwicklung wurden weiterhin Beschreibungssprachen für GUIs geschaffen, die Eigenschaften und Layout von Controls spezifizieren. Diese Beschreibungssprachen sind jedoch weitestgehend Framework-spezifisch und meist nur für eine Plattform verwendbar, beispielsweise XAML für WPF unter Windows, und FXML für JavaFX.

Eine plattformunabhängige Beschreibung wäre wünschenswert, damit GUIs von Anwendungen für einzelne Plattformen nicht neu entwickelt werden müssen.

Das Ziel dieser Masterarbeit ist, eine Bestandsaufnahme der GUI-Beschreibungssprachen und plattformunabhängigen Ansätze zu erstellen und daraus eine plattformunabhängige DSL für GUI-Anwendungen zu entwickeln. Die DSL und die dazugehörigen Generatoren werden mit Fallbeispielen evaluiert.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1 Einführung

1.1 Ausgangssituation und Problemstellung

Die Entwicklung von Anwendungen mit grafischen Benutzeroberflächen (GUIs) ist industriell relevant, aber softwaretechnisch nicht einfach. Auf Smartphones und Desktop-Rechnern laufen Anwendungen mit Benutzeroberflächen, deren Spektrum von einfachen Stoppuhren bis zu umfangreichen Office-Anwendungen reicht.

Für die Entwicklung von GUI-Anwendungen sind gute Kenntnisse über GUI-Frameworks erforderlich. Diese unterscheiden sich in Klassenhierarchien und in ihrem Entwurf stark voneinander. Dazu kommen GUI-Design-Richtlinien, die eingehalten werden sollten und die von Plattform zu Plattform unterschiedlich sind (z.B. aufgrund der Größe des Bildschirmes oder der Eingabemethode).

Bei der Entwicklung von GUI-Anwendungen ist eine Trennung von Programm- und GUI-Logik empfehlenswert: Wenn unterschiedliche Zuständigkeiten vermischt werden, lässt sich die Anwendung schwierig erweitern und warten. Um diese Trennung einzuhalten, wurden Architekturmuster entwickelt. Eine genaue Definition von „Architekturmuster“ ist die Folgende:

> „Architectural patterns express fundamental structural organization schemas for software systems. They provide a set of predefined subsystems, specify their responsibilities, and include rules and guidelines for organizing the relationships between them.“ [Buschmann1996, S. 25]

Ein modernes Architekturmuster für GUI-Programmierung ist Model-View-ViewModel (MVVM), das eine Trennung in Model, View und ViewModel vorschreibt. Dieses Muster und seine Anwendung werden in Abschnitt 4.1 erläutert. Das Architekturmuster MVVM wurde in den Frameworks Windows Presentation Foundation (WPF) und Silverlight von Microsoft eingeführt und findet sich mittlerweile ebenfalls in anderen Frameworks wieder: etwa in dem JavaScript-Framework knockout.js [KnockoutJS] oder dem Android-Framework Robobinding [Robobinding].

Um die Entwicklung von GUI-Anwendungen zu vereinfachen, wurden Beschreibungssprachen geschaffen, mit denen der Entwickler Eigenschaften von Interaktionselementen (Controls) und ihre Anordnung (Layout) bestimmt. Diese Beschreibungssprachen sind für unterschiedliche Plattformen ausgelegt (Web, Mobil, Desktop, .NET, Java Virtual Machine) und sind verschieden ausdrucksstark, z.B. bei typischen Aufgaben wie Datenbindung.

Dem Entwickler stehen eine Vielzahl an Plattformen und Frameworks zur Verfügung, von denen einige bereits veraltet sind. Eine nicht vollständige Auflistung ist in Tabelle 1.1 dargestellt.

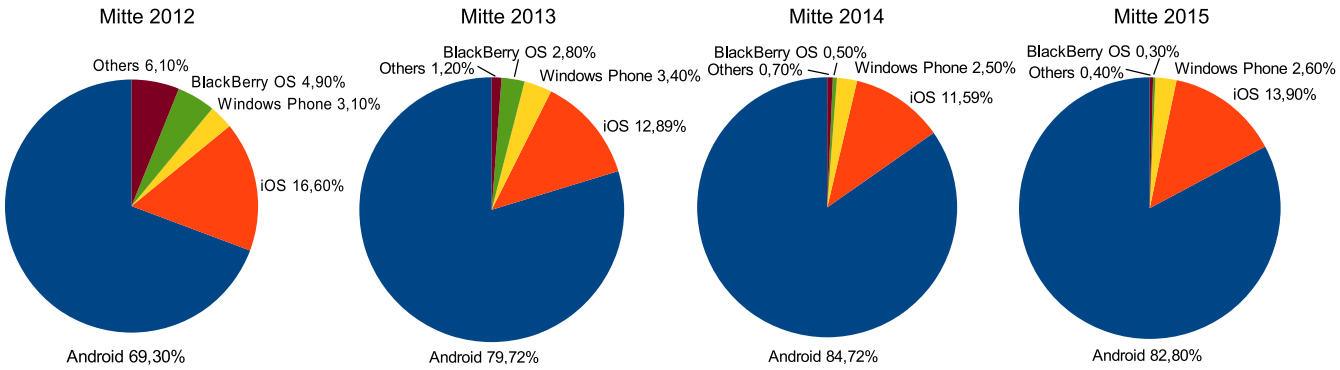


Abbildung 1.1: Marktanteile der Smartphone-Betriebssysteme von 2012 bis 2015 nach [MarketShareICD]

Plattform	Frameworks	veraltete Frameworks
Java	JavaFX, Swing	AWT
.NET	WPF	Forms
Web	HTML5, Flash	Silverlight

Tabelle 1.1: Plattformen und dazugehörige Frameworks inklusive veralteter Frameworks

Weil stets neue Frameworks entwickelt und dadurch andere verdrängt werden, ist die Frage berechtigt, wann das nächste Framework für Entwickler nicht mehr attraktiv ist. Ähnlich ist es bei Smartphone-Betriebssystemen, die durch den sich verändernden Markt unbedeutend werden können. Der Trend ist folgender: Android ist vorherrschend, iOS immerhin noch populär, während Windows Phone wesentlich weniger populär ist und Blackberry und andere nur einen marginalen Marktanteil haben. Dies wird in Abbildung 1.1 dargestellt.

Zusammenfassend besteht das Kernproblem darin, dass GUIs wichtig aber schwierig zu programmieren sind. Hinzu kommt, dass eine Vielzahl an heterogenen Plattformen und Frameworks existiert. Einige Probleme lassen sich durch folgende Konzepte lösen:

- **Beschreibungssprachen** vereinfachen GUI-Programmierung, indem sie auf einer höheren Abstraktionsebene arbeiten (nicht auf der Ebene der Implementierungssprache) und GUI-Zuständigkeiten von der Anwendungslogik trennen.
- **Architekturmuster wie MVVM** führen zur einer besseren Strukturierung der Anwendung, welche die Zuständigkeiten trennt. Damit wird die Anwendung wartbarer und besser erweiterbar.

1.2 Fragestellung und Zielsetzung

Der Gegenstand dieser Masterarbeit soll die Frage sein, inwiefern GUI-Beschreibungssprachen und plattformunabhängige Ansätze und die oben genannten Probleme lösen. Die Entwicklung einer plattformunabhängigen GUI-Beschreibungssprache für MVVM-Anwendungen ist das Ziel dieser Arbeit.

Diese Masterarbeit richtet sich an Leser, die mit den grundlegenden Konzepten des Compilerbaus und mit DSLs vertraut sind, tiefergreifendes Wissen ist aber nicht erforderlich. Das Verständnis objektorientierter Entwicklung und Vorerfahrung mit GUI-Programmierung wird hingegen vorausgesetzt.

Diese Arbeit entstand in Kooperation mit dem Unternehmen b+m Informatik AG, mit Fokus auf das Werkzeug b+m gear. Im Folgenden wird dieses Werkzeug und dessen Rolle für diese Arbeit dargestellt.

1.3 b + m gear

Das Unternehmen b + m Informatik AG benutzt für die interne Produktentwicklung das selbst entwickelte Werkzeug b + m gear [BMGear]. Dieses verwendet DSLs für verschiedene Aspekte der zu entwickelnden Anwendung: Unter anderem können Entitäten, Views und ViewModels beschrieben werden. Aus diesen Beschreibungen wird eine Web-Anwendung inklusive Backend generiert, die das Framework Vaadin [Vaadin] verwendet.

Die in dieser Arbeit entwickelten HTML- und Android-Generatoren werden in die Generatorkette von b + m gear integriert, sodass diese dort genutzt werden können. Dies ist dank der Sprachunabhängigkeit der Generatoren möglich, worauf in Abschnitt 5.8 eingegangen wird.

Die im Rahmen dieser Arbeit entwickelte Plain-DSL ist von der gear-DSL inspiriert und übernimmt einige Teile der GUI-Beschreibung. Andersherum können Teile der entwickelten Plain-DSL und der Überlegungen dahinter in die gear-DSL einfließen. In Abschnitt 7.1 findet sich eine Zusammenstellung wichtiger Überlegungen.

1.4 Vorgehensweise

Zuerst gehe ich auf den aktuellen Stand der Technik ein und dokumentiere verwandte Arbeiten und deren Ergebnisse.

Im Abschnitt „Grundlagen“ gebe ich eine kurze Einführung in das Architekturmuster Model-View-ViewModel (MVVM). Daraufhin definiere ich den Begriff der Plattformunabhängigkeit in für diese Arbeit. Anschließend gebe ich eine kurze Einführung in modellgetriebene Softwareentwicklung und DSLs.

Im nächsten Abschnitt evaluiere ich bestehende Beschreibungssprachen bekannter Frameworks (z.B. FXML des JavaFX-Frameworks und XAML des WPF-Frameworks) dahingehend, inwieweit die anfangs genannten Probleme gelöst werden. Dabei konzentriere ich mich auf folgende Kriterien und Aspekte:

- **Plattformen:** Für welche Zielplattformen ist die Beschreibungssprache gedacht?
- **Validierung:** Welche Unterstützung für Validierung von Eingaben bietet die Beschreibungssprache?
- **Datenbindung:** Unterstützt die Beschreibungssprache Datenbindungen?
- **Anpassbarkeit der GUI:** Inwiefern lassen sich GUI-Elemente visuell anpassen?
- **Anpassbarkeit des Generators:** Inwiefern lässt sich die Code-Generierung steuern?

In den darauffolgenden Abschnitten soll die Entwicklung einer eigenen plattformunabhängigen DSL für GUI-Anwendungen und der Generatoren im Vordergrund stehen. Die Entwicklung umfasst den Entwurf und die Implementierung. Hierfür wird die Architektur der Zielanwendung, die auf MVVM basiert, festlegt und erläutert. Für die Implementierung stelle ich die Möglichkeiten Parser-Generator oder Language Workbench gegenüber, beschreibe die DSL und erläutere die Designentscheidungen bei der Umsetzung der Generatoren.

Ich beschränke aus folgenden Gründen die Generatoren auf die Zielplattformen Android und Web (HTML):

- Es existiert eine Vielzahl an plattformspezifischen GUI-Frameworks mit Vor- und Nachteilen. Web-Programmierung ist unter anderem vorteilhaft, weil nicht erst Frameworks (z.B. .NET) oder andere Programme (z.B. Flash Player Plugin) installiert werden müssen. HTML hat den Vorteil, auf allen Geräten mit einem Browser zu laufen: Sowohl Desktop- als auch Mobil-Gerät-Benutzer können HTML-Anwendungen verwenden. Zudem besitzt HTML mit geeigneten Bibliotheken ein umfangreiches GUI-Spektrum.

- Android ist wie eingangs erwähnt und in Abbildung 1.1 dargestellt das populärste Smartphone-Betriebssystem. Native Android-Programmierung ist dank Java und XML jedem zugänglich, der diese beiden Sprachen kennt. Das Android-Framework und die Programmier-Werkzeuge sind frei verfügbar und erstellte Anwendungen lassen sich leicht auf Android-Geräte übertragen und ausführen.

Im Anschluss wird die DSL mithilfe von Fallbeispielen evaluiert. Zuletzt bewerte ich den Impact von Plattformunabhängigkeit auf DSLs und gehe in einem Fazit auf Probleme und Möglichkeiten des Ansatzes ein.

1.5 Stand der Technik

Die Idee, mit einem Framework für mehrere Plattformen zu entwickeln, ist für Unternehmen sehr reizvoll. Dadurch können beispielsweise Anwendungen (Apps) für eine größere Anzahl an Smartphone-Betriebssystemen veröffentlicht und verkauft werden. Bei der Entwicklung für ein einzelnes Betriebssystem fällt Portierungsaufwand auf andere Betriebssysteme an.
Bei der Entwicklung von Spielen für mobile Geräte ist dieses Vorgehen sehr beliebt und weit verbreitet (Abschnitt 1.8).

Die modellgetriebene Softwareentwicklung, die von einigen Unternehmen praktiziert wird, verwendet Modellbeschreibungen (z.B. DSLs) und plattformübergreifende Generierungsmethoden, die für mehrere spezifische Framework Anwendungsteile generieren können.

In der Wirtschaft werden diese Aspekte zum Teil voneinander getrennt verwendet: Unternehmen wie b + m Informatik AG setzen den Fokus auf modellgetriebene Entwicklung mit DSLs und Code-Generatoren.

Unternehmen wie Microsoft legen den Fokus auf Cross-Platform-Frameworks wie Xamarin, um für mehrere Plattformen zu entwickeln. Beispielsweise wird damit das Management-Produkt Intune entwickelt und ist ebenfalls auf Android-Geräten verfügbar [XamarinCustomers].

In einigen der folgenden Ansätze werden DSL- und Cross-Platform-Ansätze kombiniert, um eine neue Möglichkeit der plattformunabhängigen Entwicklung von Anwendungen zu beschreiben.

1.6 Verwandte Arbeiten

Ein Großteil der hier betrachteten wissenschaftlichen Ansätze beschränkt sich auf mobile Geräte, weil diese zu einem wichtigen Thema innerhalb der Informatikforschung und im täglichen Leben geworden sind. Daher ist es beliebt, für mehrere Mobilgeräte zu entwickeln, was in den Arbeiten auch als „Cross Platform“ bezeichnet wird, wobei die Bezeichnung „Cross Mobile“ treffender ist. Andere Autoren ignorieren Mobilgeräte und suchen nach plattformunabhängigen Entwicklungsmethoden, die nur für Desktop-Anwendungen ausgelegt sind. Nur wenige Ansätze kombinieren diese beiden Aspekte.

In einigen der untersuchten wissenschaftlichen Arbeiten wird die Frage gestellt, wie man eine Anwendung beschreiben kann, ohne sich auf eine konkrete Plattform zu beschränken. Häufig wird eine Untermenge der verfügbaren Controls einer bestimmten Plattform als Basis verwendet, etwa TextBoxes, CheckBoxes, Buttons, Labels und eine Möglichkeit, diese Controls anzuordnen. Zur besseren Übersicht werde die untersuchten Arbeiten danach gruppieren, ob sie einen modellgetriebenen Ansatz verfolgen oder nicht.

1.6.1 Modellgetriebene Ansätze

Modellgetriebene Ansätze zur plattformunabhängigen Programmierung verwenden Diagramme der Unified Modelling Language (UML) oder DSLs zur Modellierung.

Beispielweise setzt **XIS-Mobile** [Ribeiro2014] vorwiegend UML-Klassendiagramme ein, um die Domäne zu modellieren; also unter Verwendung von Klassen, Assoziationen und Vererbung. Aus Use-Case-Diagrammen

```
Application root "C:\\" titre "English Grammar Quiz" version "V 1.0",

Page title "page_1" {
  panel [ id 1 type "simple" orientation 'horizontal' header "Question 1 from 2" ] {
      panel [ id 2 type "grid" orientation 'vertical'  columns "1"] {
          label [ id 1  text 'I_____a new car last month. ' ] ,
          groupe [ id 2 type "radio"  orientation "vertical" ] {
            radio [ id 1  text 'buyed' ] ,
            radio [ id 2  text 'bought' ],
            radio [ id 3  text 'have bought' ],
            radio [ id 4  text 'did bought' ]
          } ,
          panel [ id 3 type "simple" orientation 'horizontal' columns "2" ] {
            button [ id 1  text 'Next Question >>' ]
          }
      }
  }
}
```

Abbildung 1.2: Darstellung einer Englisch-Quiz-App in einer GUI-DSL [Abdali2015, S. 30]

können Views für die Mobile-Plattformen Windows Phone und Android abgeleitet werden, wobei in der Arbeit nicht genauer auf dieses Verfahren eingegangen wird. Die Entwicklung mit dem vorgestellten Framework geschieht nur bis zu einem gewissen Punkt in UML: Die generierten C#- oder Java-Projekte sind für die weitere Bearbeitung in einer passenden Entwicklungsumgebung (IDE) gedacht. Dies erschwert eine Anpassung des Modells und erneute Generierung, da der bis dahin manuell implementierte Code übernommen werden soll. Dieser Aspekt des Frameworks wird im Paper folgendermaßen beschrieben:

> „XIS-Mobile [...] represent[s] a helpful tool that generates the skeleton of a mobile application. Namely, XIS-Mobile will generate the boilerplate code that represents the great majority of the application's code. From that point on, the developer has access to the generated code and can customize it if it does not fully fulfill his needs. For instance, he could need to improve the GUI or implement the custom actions attached to a certain widget." [Ribeiro2014, S. 1322]

In [Abeck2007] wird angemerkt, dass die Möglichkeiten der UML ohne Erweiterungen nicht immer ausreichend sind. In dieser Arbeit werden **GUIs aus erweiterten Aktivitätsdiagammen** erzeugt, wobei die UML-Aktivitätsdiagamme um Eingabe-Pins erweitert sind. Eingaben wie `passwort : String`, die an Aktivitäten angegeben werden können, erzeugen dafür passende Textfelder.

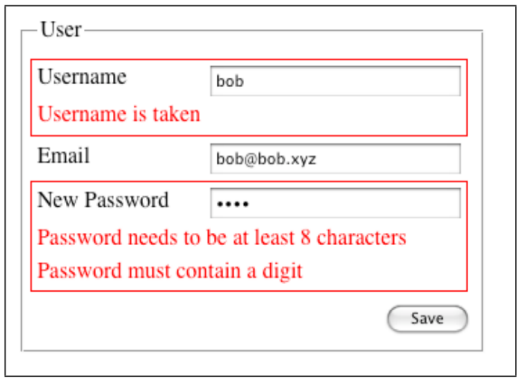
Der Ansatz der **Cross-Mobile-DSL XMOB** [LeGoaer2013] ist dadurch motiviert, dass zwar bestehende Cross-Mobile-Frameworks und -Werkzeuge keine neue Programmiersprache erfordern, aber die Schwierigkeit von nativer Programmierung bestehen bleibt. Zur Lösung dieses Problem wird eine DSL entwickelt, die Projekte der jeweiligen mobilen Plattform generiert (iOS, Android, Windows Phone und Blackberry OS). Die Projekte müssen mit der entsprechenden IDE kompiliert werden. Ein Vorteil bei diesem Ansatz ist, dass typische Aufgaben für Mobilgeräte, wie das Senden und Empfangen von SMS in der Mobile-DSL formuliert werden können. Gleichzeitig ist damit aber ausgeschlossen, dass die DSL für Desktop-Geräte verwendet werden kann.

In einer **GUI-DSL** [Abdali2015] werden Projekte für Java Server Faces und Android generiert. Hervorzuheben ist, dass dieser Ansatz nicht nur mobile Plattformen, sondern auch die Web-Plattform bedient. Leider wird hier nur GUI-Code aber kein Quelltext für Klassen generiert. Insbesondere muss Logik wie Navigation manuell implementiert werden. Zudem ist die DSL sehr technisch motiviert und beschreibt die Struktur der GUI ähnlich ausführlich wie vergleichbare XML-Formate, etwa Android-XML. In Abbildung 1.2 wird die Beschreibung einer Beispiel-Anwendung dargestellt. Es fällt auf, dass sinnvolle Standardwerte verwendet werden könnten, um die DSL-Beschreibung kürzer zu halten, beispielsweise für `id` und Attribute der `panel`-Elemente.

Die **WebDSL** [Groenewegen2013] ist für Web-Anwendungen konzipiert und wird in HTML-Seiten mit JavaScript übersetzt. In der zitierten Arbeit liegt der Fokus auf Daten-Validierung, einem wichtigen Aspekt der GUI-Interaktion, da das Generieren von Controls nicht ausreichend ist. Daten müssen beim Programmstart angezeigt und bei Speicher-Operationen validiert und gesichert werden. Bei fehlgeschlagenen Validierungen ist es notwendig, Fehlernachrichten und -markierungen mit hilfreichen Hinweisen anzuzeigen.
In diesem Ansatz werden Validierung z.B. mit regulären Ausdrücken umgesetzt (für Email- oder Telefon-Eingabefelder). Auch logische Prädikate auf den zugrundeliegenden Daten in einer DSL-eigenen Ausdrucks-

```
entity User {   username :: String (id)   password :: Secret   email :: Email   }
```

```
extend entity User {
  username(validate(isUnique(),"Username is taken"))
  validate(password.length >= 8, "Password needs to be at least 8 characters")
  validate(/[a-z]/.find(password), "Password must contain a lower-case character")
  validate(/[A-Z]/.find(password), "Password must contain an upper-case character")
  validate(/[0-9]/.find(password), "Password must contain a digit")
}
```




```
define page editUser(u:User) {
  form {
    group("User") {
      label("Username"){ input(u.username) }
      label("Email"){ input(u.email) }
      label("New Password") {
        input(u.password)
      }
      action("Save", save())
    }
  }
  action save() {
    return user(u);
  }
}
```

Abbildung 1.3: Beispiel einer HTML-Anwendung mit Validierung in der WebDSL [Groenewegen2013, S. 4]

sprache wie `user.age >= 18` werden unterstützt.
Bei der Modellierung stehen Datentypen aus der Web-Domäne zur Verfügung, etwa Email, URL, WikiText und Image. Ein Lokalisierungsmechanismus wird nicht unterstützt.
Abbildung 1.3 zeigt ein Beispiel mit der dazugehörigen HTML-Ausgabe im Browser.

Bei der plattformunabhängigen Modellierung muss die Frage geklärt werden, ob die Anwendungslogik lokal oder auf einem Server ablaufen soll. Dazu werden in [Achilleos2011] GUIs mit **Fokus auf Web-Service-Anbindung** generiert. Hier werden die Plattformen Java, J2ME, Android, Windows Mobile und Windows Desktop verwendet. Die Beschreibung der GUI erfolgt mit eigener Klassenhierarchie wie beispielsweise bei [Abdali2015]. Die GUI nutzt dann programmiersprachenunabhängige Web-Services, die von internetfähigen Geräten aufgerufen werden können. Der Ansatz wird mit einer Book-Store-Anwendung evaluiert, wobei etwa 56-74% Code der jeweiligen Zielplattform generiert wird.

In [Botturi2012] wird **Android- und Windows-Phone-Code aus UML mit der Object Constraint Language (OCL)** erstellt. Dazu wird ein plattformunabhängiges Modell entwickelt. Oberflächen werden durch UML-Objektdiagramme und Navigation durch Zustandsdiagramme beschreiben, wobei ein Großteil der Klassen und Eigenschaften ähnlich wie die Android-Pendants benannt sind. Für das Layout werden relative Angaben wie diese verwendet:
`label1.above = button2; button3.toRightOf = button4`
Die Generatoren werden über eine Beispiel-Anwendung evaluiert, die Informationen bei einzelnen Stationen im Museum bereitstellt. Die Anwendung enthält über die Standard-Controls hinaus eine Kartenansicht und einen QR-Code-Scanner.
Im Vergleich mit anderen Ansätzen wird hier neben dem GUI-Code auch die Navigationslogik generiert, inklusive anderer Artefakte wie Manifest und String-Ressourcen. Für Windows-Phone werden XAML und C#-Dateien erzeugt, allerdings wird das Layout absolut mit DPI-Angaben positioniert und ist damit sehr statisch.

Auch [Heitkötter2013] verfolgt einen Cross-Mobile-Ansatz. Dort werden **Android- und iOS-Anwendungen aus einer Mobile DSL generiert**. Der Ansatz nennt sich MD^2 und steht für „Model-driven mobile development“. Für das Frontend werden Android- und iOS-Projekte generiert, für das Backend ein Web-Service-Projekt (Java API for RESTful Services, JAX-RS).
Das Frontend wird nach dem Model-View-Controller-Architekturmuster (MVC) modelliert und ermöglicht Datenbindung und Validierung. Die Model-Daten werden an das Backend in der JavaScript Object Notation

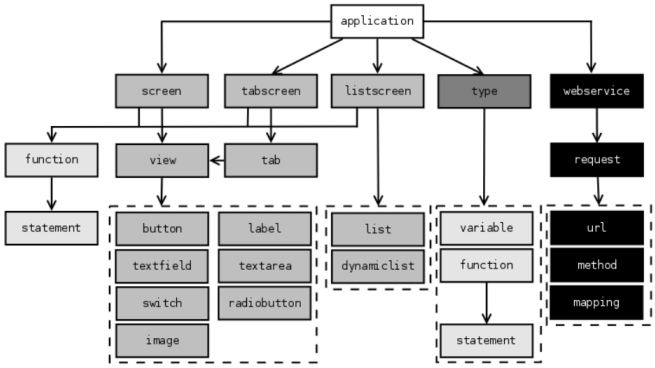


Abbildung 1.4: Darstellung der Elemente einer Cross-Mobile-Programmierprache [Rosén2013, S. 32]

(JSON) übertragen und in einer Datenbank gespeichert.
Dieser Ansatz zeichnet sich dadurch aus, dass aus einer plattformunabhängigen Beschreibung eine Anwendung generiert wird, die aus Frontend und Backend besteht. Die Trennung von Zuständigkeiten über das MVC-Architekturmuster ist sehr sinnvoll.
Die Autoren merken jedoch an, dass dieser Ansatz einige Beschränkungen hat: Der Fokus liege auf „data-driven business apps with a UI mainly consisting of form fields“ [Heitkötter2013, S. 8]. Weiterhin muss Geschäftslogik, die nicht über die DSL beschrieben werden kann, auf dem Server umgesetzt werden. Die Geräte-spezifischen Features der DSL umfassen derzeit nur GPS-Lokalisierung.

Nach [Rosén2013] soll die gesamte Anwendung in einer **Cross-Mobile-Programmiersprache** geschrieben werden. Das führt dazu, dass GUI-Beschreibung und Logik in die GUI-Ebene verschoben werden, wie in Listing 1.1 gezeigt: Weil Methoden-Aufrufe auf derselben Ebene passieren, in der die GUI-Elemente beschrieben werden, stellt dies eine Vermischung der Zuständigkeiten dar. Für kleine Anwendungen wie Todo-Apps oder Prototypen ist dies zulässig, aber für größere Anwendungen ist es von großem Nachteil. In diesem Ansatz wird eine feste Menge an gängigen Controls verwendet. Da sich dies auch in anderen Ansätzen wiederfindet (z.B. [Achilleos2011]) sind diese exemplarisch neben anderen Elementen der DSL in Abbildung 1.4 dargestellt.

Listing 1.1: Eine Cross-Mobile-Programmierprache [Rosén2013]

```
listscreen NotesScreen() {
  List<Note> notes = Note.getAll();
  title(text = "All notes");
  dynamiclist noteList(data=notes, callback=showNote);
  void showNote(Note n) { goto DetailScreen(n); }
}
```

1.6.2 Nicht modellgetriebene Ansätze

Neben den modellgetriebene Ansätzen, die auf UML oder DSLs basieren, gibt es auch solche, die mit Mitteln des funktionalen und logischen Entwurfs Anwendungen modellieren und Eigenschaften mathematisch überprüfen können.

In [Clark2011] werden **Mobil-Anwendungen im λ-Kalkül** beschreiben. GUI und Logik werden mit Funktionen modelliert und zur Korrektheitsüberprüfung kann ein Inferenzsystem verwendet werden. Die Autoren haben als Problem herausgestellt, dass die betrachteten Cross-Mobile-Frameworks komplex sind und die Anwendungslogik nicht von der GUI-Implementierung trennen [Clark2011, S. 16]. Einige der untersuchten Frameworks sind ereignisgetrieben, benutzen aber eine dynamische Event-Handler-Auflösung zur Laufzeit. Die Autoren bieten als Lösung an, eine statische Überprüfung der Handler zur Übersetzungszeit vorzunehmen. Dazu soll eine DSL verwendet werden die typische Aufgaben für Mobilgeräte umfasst (ähnlich wie bei

[LeGoaer2013]). Die dort entwickelte DSL wird als Erweiterung des einfach getypten λ-Kalkül umgesetzt, was die Sprache mit Möglichkeiten zur Parametrisierung ausstattet. Der Ansatz basiert auf Monaden, um Zustandsänderungen darzustellen.
Die entwickelte DSL ist leider durch die Zustandsmonaden, Zustandsänderungen sowie Event-Handler-Registrierung sehr imperativ, was der Idee widerspricht, elegante kurze Anwendungen auf einer hohen Abstraktionsebene zu erstellen.

In einem ähnlichen Ansatz werden statt Funktionen **Regeln in Prolog zur GUI-Beschreibung** verwendet [Deridder2004]. Die Regeln werden in drei Kategorien eingeteilt, wodurch eine Trennung der Zuständigkeiten durch diese Ebenen möglich ist. Durch Koreferenz werden Daten ähnlich wie bei Datenbindung aus der GUI an verarbeitende Prädikate weitergereicht.

- **Interaktions-Regeln** reagieren auf die Eingaben des Nutzers. Beispielsweise kann die Eingabe der Postleitzahl das Nachschlagen des zugehörigen Stadtnamens veranlassen. Der Zusammenhang wird über ein Prädikat dargestellt: `plzZuStadtname(PLZ, Stadtname).`
- **Logik-Regeln** werden ohne Nutzeraktion auf Daten angewandt. Beispielsweise können in einer Adressbuch-Anwendung Personen, die bald Geburtstag haben farblich gekennzeichnet werden, beispielsweise durch `hatBaldGeburtstag(Person).`
- **Layout-Regeln** ordnen mehrere Controls an. Beispielsweise können mehrere Panels untereinander, in mehreren Tabs oder in einzelnen Fenstern angezeigt werden.

Der Ansatz arbeitet mit der Smalltalk Open Unification Language (SOUL), die durch Metaprogrammierungs-Mechanismen Prolog-Regeln auf Smalltalk-Anwendungen anwenden kann und Codegenerierung ermöglicht. Dadurch können über Prolog-Regeln Smalltalk-Anwendungen erstellt werden. Da Smalltalk nicht mehr wirtschaftlich relevant ist und nur eine geringe Zahl der momentan wichtigen Plattformen bedienen kann, ist dieser Ansatz nur für Prolog-Programmierer und konzeptuell interessant.

In [Hanus2009] werden **GUIs deklarativ mit Curry**, einer funktional-logischen Programmiersprache mit Haskell-Syntax, programmiert. Die GUI wird als algebraischer Datentyp dargestellt. Eine konkrete GUI ist ein Wert dieses Typs, der wie in Listing 1.2 mit Funktionen zusammengesetzt werden kann.
Diese GUIs lassen sich auf der Desktop-Plattform mit der Tool Command Language und dem GUI Toolkit (Tcl/Tk) ausführen oder für das Web in HTML übersetzen. Dazu muss lediglich eine Import-Anweisung geändert werden, um die Zielplattform zu bestimmen: `import UI2GUI` für Desktop und `import UI2HTML` für das Web.
Als Beispiel dafür wird in Listing 1.2 eine Counter-Anwendung programmiert, die in Abbildung 1.5 als Desktop- und HTML-Anwendung dargestellt ist.

Listing 1.2: Eine Counter-Anwendung mit GUI, programmiert in Curry [Hanus2009, S. 7]

```
counterUI =
  col [ label "A simple counter:",
        entry val "0",
        row [button incr "Increment", button reset "Reset", button exitUI "Stop"] ]
  where
    val free
    reset env = setValue val "0" env
    incr env  = do v <- getValue val env
                   setValue val (show (readInt v + 1)) env
```

Dieser Ansatz hat den Vorteil, dass GUIs programmiert werden, indem sie als Datentypen zusammengesetzt werden und später von einer Funktion in GUIs der Zielplattform transformiert werden können. Dieses Vorgehen ist in vielen DSL-basierten Ansätzen der Fall. Hinzu kommt, dass mit den in Curry verfügbaren Metaprogrammierungs-Mechanismen ähnlich wie in Haskell auch auf Funktionen als Daten-Objekte zugegriffen werden kann. Weiterhin ist Curry als statisch typisierte Programmiersprache gut für eine typsichere Modellierung geeignet; Typ-Prüfung für DSLs muss hingegen manuell implementiert werden.
Allerdings ist der IDE-Support von Curry sehr begrenzt und Curry besitzt wie Haskell eine geringe wirtschaftliche Relevanz. Daher ist der Ansatz konzeptuell interessant aber im wirtschaftlichen Kontext nicht zu finden.

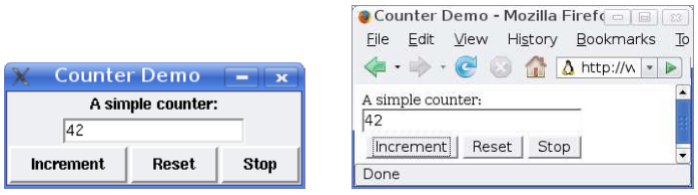


Abbildung 1.5: Ein in Curry programmierter Counter für Desktop und Web [Hanus2009, S. 8]

1.7 Aktuelle Cross-Platform-Werkzeuge

Neben den genannten wissenschaftlichen Arbeiten gibt es einen Wirtschaftszweig, der sich auf die Entwicklung von plattformunabhängigen Entwicklungswerkzeugen spezialisiert hat. Abgesehen von einigen kostenlosen Werkzeugen variiert der Preis dieser Werkzeuge sehr stark von 25$ pro Monat bis 1575$ pro Monat. In Tabelle 1.2 wird ein Teil dieser Werkzeuge mit ihren Eigenschaften aufgelistet. Vergleichende Untersuchungen verschiedener Werkzeuge finden sich auch in [Rosén2013, S. 3] und [LeGoaer2013, S. 2-4].

Werkzeug/Quelle	Preis pro Monat	Plattformen	Programmierung	einige Vor- und Nachteile
[Appmethod]	25$	Mobile + Desktop	C++	+ plattformunabhängig
[Qt]	(NC*: 0$) 25-174$	Mobile + Desktop	C++	+ pl.unabh., – nur C++
[AlphaAnywhere]	167$	Mobile + Desktop	HTML + JS	+ plattformunabhängig
[PhoneGap]	0$	Mobile	HTML + JS,	+ vollständig kostenlos
[SenchaGTX]	80$	Mobile	Java/GWT	+ objektorientiert
[Xamarin]	25-158$	Mobile	C#	+ objektorientiert
[Appcelerator]	39-259$	Mobile	HTML + JS	– kein Desktop
[Sencha]	321-1575$	Mobile	HTML + JS	– sehr teuer
[GTK]	0$	Desktop	diverse Wrapper	+ vollständig kostenlos + div. Programmiersprachen – keine Mobilgeräte

Tabelle 1.2: Cross-Platform-Werkzeuge, *NC: für nicht-kommerzielle Apps

1.8 Plattformunabhängige Ansätze der Spieleentwicklung

Ein populäres Werkzeug im Bereich der plattformunabhängigen Spieleentwicklung ist das Unreal Development Kit [UDK], das für freie Projekte (z.B. Spiele, Visualisierungen) kostenlos verwendet werden kann. Für kommerzielle Projekte hingegen muss 208$ pro Monat (2500$ pro Jahr) und ab einem Ertrag ab 50000$ ein 25-prozentiger Anteil an die Hersteller des UDK gezahlt werden.

Zu den kostenlosen Werkzeugen zählen Unity und HaXe. Unity kann mit JavaScript oder C# programmiert werden und ist auch in einer kostenpflichtigen Variante mit erweiterten Features verfügbar [Unity]. HaXe wird mit einer statisch getypten, JavaScript-ähnlichen Programmiersprache entwickelt [HaXe]. Beide Werkzeuge ermöglichen die Entwicklung von Spielen bzw. 3D-Anwendungen, die auf mehreren Plattformen laufen.

Es ist bemerkenswert, dass es im Bereich der Spieleentwicklung sehr populär geworden ist, für mehrere Plattformen zu entwickeln, wohingegen Anwendungen für Smartphones beispielsweise oft wegen fehlendem Look-and-Feel nativ programmiert werden.

2 Grundlagen

2.1 Das MVVM-Architekturmuster

Model-View-ViewModel (MVVM) ist ein modernes Architekturmuster, das aus dem verbreiteten Architekturmuster Model-View-Controller (MVC) für GUI-Anwendungen hervorgegangen ist. Abbildung 2.1 zeigt die namensgebenden Bestandteile des Musters mit Ihren Verbindungen [Brown2012, S. 880]:

- **Models:** Dies ist die Ebene der Anwendungslogik und der fachlichen Objekte. Hier werden die Aufgaben erledigt, für welche die Anwendung entwickelt wurde. Zu dieser Ebene gehören Model- und Service-Klassen.
- **Views:** Views sind Benutzeroberflächen der Anwendung. Konkret werden sie mit den Mitteln eines GUI-Frameworks umgesetzt. Meist können Teile mit einer Markup-Sprache beschrieben werden, wie in Abschnitt 3.1 gezeigt wird. Die View greift nur über ViewModels auf Models zu.
- **ViewModels (VMs):** ViewModels stellen die Schnittstelle zwischen der Ebene der Views und der Ebene der Models dar. Dazu werden die Daten der Modelle so aufbereit, dass sie in den Views angezeigt und bearbeitet werden können. Zudem wird hier Interaktionslogik wie Navigation implementiert. ViewModels verwenden keine Methoden der konkreten View-Klassen.

Das MVVM-Architekturmuster eignet sich zur Programmierung von GUI-Anwendungen, weil eine klare Trennung der beiden Zuständigkeiten der Programmierung der GUI und Anwendungslogik existiert. Die

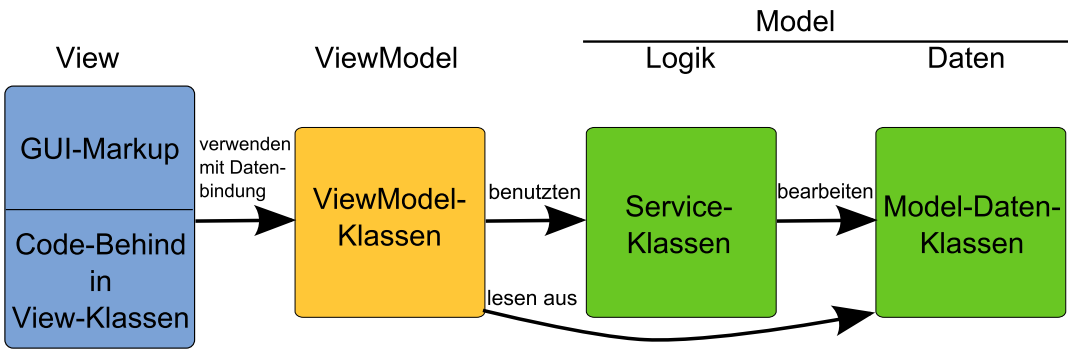


Abbildung 2.1: Die Ebenen Model, View und ViewModel des MVVM-Architekturmusters nach [Brown2012, S. 880]

ViewModel-Schicht hat die Aufgabe, Modell-Elemente anzuzeigen, zu verändern und für die Konsistenzhaltung der Daten zu sorgen.

2.2 Defintion „Plattform“

Bevor ich auf Plattformunabhängigkeit eingehe, möchte ich den Begriff der Plattform klären. Es finden sich folgende Definitionen:

> „Plattform: Moderne, uneinheitlich verwendete Bezeichnung für eine System- oder Software-Entwicklungsumgebung; nur aus dem Zusammenhang wird jeweils ersichtlich, ob die Hardware-Umgebung gemeint ist, das Betriebssystem oder ein Entwicklungswerkzeug: Programmiersprache, IDE usw.“ [Fischer2011, S. 679]

> „Die Plattform hat die Aufgabe, die Umsetzung der Domäne zu »stützen«. Das heißt, die Transformationen von formalen Modellen soll möglichst einfach sein.“ [Stahl2007, S. 32]

Da diese Definitionen in Bezug auf diese Arbeit nicht den Kern des Problems treffen, möchte ich eine eigene Definition vorschlagen. Das Problem besteht darin, dass eine Anwendung üblicherweise für eine Plattform (z.B. Web oder Android) entwickelt wird und nicht auf anderen Plattformen lauffähig ist.

> Eine Plattform ist eine Familie von Geräten, innerhalb derer es eine verbindende Eigenschaft bezüglich der Ausführbarkeit von Anwendungen gibt.

Da es eine Vielzahl an Geräten und heterogene Betriebssysteme gibt, ist es sinnvoll, diese zu Familien zusammenzufassen. Beispielsweise bezeichne ich mit Mobile-Plattform die Menge der Smartphone- und Tablet-Geräte. Die Android-Plattform umfasst solche, auf denen ein verhältnismäßig aktuelles Android-Betriebssystem läuft. Mit Desktop-Java-Plattform bezeichne ich Desktop-Geräte (PCs oder Laptops), auf denen eine verhältnismäßig aktuelle Java-Laufzeitumgebung läuft. Der Begriff „verhältnismäßig aktuell“ ist bewusst unscharf gehalten, weil nicht festgelegt ist, wie lange Anwendungen noch funktionieren, die für eine frühere Version eines Betriebssystems oder einer Laufzeitumgebung entwickelt wurden. Abwärtskompatibilität ist hier sehr wichtig, aber nicht immer gegeben. Eine für die Android-Plattform entwickelte Anwendung sollte auf neuen Android-Geräten laufen, aber auch auf solchen, die wenige Jahre alt sind. Analoges gilt für die Java- oder .NET-Laufzeitumgebung.

2.3 Defintion „Plattformunabhängigkeit“

Nun möchte ich den Begriff der Plattformunabhängigkeit für diese Arbeit präzisieren:

> Plattformunabhängigkeit ist die Möglichkeit, ein Programm auf mehreren Plattformen auszuführen.

Um ein Programm auf mehreren Plattformen auszuführen, gibt es mehrere Möglichkeiten:

- **Zwischencode und Virtual Machines:** Das Programm steht in einem Zwischencode bereit, der von einer Virtual Machine für die Plattform interpretiert oder bei Bedarf kompiliert wird, beispielsweise Java-Bytecode für die Java Virtual Machine oder Intermediate Language Code (IL Code) für die Common Language Runtime (CLR). GUIs können auf der Zielplattform ein plattformunabhängiges Aussehen besitzen oder das Look-and-Feel der Zielplattform annehmen (Java Swing).
- **Plattformunabhängiges Aussehen:** Das Programm benutzt ein GUI-Framework, das die Plattform-eigenen Routinen zum Zeichnen verwendet. Da die Controls gezeichnet werden, sehen Controls auf allen unterstützten Plattformen gleich aus (kein natives Look-and-Feel, Beispiel GTK# [GTKSharp]).

- **Cross-Compilation:** Bei der Kompilierung aus plattformunabhängige GUIs in GUIs der Zielplattformen erstellt. Für jede Plattform gibt es eine generierte Anwendung. Ein Beispiel dafür ist Xamarin, wobei ein Teil der Logik unabhängig benutzt werden kann [Xamarin].

Insbesondere wird unter „plattformunabhängig“ in den zitierten Arbeiten (Abschnitt 1.6) meist Cross-Mobile verstanden, d.h. GUIs können unabhängig vom Smartphone-Betriebssystem programmiert werden. Leider wird dort nicht der Schritt von Smartphone-unabhängig zu Plattform-unabhängig gemacht. Dies mag darin begründet sein, dass sich Desktop-Anwendungen und Anwendungen für Smartphones unterscheiden, dennoch ist es oft wünschenswert, dass man eine geschriebene Desktop-Anwendung für Smartphones oder Tablets portiert. Natürlich herrschen für Mobilgeräte andere GUI-Richtlinien als für Desktop-Anwendungen, die z.B. auf großen Bildschirmen (z.B. 50 cm Bildschirmdiagonale) angezeigt werden. Aber dennoch ist die Bildschirmbeschränkung kein unlösbares Problem: Beispielsweise können Elemente der GUI auf Mobilgeräten untereinander angezeigt, die auf Desktop-Rechnern nebeneinander stehen.

Die Entwicklung von Webseiten bietet das Konzept des Responsive Designs: Besonders gekennzeichnete Container werden abhängig von der Bildschirmgröße neben- oder untereinander angezeigt. Dieses Vorgehen ist auch für die Entwicklung von HTML-Anwendungen mit Bootstrap möglich, indem besondere CSS-Klassen (Selektoren) für Container zur Verfügung stehen. Bei GUI-Frameworks ist dieser Ansatz eher selten. Das im Browser standardmäßige Umbrechen bei vollen Zeilen kann in XAML mit dem sogenanntes WrapPanel nachbildet werden.

Bildschirmgrößen-abhängige Programmierung für iOS besteht darin, zwei GUIs zu entwickeln: eine für Tablets und eine für Smartphones. Für Android-Geräte ist dieses Vorgehen nicht praktikabel, weil eine höhere Diversität an Bildschirmgrößen existiert [AndroidUIPatterns] und daher eine feinere Unterscheidung vorgenommen werden sollte. Es können mehrere GUIs programmiert werden: sechs Unterteilungen der Pixeldichte oder vier Unterteilungen der Bildschirmgröße.

GUI-Frameworks, die die Technik **plattformunabhängiges Aussehen** verwenden, werden nicht immer eingesetzt, weil das native „Look-and-Feel“ fehlt. Anwendungen, die beispielsweise auf mehreren Mobil-Plattformen laufen, sollen auf dem Endgerät wie native Anwendungen aussehen und sich bedienen lassen. Dies ist bei Spielen nicht erforderlich, weil jedes Spiel einen anderen visuellen Stil haben darf. Dies umfasst auch GUIs in Spielen.

Da erstens natives Look-and-Feel wichtig genug ist, dass native Anwendungen wirtschaftliche Relevanz haben und zweitens die modell-getriebene Software-Entwicklung mit Generatoren für verschiedene Plattformen arbeitet, verwende ich die Methode **Cross-Compilation** für die hier entwickelte DSL.

2.4 DSLs und MDSD

Es findet sich beispielsweise folgende Erklärung für Domain Specific Languages (DSLs):

> „What’s a DSL?
> A DSL is a programming language that’s targeted at a specific problem; other programming languages that you use are more general purpose. It contains the syntax and semantics that model concepts at the same level of abstraction that the problem domain offers.“ [Ghosh2011, S. 10-11]

Kurz gesagt ist eine DSL eine Beschreibungssprache für ausgewählte Konzepte einer Domäne. Bei der Arbeit mit DSLs geht es darum, ein Modell der Domäne mittels einer textuellen Beschreibung zu erstellen.

Die modellgetriebene Softwareentwicklung (MDSD) stellt Modelle in den Vorderdgrund, im Gegensatz zu allgemeiner Softwareentwicklung, bei der der Quellcode das Hauptaugenmerk ist. Modelle können sowohl UML-Diagramme als auch andere Dokumente sein, die der Computer ohne aufwendige Aufbereitung verarbeiten kann, beispielsweise DSLs. Die verwendeten Tools für dieses Vorgehen sind unter anderem UML-Tools, die Entwicklungsumgebung (IDE) Eclipse, und das Modellierungs-Tool Enterprise Architect.

Im Rahmen dieser Arbeit ist das Modell ein DSL-Dokument, welches eine Anwendung beschriebt. Der Entwicklungsprozess orientiert sich daran, das Modell zu erstellen und zu verfeinern. Daraus wird dann die Anwendung erstellt. Eine umfangreiche Einleitung in DSLs in Bezug auf MDSD gibt [Stahl2007, S. 97ff.].

Es werden zwei Arten von Modellen unterschieden [Botturi2012]: Platform Independent Model (PIM) und Platform Specific Model (PSM). Das PIM ist die plattformunabhängige Beschreibung der Anwendung. Das PSM ist das Modell, das aus der Transformation des PIM für eine spezifische Plattform entsteht. Diese Transformation geschieht automatisiert. Das PSM wird in Quellcode der Zielplattform übersetzt. Modellgetriebene Softwareentwicklung besteht also grob gesehen aus diesen Schritten:

1. Die Erstellung einer plattformunabhängigen Beschreibung des Anwendung (als PIM).
2. Die Transformation der Beschreibung in plattformspezifische Modelle (in PSMs) für jede Zielplattform.
3. Die Erzeugung von Quelltext aus den spezifischen Modellen.

Eine Variation dieses Vorgehens ist, den Quelltext direkt aus der plattformunabhängigen Beschreibung zu generieren. Dies ist zum Beispiel dann möglich, wenn die plattformunabhängige Beschreibung bereits alle Informationen enthält, die für die plattformspezifische Generierung vonnöten sind.

Ein anderer Grund ist, dass sich die PSMs nicht adäquat in Modell-Objekte umsetzen lassen. Soll beispielsweise eine Java-Anwendung erzeugt werden, wäre es möglich, ein Code Document Object Model (Code DOM) zu erzeugen. Dieses lässt sich dann in Quelltext übersetzen. Dazu ist allerdings eine entsprechende Klassenhierarchie erforderlich, in die man sich einarbeiten muss. Zudem sind damit evtl. nicht alle Eigenschaften der Zielsprache darstellbar oder die Darstellung ist mit großem Aufwand verbunden.

Daher wähle ich hier den Weg, aus dem PIM direkt Quelltext zu generieren. Im Rahmen dieser Arbeit sind das HTML, CSS und JavaScript für Web-Anwendungen und Android-XML und Java für Android-Anwendungen.

2.5 Vorteile der Codegenerierung

Eine Schwierigkeit bei der Programmierung mit Frameworks besteht darin, dass diese nicht so verwendet werden, wie sie entworfen wurden. Dass alle Framework-Klassen richtig verwendet werden, ist aus mehreren Gründen schwierig sicherzustellen:

- Die Teile des Frameworks, die mit Vererbung arbeiten, müssen gut dokumentieren, welche Methoden zwingend angepasst werden müssen oder optional angepasst werden können.
- Bestimmte Schnittstellen von Klassen erfordern ein Protokoll, nach welchem die Methoden der Schnittstelle aufgerufen werden dürfen. Gute Dokumentation ist auch hier erforderlich.
- Ein Framework richtig zu verwenden, heißt auch unterliegende Muster richtig zu verwenden. Beispielsweise werden Windows-Forms-Programmierer nicht ohne Weiterbildung WPF korrekt verwenden, weil Windows Forms keine Datenbindung unterstützt, diese aber in WPF ein zentraler Bestandteil ist.

Nach [Stahl2007, S. 127] bietet MDSD hier den großen Vorteil, dass nur der generierte Code das Framework richtig verwenden muss; manuelle Implementierungen hingegen sind oft fehleranfällig. Dafür muss natürlich die Codegenerierung sehr gründlich geprüft werden, beispielsweise durch Unittests. Der Vorteil ist aber, dass auf Modell-Ebene keine Framework-Fehler passieren können. Damit sind Beschreibungssprachen von Framework-Details unberührt, denn der entsprechende Code wird generiert.

Ein weiterer Vorteil ist das Generieren von sogenanntem „Boilerplate-Code“, d.h. Code, der häufiger im Quelltext vorkommt und wiederkehrende Aufgaben löst. Dies kann z.B. dadurch begründet sein, dass die eingesetzte Programmiersprache diese Teile nicht abstrahieren kann.

3 Evaluation bestehender Beschreibungssprachen

In diesem Abschnitt werde ich bestehende gängige GUI-Beschreibungssprachen in XML im Hinblick auf diese Kriterien evaluieren: Plattformen, Validierung, Datenbindung, Eignung MVVM, GUI-Anpassbarkeit und Generator-Anpassung. Eine Zusammenfassung der Vor- und Nachteilen dieser Beschreibungssprachen schließt diese Evaluation.

3.1 Strukturierung: GUI und Logik

In aktuellen GUI-Frameworks für Java (Java-FX) oder .NET (WPF) und ebenso für Android besteht die GUI-Programmierung aus zwei Teilen: Einerseits die Beschreibung der GUI in einer Beschreibungssprache, andererseits das Programmieren von Interaktionslogik in einer sogenannten Code-Behind-Datei. GUI-Beschreibungssprachen sind meist XML-basiert und beschreiben deklarativ die Struktur der GUI als Objektgeflecht. In einer Code-Behind-Datei werden Callback-Methoden implementiert, sodass in der Beschreibungssprache referenzierte Methoden beispielsweise beim Klicken eines Buttons aufgerufen werden. In diesen Methoden wird das Verhalten der GUI-Anwendung programmiert.
Man sieht auch am erforderlichen Anteil des GUI-Anpassungs-Codes in der Code-Behind-Datei, welche Anpassungen in der Beschreibungssprache nicht vorgenommen werden können.

3.2 WPF: XAML und C#

WPF stellt eine umfangreiche Bibliothek zur Entwicklung von GUI-Anwendungen zur Verfügung und ist Bestandteil des .NET-Frameworks von Microsoft. In WPF werden GUIs mit der Beschreibungssprache XAML beschrieben. Die Logik wird in einer Code-Behind-Datei in C# programmiert.

XAML steht für E**x**tensible **A**pplication **M**arkup **L**anguage, ist eine XML-basierte Sprache und beschreibt das Objektgeflecht der GUI über hierarchisch-verschachtelte Knoten. In XAML gibt es einige Kurzschreibweisen, beispielsweise für Bindungen, die verwendet werden sollte. Der äquivalente C#-Quellcode wäre umfangreicher. Der Datenbindung-Mechanismus von WPF bindet Properties aneinander, beispielsweise den Namen eines Kunden-Objektes an den Inhalt einer TextBox.

Die in Listing 8.1 gezeigte Beschreibung stellt ein Fenster mit zwei untereinander angeordneten Buttons und zwei TextBox-Controls dar. Der relevante Teil ist in Listing 3.1 dargestellt, die GUI in Abbildung 3.1.

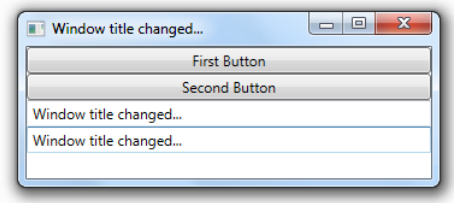


Abbildung 3.1: Ein Fenster mit zwei Buttons und TextBox-Controls in WPF

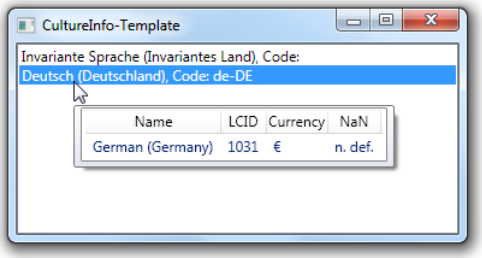


Abbildung 3.2: GUI für ein Template-Beispiel mit CultureInfo-Objekte in WPF

Beim Klick auf die Buttons erscheint ein kleines Fenster mit einer Meldung. Es existiert eine Datenbindung zwischen den TextBox-Controls und dem Titel des Fensters: Ändert der Benutzer den Text einer TextBox, wird der Text der anderen sowie der Fenstertitel ebenfalls geändert.

Listing 3.1: Ein einfaches Datenbindungsbeispiel in XAML

```
<TextBox Text="{Binding Title}" />
<TextBox Text="{Binding Title}" />
```

Dieses Beispiel ist bewusst nicht nach dem MVVM-Muster programmiert, da in einem überschaubaren Rahmen Datenbindung demonstriert werden soll. Zudem sind die zugrundeliegenden Daten in diesem Fall nur der Titel des Fensters.

Die Klasse `ExampleWindow` enthält die Programmlogik und ist in Listing 8.2 gezeigt. Dort wird das Datenbindungsziel (`DataContext`) festgelegt, damit die Bindungen in XAML wie oben formuliert werden können und sich auf ein Objekt beziehen. Auf diese Weise wird in XAML die Verbindung von View und ViewModel hergestellt. Der relevante Code-Ausschnitt ist in Listing 3.2 gezeigt.

Listing 3.2: Aufsetzen der Datenbindungen unter WPF in C#

```
public ExampleWindow() {
  InitializeComponent();
  DataContext = this;
}
```

In XAML lassen sich Templates definieren, welche Model-Objekte in GUI-Objekte übersetzen können.
Ein Beispiel für den Templating-Mechanismus wird in Listing 8.3 gezeigt, ein Ausschnitt davon in Listing 3.3 und in Abbildung 3.2 dargestellt. Als Datengrundlage dienen CultureInfo-Objekte, die angeben, in welcher Sprache das System läuft. Die aktuell im System eingestellte Sprache sowie eine Standardsprache werden in einer Liste angezeigt. Dabei beschreibt ein Template, wie Objekte dieses Typs dargestellt werden sollen. In dem Beispiel sollen Anzeigename und Code hintereinander stehen. Weitere Informationen erscheinen bei Mauszeiger-Berührung: die englische Bezeichnung, die Locale-Chart-ID (LCID), das Währungssymbol und eine Darstellung für „keine Zahl“ (NaN).

Listing 3.3: Ein Template für CultureInfo-Objekte in XAML

```
<ListBox.ItemTemplate>
```

```
  <DataTemplate DataType="sys:CultureInfo">
    ...
    <TextBlock Text="{Binding DisplayName}" />
    ...innerhalb des ToolTips dieses Elementes:...
    <GridViewColumn Header="Name" DisplayMemberBinding="{Binding EnglishName}" />
    <GridViewColumn Header="Currency"
      DisplayMemberBinding="{Binding NumberFormat.CurrencySymbol}" />
    ...
  </DataTemplate>
</ListBox.ItemTemplate>
```

Visuelle Anpassungen der GUI funktionieren über Styles. Styles können für Objekte eines bestimmten Control-Typs Eigenschaften festlegen, z.B. die Größe der Schriftart oder der Abstand zu anderen Controls. Sie können für verschiedene Gültigkeitsbereiche angegeben werden: die gesamte Anwendung, ein bestimmtes Fenster, einen Container oder nur ein bestimmtes Control.

WPF hat den Nachteil, dass es nur auf Windows mit .NET zur Verfügung steht. Weil es sehr stark an das Windows-Betriebssystem gekoppelt ist, steht es nicht für andere Plattformen zur Verfügung, z.B. Linux, OSX bzw. Android, iOS und auch nicht für Windows Phone.

3.3 JavaFX: FXML und Java

JavaFX stellt umfangreiche Möglichkeiten zur GUI-Programmierung bereit. Es ist der Nachfolger des Swing-GUI-Frameworks. Als Bestandteil der Java-Laufzeit von Oracle ist es auf Geräten verfügbar, auf denen die diese installiert ist. Die GUI-Beschreibungssprache ist ein XML-Format namens FXML; Java wird für die Code-Behind-Datei verwendet. Visuelle Anpassungen können im CSS-Format vorgenommen werden.

Als Beispiel wird ein Container beschrieben, der ein Eingabe-Textfeld und mehrere Ausgabefelder besitzt. Die komplette GUI-Beschreibung findet sich in Listing 8.4, der relevante Teil in Listing 3.4 und die Darstellung der GUI in Abbildung 3.3.

Listing 3.4: Textfelder für das Datenbindungsbeispiel in JavaFX, beschreibende Labels sind ausgelassen

```
<FlowPane ...> ...
  <TextField fx:id="test" /> ...
  <Label text="${test.text}" /> ...
  <Label fx:id="lengthOutput" />
</FlowPane>
```

Die Funktionalität des Programmes ist sehr einfach: Ein eingegebener Text wird mit seiner Länge wieder ausgegeben. Dies wird hier mittels Datenbindung realisiert, indem die Text-Property des Ausgabe-Labels an die Text-Property des Eingabefeldes gebunden wird. Diese Bindung lässt sich ähnlich wie in XAML in der Beschreibungssprache FXML ausdrücken.

Die Code-Behind-Datei ist in Listing 8.5 gezeigt, wobei der interessante Ausschnitt Listing 3.5 ist. Um die Länge auszugeben, wird eine Bindung dafür erzeugt: Die Methode `length()` ermittelt die Länge der einer String-Property, die dann an die Text-Property des entsprechenden Labels gebunden wird. Zuvor muss eine String-Konvertierung durchgeführt werden. Ändert sich das Text-Eingabefeld, wird die Text-Property im Controller und beide Labels aktualisiert. Umgekehrt ist eine Änderung der Property an der View sichtbar.

Listing 3.5: Aufsetzen der Datenbindung in JavaFX

```
public void initialize(URL location, ResourceBundle resources) {
  input.textProperty().bindBidirectional(text);
  lengthOutput.textProperty().bind(SimpleStringProperty.stringExpression(text.length()));
}
```

Neben Datenbindungen ist auch die visuelle Anpassung von GUI-Elementen möglich. Dazu wird ein CSS-Format verwendet und eine Reihe von Selektoren wie `.button` und Attribute wie `-fx-background-color` bereitgestellt. In einer CSS-Datei können dann Einträge dieser Form vorgenommen werden:

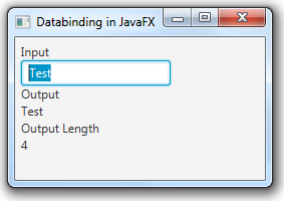


Abbildung 3.3: Ein Fenster mit Datenbindung in JavaFX

`Selektor { Attribut`$_1$`: Wert`$_1$`; ... Attribut`$_n$`: Wert`$_n$`; }`
Diese Einträge haben folgende Bedeutung: Für jedes Control, das zum `Selektor` passt, setze dessen Attribute `Attribut`$_k$ ($k = 1$ bis n) auf den zugehörigen Wert `Wert`$_k$.

Neben Selektoren für bestimmte Typen wie Button und Label, lassen sich auch eigene Selektoren definieren, sodass bestimmte in FXML spezifizierte Controls angepasst werden können [JavaFXCSS]. Im nächsten Beispiel werden die vordefinierten Selektoren verwendet, um den Hintergrund auf schwarz und die Textfarbe von Labels auf weiß setzen:

```
.root  { -fx-background-color: black; }
.label { -fx-text-fill: white; }
```

JavaFX stellt mit Datenbindung und einer umfangreichen Control-Bibliothek eine gute Grundlage für GUI-Programmierung dar, zumal Java auf vielen Geräten zur Verfügung steht. Allerdings werden JavaFX-Anwendungen nicht direkt von Mobilgeräten mit iOS, Android oder dem Windows Phone unterstützt. Es gibt Projekte für die Programmierung von Mobilgeräten mit JavaFX, eines davon wird in [JavaFXMobile] vorgestellt.

3.4 Android: Android-XML und Java

Für Android findet sich folgende Definition:

> „Android: Betriebssystem mit vollständiger Entwicklungsumgebung für Mobilgeräte basierend auf eine [sic] Linux Kernel; hinter dem zwar quelloffenen A[ndroid] steckt aber der Informationsgigant Google“ [Fischer2011, S. 40]

Android ist als vorherrschendes Betriebssystem für Smartphones eine wichtige Plattform für Entwickler. Die Programmierung mit Java und eine verhältnismnäßig einfache Installation von Anwendungen auf dem Gerät machen die Android-Plattform für Entwickler interessant (siehe dazu auch Abschnitt 1.4). GUIs können in Android mit einer XML-Beschreibungssprache entwickelt werden, die XAML ähnelt, allerdings keine Datenbindung unterstützt.

Die kleine Beispielanwendung hat die Aufgabe, bei einem Klick auf einen Button die aktuelle Uhrzeit anzuzeigen. Die GUI ist in Abbildung 3.4 dargestellt, der relevante Teil der GUI-Beschreibung in Listing 8.6 und der Code-Behind-Ausschnitt in Listing 3.7. Das vollständige Beispiel findet sich in Listing 8.7 bzw.Listing 3.6.

Listing 3.6: Ein Container mit einem Label und Button in Android-XML

```
<LinearLayout a:orientation="vertical" ...>
  <TextView ... a:text="Click the button to show current time!" />
  <Button a:onClick="updateTime" a:text="Click me!" ... />
</LinearLayout>
```

Die Programmlogik steht in der Klasse `StartActivity`, die per `R.layout.activity_start` auf die dargestellte XML-Beschreibung zugreift, um sie zu laden. Zunächst wird das Label-Control durch Angabe seiner ID

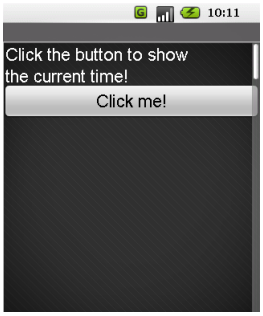


Abbildung 3.4: Ein Fenster mit Label und Button in Android

ermittelt und in einem Feld gespeichert. Es kann nun in der Methode `updateTime` verwendet werden: Der Text-Property des Labels wird die aktuelle Zeit als String zugewiesen.

Listing 3.7: Code-Behind-Datei einer Android-Anwendung

```
private TextView textview;
@Override
public void onCreate(Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  setContentView(R.layout.activity_start);
  textview = (TextView)findViewById(R.layout.textview);
}
public void updateTime(View view) {
  textview.setText(new Date().toString());
}
```

Die Programmierung von Android-Anwendungen ist im Vergleich zu WPF und JavaFX eingeschränkter, weil keine Datenbindung zur Verfügung steht. Stattdessen werden Callback-Methoden und anderer Boilerplate-Code programmiert werden muss (z.B. die nicht dargestellten Adapter-Klassen). Zudem gibt es einige Besonderheiten bei der Programmierung für Android, auf die in Abschnitt 5.5.5 eingegangen werden.

3.5 Web: HTML, CSS und JavaScript

Es findet sich für HTML folgende Definition:

> „HTML: Hypertext Markup Language; plattformunabhängige Seitenbeschreibungssprache für die Verknüpfung von Informationen zu einem Hypertext-Dokument und die visuelle Präsentation desselben; die in das Dokument eingebundenen Befehle heissen Tags und werden durch den Web-Browser interpretiert; siehe auch HTTP; H[TML] wurde am CERN in Genf entwickelt und zwar für Intranet-Aufgaben, lange bevor es dieses Schlagwort gab; Standard 4.0 des W3C seit Herbst 1997; HTML ist eine so genannte Anwendung von SGML – und gleichzeitig deren einzige erfolgreiche; siehe SELFHTML; vergleiche Webseite“ [Fischer2011, S. 404]

Da sich die Verarbeitung von XML seit der Entstehung von HTML immer mehr durchgesetzt hat, wird häufig die XML-Variante von HTML namens XHTML verwendet. HTML wird in Verbindung mit Cascading Style Sheets (CSS) zur visuellen Anpassung und JavaScript (JS) zur Programmierung der Interaktionslogik eingesetzt.

Da das Standard-Aussehen von HTML unbefriedigend ist, werden häufig externe GUI-Bibliotheken verwendet. Eine populäre GUI-Bibliothek ist Bootstrap, die neben verbessertem Aussehen auch weitere Controls, Menüs und Layoutcontainer bietet [Bootstrap]. Bootstrap wurde ursprünglich für Twitter entwickelt und kann frei verwendet werden. Dazu müssen die erforderlichen CSS- und JS-Dateien eingebunden werden, dazu gehört auch die JS-Bibliothek jQuery [JQuery].

Abbildung 3.5: Ein Counter als Web-Anwendung in JavaScript und HTML mit Bootstrap

Im Folgenden wird ein einfacher Counter mit HTML und JS programmiert, der sich über drei Buttons mit diesen Aktionen steuern lässt: Erhöhen, Verringern und Zurücksetzen. Die Knöpfe werden in HTML als Tags definiert und rufen JavaScript-Funktionen auf, welche den Counter manipulieren und Teile der HTML-Seite im Browser verändern. Der relevante HTML-Teil ist in Listing 3.8, der JS-Teil in Listing 3.9, der vollständige Quellcode findet sich in Listing 8.8. Die Darstellung der HTML-Seite findet sich in Abbildung 3.5.

Listing 3.8: HTML-Teil des HTML-Counters

```
<html>
  <head>...Titel, JS-Code...</head>
  <body onload="reset()"> ...
    <h2 id="counter"></h2>
    <div class="btn-group" role="group">
      <a class="btn btn-default" onclick="decr()">-1</a>
      <a class="btn btn-default" onclick="reset()">=0</a>
      <a class="btn btn-default" onclick="incr()">+1</a>
    </div> ...
  </body>
</html>
```

Die in den Buttons referenzierten Funktionen können in derselben HTML-Datei oder einer externen JS-Datei stehen. Für dieses Beispiel wurde nur eine Datei verwendet.

Listing 3.9: JavaScript-Teil des HTML-Counters

```
<script>
  var count = 0;
  function refresh() { $("#counter").html("Counter: " + count); }
  function decr() { count--; refresh(); }
  function reset() { count = 0; refresh(); }
  function incr() { count++; refresh(); }
</script>
```

Bereits bei diesem kleinen Beispiel wird ein Problem deutlich: Immer wenn sich der Zustand des Zählers ändert, muss die View aktualisiert werden. Diese Aufgabe übernimmt die Funktion `refresh`. Mit Datenbindung ließe sich das Problems eleganter lösen: Der Zustand des Zählers wird an ein Element der View gebunden und Änderungen werden propagiert. Dieses Vorgehen wird für die generierte Web-Anwendung gewählt, siehe dazu Abschnitt 5.4.1.

HTML hat den großen Vorteil, auf allen Plattformen zu laufen, auf denen ein Browser zur Verfügung steht. Auf Desktop-Rechnern ist die Verwendung von HTML-Anwendungen optimal. Jedoch besteht der Nachteil, dass die Benutzung von HTML-Anwendungen auf Smartphones nicht immer komfortabel ist.
Dies kann vielfältige Gründe haben:

- **Der Browser** ist unzureichend zu bedienen oder zu langsam.
- **Die Hardware** ist zu langsam für die Anwendung.
- **Die Anwendung** ist nicht für Smartphones ausgelegt oder schlecht zu bedienen.

Dazu kommt, dass HTML zwar plattformunabhängig ist, aber dennoch spezifische Unterschiede der Browser existieren. Diese Browserabhängigkeit von HTML ist auch in unterstützten Controls zu sehen, wie etwa dem Eingabe-Control für Zeitpunkte, das in Opera zur Verfügung steht, in Firefox jedoch nicht. Auch einige JavaScript-Funktionen stehen nicht in allen Browsern bereit.

Aus den oben genannten Gründen sind native Anwendungen für Smartphones weiterhin wichtig und können nicht durch HTML-Anwendungen abgelöst werden.

3.6 Konsens: Beschreibung von GUIs mit Beschreibungssprachen

Die vorhergehenden Beispielen illustrieren den allgemeinen Aufbau von GUI-Anwendungen: GUIs werden mit Beschreibungssprachen beschrieben werden, in Code-Behind-Dateien wird mittels einer generellen Programmiersprache die Interaktions- und Programmlogik definiert. Es aus mehreren Gründen vorteilhaft, wenn die Erzeugung der GUI und deren Anpassung über Beschreibungssprachen geschieht:

- **Trennung von Zuständigkeiten:** Die GUI-Erzeugung ist eine eigene Zuständigkeit und sollte nicht mit Interaktionslogik gemischt werden. Beschreibungssprachen sind genau dafür geacht, GUIs zu erstellen und anzupassen.
- **Vereinfachte Erzeugung:** Die Erzeugung des GUI-Objektgeflechtes über nicht-deklarativen Quellcode ist fehlerträchtig und wird von einer Beschreibungssprache vereinfacht. Diese beschreibt hierarchisch-strukturierte Elemente mit angepassten Attributen deklarativ.
- **Ausdrucksstärkerer Code:** Beschreibungssprachen sind oft kürzer und prägnanter als die Erzeugung im Quellcode über imperative Zuweisungen und Methodenaufrufe. Sie geben damit die Intention des Programmierers deutlicher wieder.
- **Vorschau-Möglichkeit:** Für viele Beschreibungssprachen gibt es in modernen IDEs eine Vorschau-Möglichkeit, sodass das Programm nicht erst kompiliert und gestartet werden muss, damit die GUI sichtbar ist.

3.7 Zusammenfassung

Im Folgenden stelle ich die Ergebnisse dieses Kapitels kurz tabellarisch dar und stelle die wichtigsten Unterschiede und Gemeinsamkeiten der Beschreibungssprachen dar.

Die in dieser Arbeit entwickelte Beschreibungssprache nenne ich „Plain“, was für **Pla**tform**in**dependent steht. Sie wird zur besseren Illustration in die zusammenfassende Tabelle 3.1 eingereiht. Die Tabelle zeigt die Bewertung der Frameworks hinsichtlich der Kriterien aus Abschnitt 1.4: Plattformen, Validierung, Datenbindung, GUI-Anpassbarkeit und Generator-Anpassung.

Kriterium / DSL	FXML	Android-XML	HTML	XAML	Plain-DSL
Plattformen	Java	Android	Web	Windows	Erweiterbar
Validierung	Nein (Bibl.)	Nein (GUI)	Nein (Bibl.)	Ja	Erweiterbar
Datenbindung	Ja	Nein (Bibl.)	Nein (Bibl.)	Ja	Ja
Generator-Anpassung	Nein (Parser)	Nein (Parser)	Nein (Parser)	Markup-Extensions	Erweiterbar
GUI-Anpassbarkeit	Ja (Theme)	Ja (Theme)	Ja (Theme)	Ja (Themes und Templates)	Ja

Tabelle 3.1: Evaluation bestehender Beischreibungssprachen

Es folgt eine Erklärung der Einträge der Tabelle:

- **Nein (Bibl.):** Nicht in der Beschreibungssprache vorgesehen, aber durch externe Programmbibliotheken möglich.
- **Nein (Parser):** Nicht in der Beschreibungssprache vorgesehen, aber durch Anpassung des Parsers möglich. So können Tags der Beschreibungssprache benutzerspezifische Daten enthalten, die zur Laufzeit interpretiert werden, um weitere Funktionalität bereitzustellen.
- **Nein (GUI):** Es ist nicht in der Beschreibungssprache vorgesehen, aber in der Code-Behind-Datei durch das GUI-Framework verwendbar.
- **Ja (Theme):** Durch einen Theme-Mechanismus können Controls visuell verändert werden.
- **Ja (Templates):** Durch einen Templating-Mechanismus können Modell-Objekte in andere Controls transformiert werden.
- **Markup-Extensions:** Markup-Extension sind Klassen, die einen Wert bereitstellen und an allen Stellen eingesetzt werden können, an denen Attributwerte benötigt werden.
- **Inline-Code:** Das Einfügen von Nicht-Beschreibungssprachen-Code (z.B. JavaScript) im GUI-Markup lässt beliebige Anpassungen zu.
- **Erweiterbar:** An einer geeigneten Stelle können manuell programmierte Anpassungen vorgenommen werden. In Bezug auf GUI-Anpassung wird dies in Abschnitt 5.6, für Generator-Anpassung in Abschnitt 5.7 erläutert.

In FXML, Android-XML und XAML können anstelle der plattformspezifischen Control-Klassen auch selbsterstellte Unterklassen dieser Klassen verwendet werden, sodass durch Vererbung ggf. weitere Anpassungen möglich sind. Voraussetzung ist, dass die Basisklasse für Unterklassenbildung konzipiert ist.

Die Punkte Markup-Extensions und Inline-Code möchte ich hier kurz ausführen: Mit Markup-Extensions lassen sich selbstprogrammierte Klassen in XAML-Attributen verwenden. Dadurch können beispielsweise Daten aus einer Datenquelle an Controls gebunden werden (Datenbindung) oder den Zugriff auf bestimmte Werte gewähren, die in XAML nicht ausgedrückt werden können.

In HTML lassen sich an einigen Tags Callbacks mit JavaScript-Code angeben. Beispielsweise kann man damit das Verhalten eines Buttons so anpassen, dass sich die Farbe bei Mausberührung ändert. Angenommen, entsprechende Farb-Änderungs-Funktionen sind implementiert, dann können diese als Callbacks folgendermaßen angegeben werden:

```
<button id="coloredButton" onMouseOver="changeColor()" onMouseOut="revertColor()" />
```

Auf diese Weise können zwar Verhaltensanpassungen vorgenommen werden, die hier nur beispielhaft visuell sind, dennoch sollte auf Inline-Code verzichtet werden. Denn das Erstellen von weiteren Buttons dieser Art führt zu Code-Duplizierung und sollte daher mit anderen Mitteln umgesetzt werden. Inline-Code führt auch zu einer Vermischung von Zuständigkeiten, weil Probleme in der Beschreibungsebene gelöst werden.

XAML unterstützt zwar die in der Tabelle genannten Kriterien weitestgehend, ist aber jedoch nur für Windows-Plattformen verfügbar. Anderen Frameworks hingegen fehlt beispielsweise Validierung, die dann selbst oder über externe Bibliotheken programmiert werden müssen. Allen genannten Frameworks ist gemein, dass sie nur für eine begrenzte Zahl an Plattformen verwendet werden können, wobei JavaFX durch die Java-Laufzeit auf vielen Plattformen verfügbar ist, abgesehen von Mobilgeräten. HTML kann wie in Abschnitt 3.5 erläutert, nicht uneingeschränkt für Mobilgeräte verwendet werden.

Im Folgenden werde ich die oben aufgelistete Plain-DSL entwerfen und implementieren. Mit dieser DSL, unter Verwendung der dazugehörigen Generatoren, soll die plattformunabhängige Entwicklung von Anwendungen untersucht werden.

4 Design einer plattformunabhängigen DSL für GUIs

Beim Design einer plattformunabhängigen DSL für GUIs sind viele Dinge zu beachten. Im Folgenden werden Designentscheidungen für die DSL und die Generator-Infrastruktur diskutiert. Dazu wird das zugrundeliegende Architekturmuster und das allgemeine Zusammenspiel von Programmiersprachen, Frameworks und Generatoren geklärt.

An einem Beispiel wird zunächst illustriert, warum Architekturmuster sinnvoll sind und welche Probleme sich ergeben, wenn ohne Trennung von Zuständigkeiten programmiert wird.

4.1 Architektur der Zielanwendung: MVVM

Man stelle sich vor, ein Entwickler verwendet ein ihm unbekanntes GUI-Framework das erste Mal. Er entwickelt damit im Folgenden eine Anwendung schrittweise:

Zuerst erstellt er ein Hallo-Welt-Programm, um zu sehen, ob seine Entwicklungswerkzeuge richtig konfiguriert sind. Wenn das Programm kompiliert, fügt der Programmierer ein Listen-Control hinzu, um die Controls näher kennenzulernen. Er verwendet dieses Control dazu, Strings anzuzeigen. Dann fügt er die Möglichkeit hinzu, neue Einträge in die Liste einzufügen und zu löschen.

Damit ist bereits ein Prototyp einer Todo-Listen-Anwendung entstanden. Der Programmierer fügt noch Features wie Persistenz in Dateien und ein In- und Export für ein Kalenderformat hinzu.

Nehmen wir an, dass all diese Schritte so verlaufen, dass die Hauptfenster-Klasse um immer mehr Daten und Methoden erweitert wird. Das entstandene Programm besteht dann nur aus der Hauptfensterklasse, die mittlerweile mehrere hundert Zeilen Quelltext groß und damit unübersichtlich, fehlerträchtig und schwer zu erweitern und zu ändern ist. Eine solche Anwendung kann bis zu einem gewissen Punkt noch mit Refactorings strukturiert werden. Aber ab einer gewissen Größe ist es sinnvoller, die Anwendung neu zu entwickeln und gleich eine Trennung der Zuständigkeiten einzuhalten.

Dieses Beispiel verdeutlicht die Wichtigkeit der Trennung verschiedener Zuständigkeiten. Das in Abschnitt 2.1 vorgestellte Architekturmuster MVVM ermöglicht eine solche Aufteilung bei der GUI-Programmierung. Die Anwendung zur Aufgabenverwaltung wird mit dem MVVM-Architekturmuster wie folgt entwickelt.

1. Definiere Modelle und Services:
   (a) Definiere eine Klasse `Todo` mit einigen Properties wie `Titel`, `Notiz`, `Fälligkeitsdatum` und `Status`.

(b) Definiere eine Model-Klasse `TodoListe`, die die Daten der Anwendung enthält, d.h. eine Liste von Todo-Objekten.

(c) Definiere eine Service-Klasse: `TodoPersistenz` mit Operationen `Speichern` und `Laden`.

2. Definiere das ViewModel für `TodoListe`. Das ViewModel arbeitet auf der Model-Klasse und verwendet den Service. Im ViewModel werden Methoden für Aktionen implementiert, die man über die View ausführen kann, beispielsweise `ÖffneBearbeitenFenster` oder `Speichern`.

3. Modelliere die View mit einem GUI-Framework (ggf. unter Verwendung einer Beschreibungssprache). Diese View besitzt ein ViewModel-Objekt und ruft Operationen des ViewModels auf, wenn die Anwendung gestartet und beendet wird oder Buttons gedrückt wurden.

4. Je nach GUI-Framework müssen Modell-Objekte transformiert werden, um in einer Liste angezeigt zu werden. Steht ein Datenbindungsmechanismus zur Verfügung, können Daten an die Controls gebunden werden, ansonsten muss dies manuell implementiert werden.

Zur Programmierung einer gesamten Anwendung sind über die drei Bestandteile Model, ViewModel und View hinaus noch andere Aspekte zu berücksichtigen, die im MVVM-Architekturmuster nicht geklärt sind: Beispielsweise globale visuelle Anpassungen von GUIs (Themes), die Entwicklung von Views und deren Verbindung mit den ViewModels. Die Details dieser Aspekte werden für die Zielplattform Web in Abschnitt 5.4 und für Android in Abschnitt 5.5 diskutiert.

4.2 Einsatz von Programmiersprachen

Frameworks sind selten an eine bestimmte Programmiersprache gebunden, wodurch man zwischen mehreren Programmiersprachen wählen kann. In einigen Fällen können auch mehrere Beschreibungssprachen für GUIs verwendet werden, beispielsweise HTML-Template-Sprachen in Ruby on Rails [Rails]. Es ist üblich, zur Programmierung einer Anwendung mehrere dieser Sprachen einzusetzen. Man nennt dies auch polyglotte (vielsprachige) Programmierung.

Ein Softwareentwickler setzt eine bestimmte Programmiersprache aus mehreren Gründen ein:

- **Beherrschung der Programmiersprache:** Der Softwareentwickler hat mit der Programmiersprache Erfahrung. Er kann damit nicht nur Stoppuhr-Programme oder kleine Spiele erstellen, sondern er traut sich zu, auch größere Anwendungen wie Finanzverwaltungs- oder Textverarbeitungsprogramme zu erstellen.
- **Beherrschung des Paradigmas der Programmiersprache:** Ein Softwareentwickler beherrscht ein Paradigma zu einem gewissen Grad. Es ist sinnvoll, eine Programmiersprache erst dann für größere Anwendungen zu verwenden, wenn er das ihr zugrundeliegende Paradigma beherrscht. Dazu gehören beispielsweise beim objektorientierten Paradigma Entwurfsmuster und beim funktionalem Paradigma Funktionen höherer Ordnung.
- **Problemangemessenheit der Programmiersprache:** Eine Programmiersprache ist angemessen, wenn sie aktiv weiterentwickelt wird, hilfreiche Toolunterstützung existiert und der Entwurf innerhalb des Paradigmas begünstigt wird.
- **Frameworks und Kontext:** Wenn sich existierende Frameworks für das Vorhaben sehr gut eignen, kann auch dies ausschlaggebend sein. Dann sollten auch für das Framework verwendbare Programmiersprachen verwendet werden. Das Zusammenspiel von Programmiersprachen, Frameworks und Beschreibungssprachen ist zu beachten, wenn mehrere Frameworks verwendet werden.
- **Vorgabe innerhalb eines Projektes:** Neben den Überlegungen über Angemessenheit müssen auch eventuelle Projektvorgaben berücksichtigt werden.

Ähnliche Gründe gelten auch für Beschreibungssprachen, wobei natürlich beachtet werden muss, dass diese nicht das primäre Werkzeug, sondern eine sinnvolle Ergänzung zu einer generellen Programmiersprache sind. Denn Beschreibungssprachen adressieren Probleme einer bestimmten Domäne (z.B. GUI, Geschäftslogik, Datenbank) und werden verwendet, wenn sie einen Mehrwert in Bezug auf die generelle Programmiersprache bieten. Beispielsweise ist der Mehrwert bei der GUI-Entwicklung eine vereinfachte Beschreibung der GUI (siehe auch Abschnitt 3.6).

Zusammengefasst werden Programmiersprachen und Frameworks verwendet, wenn sie die Entwicklung vereinfachen. Sie sind prinzipiell austauschbar, es sei denn, Rahmenbedingungen oder fehlende Programmiererfahrung erzwingen bestimmte Kombinationen.

Was in den betrachteten Arbeiten aus Abschnitt 1.6.1 fehlt, ist diese Austauschbarkeit: Dort sind die DSL und die dafür entwickelten Generatoren untrennbar miteinander verwoben. Dies führt leider dazu, dass die vielversprechenden Ansätze und der in diese Arbeiten eingeflossene Programmieraufwand nicht für andere DSLs wiederverwendet werden können.

4.3 Überlegungen zur Plain-DSL

Aus den erwähnten Gründen im vorherigen Abschnitt ist bei der Entwicklung einer neuen Sprache darauf zu achten, dass die Funktionalität auch in anderen Kontexten wiederverwendet werden kann. Im Rahmen dieser Arbeit wird die Plain-DSL dazu verwendet, eine plattformunabhängige Beschreibungssprache darzustellen. Die Funktionalität ist hingegen nicht von der Plain-DSL abhängig.

Damit eine andere Beschreibungssprache als die Plain-DSL für die Generatoren verwendet werden kann, wird eine Indirektionsschicht verwendet, die in Abschnitt 5.8 beschrieben wird. Dazu wird ein Interface verwendet, das die erforderlichen Informationen aus der Beschreibungssprache extrahiert. So können auch andere Editoren (z.B. mit Xtext erstellte) und Modellierungsmittel (z.B. UML) verwendet werden, welche für die Generatoren plattformunabhängige Beschreibungen erstellen. In dieser Masterabeit werden die Generatoren beispielsweise auch mit der im Abschnitt 1.3 genannten gear-DSL verwendet.

Angenommen C# soll für die Definition von Models und ViewModels und XAML für Views verwendet werden. Dazu müsste das oben genannte Interface so implementiert werden, dass aus den entsprechenden Dateien die Informationen extrahiert werden. Dies ist möglich, weil in C# ebenfalls Properties existieren und Elemente der View mit konkreten Control-Klassen in XAML simuliert werden können. Datenbindung steht hier ebenfalls zur Verfügung. Dies darf nicht damit verwechselt werden, die Anwendung mit C# und XAML programmieren zu können. Es sollen nur dieselben Informationen modelliert werden. Trotzdem kann das Modell in C# und XAML ein ausführbares Programm ergeben.

Als Ausgangspunkt für die Plain-DSL wird den namensgebenden Bestandteilen von MVVM jeweils ein DSL-Element zugeordnet: `model`, `view` und `viewmodel`. Für die Modellierung der View diente auch die im Abschnitt 1.3 genannte gear-DSL als Inspiration, insbesondere die Elemente `edit` und `table` betreffend.

Der Aufbau einer MVVM-Anwendung wird in der Plain-DSL folgendermaßen dargestellt: Die Anwendung wird unter dem Knoten `app` gruppiert, welcher neben den Bestandteilen `model`, `view` und `viewmodel` ein für spezifische Anpassungen vorgesehenes Element `transform` enthält. Anstelle der plattformspezifischen Control-Klassen werden folgende abstrakte Elemente verwendet, die ich im Folgenden View Items nenne:

- `edit` zum Bearbeiten nicht listenwertigen Properties.
- `table` zum Bearbeiten von listenwertigen Properties.
- `panel` zum Gruppieren von View Items.
- `action` zum Auslösen einer Aktion, d.h. einer Operation im ViewModel.

Der genaue Aufbau und die Verwendung dieser Elemente wird im nächsten Abschnitt diskutiert und an Beispielen dargestellt.

4.4 Fallbeispiele

Im Folgenden sollen drei Beispiel-Anwendungen mit steigender Komplexität unter Verwendung der Plain-DSL modelliert werden. So wird deutlich, welche Schritte bei der GUI-Programmierung üblicherweise unternommen werden müssen und wie eine DSL dies vereinfachen kann. Zudem werden diese Beispiele für die Evaluation der DSL und Generatoren in Abschnitt 6 verwendet.

- **Die Todo App** stellt eine Anwendung dar, um zu erledigende Aufgaben (Todos) zu organisieren. Dazu wird eine Liste mit Todos zum Bearbeiten angezeigt. Die Aktionen beschränken sich auf das Einfügen, Verändern und Löschen von Einträgen. Ein Eintrag selbst besteht aus einem kurzen Titel und einer längeren Beschreibung.
- **Die Blog App (Abschnitt 4.4.2)** ermöglicht es, textbasierte Artikel zu schreiben. Diese Artikel sind einer Kategorie und einem Autor zugeordnet. Die Anwendung ermöglicht die Verwaltung der drei Model-Klassen Artikel, Kategorie und Autor über Tabellen mit den Aktionen Einfügen, Bearbeiten und Löschen. Die Artikel können zusammen als HTML-Dokument exportiert werden, das veröffentlicht werden kann.
- **Die Car Share App (Abschnitt 4.4.3)** organisiert den Verleih von Autos. Dazu kann der Ausleih-Status der Autos im System eingesehen und verändert werden. Der Benutzer kann sich einloggen und eine Ausleihe initiieren, indem er einen Zeitraum und ein ausleihbares Auto wählt. Nur Administratoren haben Zugriff auf alle Benutzer und Autos. Hier nicht dargestellt, aber vorstellbar ist, dass die Anwendung die Daten aus einer Datenbankquelle bezieht. Die Daten können in einem bestimmten Zeitintervall von der Quelle aktualisiert werden. Veränderte Daten werden zurückgeschrieben, sodass diese Anwendung mehrbenutzerfähig ist.

4.4.1 Todo App

Die Todo-Anwendung sehr kompakt: Es gibt nur eine View, die eine Tabelle und mehrere Eingabefelder enthält, mit denen Todo-Einträge bearbeitet werden können. Zudem kann ein Titel für die Todo-Liste angegeben werden. Der zu speichernde Zustand der Anwendung besteht aus diesem Titel und der Todo-Liste. In Listing 8.9 ist die komplette Anwendung, in Listing 4.1 nur ein Überblick über die Top-Level-Elemente dargestellt. Screenshots der Referenzimplementierung der Todo-Anwendung sowie die generierte Version finden sich in Abschnitt 6.1.

Listing 4.1: Grober Aufbau der Todo-App in der Plain-DSL

```
app TodoApp {
  model Todo { ... }
  model Todos { ... }
  viewmodel TodosVM uses Todos { ... }
  view TodoList { ... }
}
```

An diesem Beispiel wird die Syntax der entwickelten DSL erklärt: Das Wurzelelement `app TodoApp` enthält alle Bestandteile dieser Anwendung. Die möglichen Kindelemente von `app` sind `model`, `viewmodel`, `view` und `transform`, die Reihenfolge ist dabei beliebig.

Existieren im Editor mehrere Plain-DSL-Dokumente, werden diese nach App-Knoten durchsucht und alle gleichnamigen Knoten zusammengefasst. So könnten beispielsweise Model-Elemente in einer anderen Datei stehen als View-Elemente. Vor allem für Lokalisierungen und Styles über `transform`-Elemente sind eigene Dateien sinnvoll.

Ein Model-Element stellt ein fachliches Objekt der Anwendung dar: Es enthält alle Daten, um das Objekt im Kontext der Anwendung zu beschreiben. Für diese Anwendung hat ein Todo-Eintrag einen Namen und eine Notiz (Listing 4.2), weitere Properties wie `done : boolean` oder `deadline : Date` wären denkbar.

Listing 4.2: Model-Element Todo der Todo-App in der Plain-DSL

```
model Todo {
  prop name : String = "Important thing"
  prop note : Text = "Don't forget..."
}
model Todos [state] {
  prop title : String = "My Todo List"
  prop todos : List<Todo>
}
```

Der Typ `Text` wird auf der Zielplattform nach String gemappt. Aber bei der Generierung für ein `edit`-Element, das an eine `Text`-Property gebunden ist, wird ein mehrzeiliges Eingabefelder erzeugt. So lässt sich die Information „Diese Eigenschaft erlaubt mehrzeiligen Text.“ über einen Typen formulieren. Hier nicht dargestellt sind die Typen `Email` und `Phone`, die einen ähnlichen Zweck verfolgen: Sie werden in String umgesetzt und beeinflussen das Control für ihre Bearbeitung. Die Idee, bei der Modellierung einige andere primitive Datentypen benutzen zu können, ist von [Groenewegen2013] inspiriert. Dort sind Typen der Web-Domäne wie URL, WikiText und Email verfügbar.

Für primitive Properties können Standardwerte angegeben werden, die beim Start der Anwendung angezeigt werden. Ebenso werden Listen stets mit einem Element ausgestattet, um eine sinnvolle Bearbeitungsgrundlage zu bieten.

Das Thema Persistenz ist im Rahmen dieser Arbeit vereinfacht gelöst: Es existiert ein Model-Element, das den persistierbaren Zustand der Anwendung darstellt. Im Falle der TodoApp ist dies ein Titel der Liste und die Todo-Liste selbst, die im Model `Todos` gespeichert sind. Dieses Model-Element wird mit dem `state`-Attribut gekennzeichnet.

Ein ViewModel bezieht sich auf ein Modell-Element (dargestellt mit `uses`), aus dem es seine Daten bezieht. ViewModels erweitert Models um weitere Eigenschaften, die nicht persistiert werden und nur für die Bearbeitung oder Anzeige durch die View benötigt werden, z.B. temporäre Werte. Das ViewModel der Todo-App ist in Listing 4.3 gezeigt.

Neben weiteren Daten besitzt ein ViewModel Operationen, die von der View aufgerufen werden können. Eine solche Operation kann die Daten des ViewModels (und seines Models) verändern; diese Änderungen sind an der View sichtbar.

Für das Speichern des Zustandes sowie zur Navigation existiert eine Kurzschreibweise: Eine Operation mit dem Namen `saveState` ist so implementiert, dass sie den Zustand lokal speichert. Um zu einer anderen View zu navigieren, wird die Operation `showXY`, benannt, wobei `XY` der Name der View ist, zu der navigiert werden soll.

Listing 4.3: Das ViewModel der Todo-App in der Plain-DSL

```
viewmodel TodosVM uses Todos {
  prop todo : Todo
  op saveState
}
```

Eine View bezieht sich auf ein ViewModel (wieder dargestellt mit `uses`), aus dem es seine Daten bezieht. In `edit`-, `table`- oder `action`-Elementen beziehen sich Datenbindungen auf Properties bzw. Operationen dieses referenzierten ViewModels. Die Start-View wird mit dem Attribut `start` gekennzeichnet. Ein Beispiel zeigt Listing 4.4.

Listing 4.4: Die einzige View der Todo-App in der Plain-DSL

```
view TodoList uses TodosVM [start] {
  edit "Title for your list": [title]
  panel "Selected Todo" {
    edit [todo.name]
    edit [todo.note]
  }
  table [todos]
```

```
  action [saveState]
}
```

Wurde ein Label nicht angegeben, versucht der Generator eines aus der Datenbindung zu erstellen, was in vielen Fällen ausreicht. Dabei wird der letzte Teil des Property-Pfades als Grundlage verwendet, z.B. `name` bei `todo.name` und dieser in Großschrift angezeigt: „Name“. Bei Bezeichnern in Camel Case wie etwa `saveState` wird zuvor eine Zerteilung an einem Übergang von Klein- zu Großbuchstaben vorgenommen: „Save State“. Für den etwas selteneren Fall, dass kein Label verwendet werden soll, muss der leere String `""` als Label verwendet werden.

Das View Item `table` ist für die Bearbeitung von Listen gedacht. Dafür existieren Knöpfe zum Hinzufügen oder Löschen von Einträgen und ein Knopf, der ein ausgewähltes Element selektiert, sodass es in derselben oder einer anderen View bearbeitet werden kann. Die Property, die auf das selektierte Element gesetzt werden soll, wird über den Singular der Tabellen-Bindung ermittelt.

Diese Konvention findet sich auch in der Bearbeitung von Properties wieder, deren Typ ein Model-Typ ist. Der Eintrag `edit [todo] [source=todos]` wird in eine Combo-Box umgesetzt werden, welches das gewählte Element der Property `todo` anzeigt und bei Änderung zuweist. Als mögliche Elemente werden die Elemente verwendet, die im Attribut `source` angegeben werden, hier `todos`. Um also eine Property eines Model-Typen zu bearbeiten, wird ein existierendes Objekt ausgewählt. Ist kein Attribut `source` angegeben, wird der Plural des Typnamens der Property verwendet.

4.4.2 Blog App

Die Blog-App unterscheidet sich bezüglich der Komplexität von der Todo-App darin, dass mehrere Views sowie sich untereinander referenzierende Model-Elemente existieren. Sie ist in Listing 8.10 dargestellt. Screenshots der Referenzimplementierung der Blog-Anwendung sowie die generierte Version finden sich in Abschnitt 6.2.

Die Bearbeitung von Model-Elementen geschiet auf einzelnen Seiten. Beispielsweise lassen sich in der View `Article` (Listing 4.5) alle Eigenschaften des ausgewählten Article-Objektes bearbeiten, das über die Property `article` verfügbar ist.

Listing 4.5: Die View der Blog-App zur Bearbeitung eines Article-Objektes in der Plain-DSL

```
view Article uses BlogVM {
  action "Back to Articles" : [showArticles]
  edit [article.title]
  edit [article.content]
  edit [article.author]
  edit [article.category]
  edit [article.published]
}
```

Auf ähnliche Weise lassen sich Autoren und Kategorien bearbeiten. Damit die modellierte Anwendung nicht nur Daten bearbeiten kann, sondern auch Logik enthält, kann eine Implementierung der ViewModel-Operationen angeben werden. So kann prototypische Entwicklung mit der Plain-DSL betrieben werden.

Zur besseren Illustration ist hier die Implementierung einer Operation `generate` vorhanden, die alle veröffentlichten Artikel in eine HTML-Darstellung umwandelt (Listing 4.6). Das resultierende HTML-Fragment ließe sich beispielsweise auf einer Webseite veröffentlichen. Die Implementierung erfolgt in einer Untermenge der Programmiersprache Xtend. Hier wird die Konventionen eingeführt, dass temporäre Variablen mit einem Dollar (`$`) beginnen und externe Funktionen mit einem Unterstrich (`_`). So kann besser getrennt werden, welche Daten aus dem ViewModel stammen, welche Daten temporär sind und welche Operationen extern implementiert sind.

Listing 4.6: Implementation der ViewModel-Operation `generate` der Blog-App in der Plain-DSL

```
prop preview : Text = "Not generated"
```

```
op generate = {
  preview = "<h1>" + title + "</h1>"
  for($a : articles)
    if($a.published)
      preview = preview + "<h2>" + $a.title + "</h2>" + $a.content
  _alert("Done!")
}
```

4.4.3 CarShare App

Die Car-Share-App unterscheidet sich die Komplexität betreffend von der Blog-App darin, dass mehrere ViewModels und eine Navigationsfluss existieren. Die verschiedenen ViewModels stellen die Perspektiven (Benutzer oder Administrator) dar. Ein Navigationsfluss kann dann weitergeführt werden, wenn bestimmte Eigenschaften erfüllt sind. Die gesamte Anwendung ist in Listing 8.11 dargestellt. Screenshots der Referenzimplementierung der CarShare-Anwendung sowie die generierte Version finden sich in Abschnitt 6.3.

Alle Daten und Operationen, die mit dem Login und Ausleihen zusammenhängen, sind im ViewModel `LoginAndUserPerspective` gespeichert, welches von der Login-View und den beiden Rent-Views benutzt wird. Die Login-View kann erst dann verlassen werden, wenn man als Benutzer oder Administrator angemeldet ist.

Zur Umsetzung des Navigationsflusses sind die Operationen des ViewModels so implementiert, dass boolsche Properties angeben, ob die Voraussetzung zur Navigation erfüllt sind. Durch Bindung dieser Properties an das Attribut `enabled` in der View wird angegeben, ob ein Button zur Navigation klickbar ist. Auf ähnliche Weise wird über das Attribut `visible` angegeben, ob ein View Item sichtbar ist. Weitere Informationen zur Ausleihe werden beispielsweise nur angezeigt, wenn das selektierte Auto ausleihbar ist.

Der letzte Schritt der Ausleihe sollte nur dann ausführbar sein, wenn der Benutzer angibt, dass der Benutzer mit allen Bedingungen einverstanden ist. Dies findet sich häufig in Web-Anwendungen, erfordert aber in vielen Web-Frameworks mehr Aufwand als die hier dargestellten drei Zeilen: In der View verwenden zwei Controls eine Property des Viewmodels, wie in Listing 4.7 gezeigt.

Listing 4.7: Umsetzung von „Ich habe die Bedingungen verstanden“ im CarShare-Beispiel

```
...im ViewModel: prop agreed : boolean...
edit "I have read and understood the agreement" : [agreed]
action "Confirm" : [confirm] [enabled = agreed]
```

Wie auch beim Blog sind die Operationen mit einer Implementierung versehen. Dazu zählt die Login-Operation, die das Passwort überprüft, die Ausleihe eines Autos und das Zurücksetzen des Passwortes eines Nutzers in der Administrator-Perspektive.

Der Login ist so implementiert, dass in der User-Liste derjenige gesucht wird, dessen Benutzername und Passwort eingegeben wurde (Listing 4.8).

Listing 4.8: Login-Funktionalität der Car-Share-App in der Plain-DSL

```
prop loggedInUser : User
prop userName : String = ""
prop userPassword : Password = ""
prop loggedInAsUser : boolean = false
prop loggedInAsAdmin : boolean = false
op login = {
  loggedInAsUser = false
  loggedInAsAdmin = false
  for($u : users)
    if($u.name == userName && $u.password == userPassword) {
      loggedInUser = $u
      loggedInAsAdmin = loggedInUser.isAdmin
      loggedInAsUser = true
   }
}
```

Die Ausleihe eines Autos geschieht, indem eine E-Mail mit den erforderlichen Informationen an den Administrator der Anwendung geschickt wird. Gleichzeitig soll eine E-Mail an den Ausleihenden versendet werden. Danach wird eine Erfolgsmeldung angezeigt und zur Login-View navigiert.

Listing 4.9: Ausleih-Funktionalität der Car-Share-App in der Plain-DSL

```
prop car : Car
prop startRent : Date
prop endRent : Date
op confirm = {
  var $m = loggedInUser.email
  if(_validMail($m)) {
    var $text = car.name + ": " + startRent + " - " + endRent
    _sendMail("admin@carshare.com", "Request", $text)
    _sendMail($m, "CarShare - Rent request", $text)
    _alert("Your request was sent, you will receive an email shortly.")
    showLogin()
  } else {
    _alert("Check your email address.")
  }
}
```

Der Administrator kann das Passwort eines Benutzers zurücksetzen (Listing 4.10). Eine naive Umsetzung dieses Features wird dadurch realisiert, das eine Hashsumme gebildet wird und diese als neues Passwort verwendet wird. Der Benutzer wird dann über dies benachrichtigt.

Listing 4.10: Passwort-Reset-Funktionalität der Car-Share-App in der Plain-DSL

```
prop user : User
op resetPassword = {
  user.password = _md5(user.email + user.password)
  _sendMail (user.email, "CarShare - Password reset", "Password reset: " + user.password)
  _alert ("User was reset and notified.")
}
```

4.5 Konventionen

Zusammenfassend werden hier alle Konventionen der Plain-DSL und der Generatoren aufgeführt:

- Eine Operation `showXY` im ViewModel ermöglicht eine Navigation zu View `XY`.
- Eine Operation `saveState` im ViewModel speichert den Zustand der Anwendung lokal ab.
- Zur standardmäßigen Darstellung eines Model-Elementes wird die erste Property verwendet, daher ist es sinnvoll eine Information wie den Namen oder den Titel als erste Property zu definieren.
- Persistenz wird vereinfacht, indem ein Model-Element als Zustand gekennzeichnet wird. Dieses kann über `saveState` gespeichert werden und beim Start der Anwendung wiederhergestellt wird.
- Beim ersten Start der Anwendung sieht der Benutzer ein initialisiertes Zustands-Model. Alle Properties werden geeignet initialisiert: Listen enthalten ein Element; primitive Properties erhalten einen Standardwert; Properties eines Model-Typs referenzieren ein Model-Objekt. Dies geschieht folgendermaßen:
- Bei der Ermittlung eines Initialwertes für Properties eines Model-Typs wird die Liste der möglichen Werte herangezogen. Diese Liste findet sich in der Property des Zustands-Models, deren Name der Plural des Typs ist.

- Eine Tabelle, die beispielsweise an eine Property `items : List<E>` gebunden ist, besitzt als Standard-Implementierung für die Bearbeitung eines Eintrags folgende Logik: Weise dem Singular der Bindung (`item`) das gewählte Element zu und zeige die View mit dem Namen `E` an, um den Element-Typ `E` zu bearbeiten (sofern diese View existiert).

Diese Konventionen ermöglichen es, auf die Implementierung von Operationen weitestgehend zu verzichten. Dies ist für Prototypen, die mit der DSL entstehen, vorteilhaft. Allerdings diktieren diese Konventionen, wie die Anwendung modelliert werden sollte: Im Zustands-Model müssen sich Listen der Model-Typen finden, deren Name der Plural des Typs ist. In ViewModels sollte der Name von Listen-Properties im Plural stehen. Zu jeder Listen-Property (z.B. `items : List<E>`) sollte eine Property existieren, deren Name der Singular ist und Typ der Elementtyp ist (`item : E`).

Dies ist aber insofern nicht problematisch, als dass beispielsweise bei Web-Anwendungen die Liste der Modell-Elemente wahrscheinlich lokal zwischengespeichert werden muss und der Code an diversen Stellen diese Listen benutzt. Genau diese Rolle übernimmt die entsprechende Liste im Zustands-Model.

4.6 Funktionsumfang

Soll eine Operation im ViewModel selbstprogrammierte Logik enthalten, kann diese in der Plain-DSL implementiert werden. So können bereits prototypische Implementierungen der Anwendungen umgesetzt werden. Dafür wird folgender Teil von Xtend als Ausdruckssprache verwendet:

- Fallunterscheidung mit `if`, `else`, `elseif`.
- Schleifen über Liste mit `for` über Listenelemente.
- Aufruf der ViewModel-Operationen `operation()` und Hilfsfunktionen `_hilfsfunktion(param1, ...)`
- Infix-Operatoren für Arithmetik `+ - * / %`, Vergleiche `> >= < <= == !=` und boolsche Operatoren `&& ||` sowie die Präfix-Operatoren `- !`.
- Lokale Variablen mit `var $variable = ...`

Hier werden die Konventionen verwendet, dass Hilfsfunktionen mit _ beginnen, um sie von Operationen des aktuellen ViewModels abzugrenzen und lokale Variablen mit $ beginnen, um sie von Properties abzugrenzen. Eine Trennung der Schreibweise, um lokale Variablen von Properties bzw. Operationen von Hilfsfunktionen zu trennen, vereinfacht auch die Übersetzung in Zielcode. Folgende Hilfsfunktionen stehen zur Verfügung:

- `_validMail(email)` ermittelt, ob die angegebene Mail-Adresse gültig ist.
- `_sendMail(email, subject, text)` öffnet ein Mail-Programm, in welchem eine Mail mit dem angegebenen Inhalt versendet werden kann.
- `_md5(string)` berechnet die MD5-Hashsumme des angegebenen Strings.
- `_alert(string)` gibt eine Benachrichtigung aus.
- `_dateBefore(first, second)` ermittelt, ob das erste Datum zeitlich vor dem zweiten liegt.

Die Menge der Hilfsfunktionen ist nicht auf diese beschränkt, sondern erweiterbar. Da die Erweiterung allerdings plattformspezifisch ist, wird darauf in Abschnitt 5.4.4 für JavaScript und Abschnitt 5.5.4 für Android eingegangen.

5 Implementierung einer plattformunabhängigen DSL für GUIs

5.1 Tools

Zur Erstellung von DSLs oder anderer künstlicher Sprachen kann der Entwickler auf ausgereifte und mächtige Tools zurückgreifen. Dadurch verliert man evtl. die Kontrolle über Einzelschritte des Kompilierungsvorgangs, gewinnt aber andere Vorteile und ist wesentlich schneller beim Ergebnis. Hier sollen die beiden Möglichkeiten Parsergeneratoren und Language Workbenches in Betracht gezogen werden.

5.1.1 Erzeugung von Lexer und Parser mit Parsergeneratoren

Die klassischen Schritte im Kompilierungsvorgang sind diese: „lexikalische Analyse, Syntaxanalyse, semantische Anayse, Zwischencode-Erzeugung, Code-Optimierung und Code-Erzeugung“ [Fischer2011, S. 181]. Man benötigt ein Programm oder eine Funktion, die die jeweiligen Schritte durchführt: Lexer, Parser, Übersetzer.

Die Programme Lexer und Parser lassen sich ausgehend von einer Grammatik generieren, sodass ein Übersetzer ohne Details wie String-Verarbeitung programmiert werden kann. Insbesondere der Verarbeitungsschritt, der Quelltext in einen Token-Strom umwandelt, muss nicht manuell durchgeführt werden, stattdessen kann man sich auf die Weiterverarbeitung der Nonterminale seiner Grammatik konzentrieren.

Ein Werkzeug wie fslex/fsyacc [FsLexYacc] erzeugt für für eine selbstdefinierte Sprache einen Lexer und einen Parser, wobei die Ausgabe des Parsers selbst programmierbar ist. Die Definition der eigenen Sprache liegt in einer Art erweiterten Backus-Naur-Form (EBNF) vor. Werkzeuge wie fslex/fsyacc nennt man „Parsergeneratoren“. Hier kann der Programmierer in F# geschriebene Funktionen angeben, die den semantischen Wert für die Nonterminale der linken Seite einer Regel der Grammatik bestimmen.

Die Verwendung von abstrakter Syntax ist empfehlenswert, um das eingelesene Programm in einer Baumstruktur darzustellen, die nicht von der konkreten Syntax abhängig ist. Die erforderlichen Schritte dafür müssen in fslex/fsyacc manuell implementiert werden: Dazu gehört die Erstellung von entsprechende Typen und das Zusammensetzen der Werte in den Regeln der Grammatik. Ist dies umgesetzt, ist ein Vorteil in fslex/fsyacc, dass das Parsen eine Funktion ist, die Quelltext in einen zusammengesetzten Wert abbildet. Die Funktion ist einfach zu verwenden und zu testen.

Weitere Schritte wie die Übersetzung der Programm-Bäume in effizientere Programm-Bäume oder die Erzeugung des Ziel-Codes wird nicht wesentlich durch Bibliotheksfunktionen vereinfacht. In F# stehen lediglich

Funktion höherer Ordnung (`map`, `filter`, `fold`) zur Verfügung, die für Teile dieser Aufgaben nützlich sind. Zudem sind Fehlernachrichten bei ungültiger Eingabe schwer zu verstehen.

5.1.2 Programmierung mit einer Language Workbench

Eine Language Workbench ist eine Entwicklungsumgebung (IDE) für DSLs. Eine sehr ausgereifte Language Workbench ist Eclipse Xtext [Xtext]. Dabei handelt es sich um die IDE Eclipse mit dem Plugin Xtext, das viele Aufgaben bei der DSL-Entwicklung übernimmt.

Bei Xtext wird aus der Grammatik ein Lexer, Parser und eine Klassenhierarchie für die abstrakte Syntax erstellt. Die Klassen dieser Hierarchie werden als Metamodell und eine Instanz wird als Modell bezeichnet.

Die Validierung prüft die Gültigkeit des Modells, damit die Generierung nur auf validen Modellen durchgeführt wird. Dazu implementiert man eine Validierer-Klasse, die das Modell als Eingabe erhält und Teile des Modells mit Warnungen oder Fehlern versehen kann, damit gemeldet werden kann, welcher Teil des Modells ungültig ist.

Möchte man aus dem Modell Quellcode oder andere textbasierte Artefakte generieren, implementiert man eine Generator-Klasse, die das Modell als Eingabe erhält und Text-Dateien erzeugen kann. Dies ist leider auch eine Einschränkung: Bei Xtext-Generatoren ist vorgesehen, dass diese nur textbasierte Artefakte erzeugen, d.h. String-Daten, keine beliebigen Binärdateien. Für Strings steht allerdings eine mächtige String-Expandierungs-Funktionalität zur Verfügung. Dazu werden die mit `'''`-begrenzten Strings (sog. Templates) verwendet, die auch Schleifen und Fallunterscheidung enthalten können [XtendTemplates]. Richtig verwendet, ist der generierte Code korrekt eingerückt. Dies ist insbesondere bei der Entwicklung der Generatoren von Vorteil ist, da man teilweise den generierten Code selbst lesen möchte.

Ein großer Vorteil von Xtext ist, dass ein Editor miterzeugt werden kann, der über die üblichen Features von IDEs verfügt, wie etwa Syntax-Hervorhebung, Autovervollständigung und Code-Outline. Zudem wird der Validierer im Editor nach jeder Eingabe verwendet, um Warnungen und Fehler als Unterstreichungen mitzuteilen. Der Generator wird beim Speichern ausgeführt und erzeugt die Text-Dateien in einem Ausgabeordner.

Auch in diesem Ansatz lassen sich die Validierer- und Generatorklassen testen. Im Fall von Xtext stehen Java-Unit-Tests zu Verfügung.

5.1.3 Entscheidung für Xtext

Da die Vorteile der Language Workbench Xtext die des Parsergenerators fslex/fsyacc überwiegen, habe ich mich für Xtext entschieden. Die Nachteile von fslex/fsyacc wie unzureichende Fehlerausgaben und viel manueller Implementierungsaufwand sind ein Hindernis bei der Umsetzung.

Die Generierung eines Editors für die DSL sowie in den Editor integrierte Validierer und Generatoren sind ein großer Vorteil. Zudem erfolgt die Entwicklung in der Programmiersprache Xtend, die sich gut für die Generator-Entwicklung eignet. Demgegenüber steht die Entwicklung mit F#, die zwar mit den zugrundeliegenden Paradigmen (objektorientiert und funktional) mächtig ist, aber beispielsweise für die Entwicklung von Generatoren nicht besonders geeignet ist. Xtend verfügt ebenfalls über Funktionen höherer Ordnung und stellt mit String-Templates eine mächtige Grundlage dar.

5.1.4 Xtext-Modell-Generierung aus Grammatik

Aus einer DSL-Beschreibung erzeugt Xtext Java-Klassen, die das geparste DSL-Dokument als Objektgeflecht darstellt [XtextGrammar]. In Xtext wird die Beschreibung in einer xtext-Datei definiert. Diese ähnelt der erweiterten Backus-Naur-Form (EBNF), aber erzeugt bei einer Regelanwendung zusätzlich ein Objekt, das die geparsten Daten enthält. Es kann angegeben werden, wie die zu erzeugende Klasse heißt und wie die

Properties heißen sollen, in denen die Daten gespeichert werden. Alle Syntax-Regeln beschreiben zusammen, wie das resultierende Metamodell aussieht:

Aus der Information `Regelname: Klasse1 ... | Klasse2 ...` wird abgeleitet, dass der semantische Wert dieser Regel vom Typ `Regelname` ist. Dabei wird für die linke Seite der Alternative, die mit `|` notiert wird, ein Objekt der Klasse `Klasse1` erzeugt, für die rechte Seite `Klasse2`. Beide Klassen müssen den Obertyp `Regelname` haben. Die mit `...` ausgesparten Bestandteile der Regel sind folgendermaßen aufgebaut:

- Keywords wie etwa `'keyword'` oder `'{'`, die nur konsumiert werden und für den semantischen Wert nicht interessant sind.
- Property-Zuweisungen wie etwa `name=NAME`, die den semantischen Wert der Regel `NAME` einer Property `name` mit passendem Typ zuweisen.
- Property-Listen-Zuweisungen wie etwa `items+=UIItem`, die den semantischen Wert der Regel `UIItem` der Listen-Property `items` hinzufügen.
- Boolsche-Property-Zuweisungen wie etwa `isStatic?='static'`, die eine Property `isStatic` vom Typ `boolean` auf `true` setzt, wenn `'static'` folgt, sonst `false`. Dies entspricht `('static')?` mit dem Zusatz, dass das Ergebnis gespeichert wird.
- Referenzen wie etwa `person=[Person]`, die den Namen eines bestehendes Person-Objektes erwarten (indem sie auf die Name-Property zugreifen); das referenzierte Objekt wird der Property `person` mit passendem Typ zugewiesen.
- Eine Stern-Regel dieser Bestandteile (keine bis beliebig viele Vorkommen), die mit `*` suffix-notiert ist.
- Eine Plus-Regel dieser Bestandteile (eine bis beliebig viele Vorkommen), die mit `+` suffix-notiert ist.
- Eine optionale Regel dieser Bestandteile (eine oder kein Vorkommen), die mit `?` suffix-notiert ist.
- Eine Alternative dieser Bestandteile, die mit `|` infix-notiert ist.
- Eine durch Hintereinanderschreiben notierte Folge dieser Bestandteile.
- Eine Klammerung dieser Bestandteile.

5.2 Syntax der Plain-DSL

Nachdem der Aufbau von Xtext-Regeln geklärt wurde, werden nun die Syntax-Regeln der Plain-DSL und der Aufbau der DSL-Elemente erläutert. Die folgenden Listings zeigen jeweils Ausschnitte der xtext-Datei, die zusammen die gesamte Syntax-Beschreibung darstellen.

Die Anwendung ist ein Objekt vom Typ `App` (Listing 5.1), das einen Namen und einen Titel besitzt und Models, ViewModels, Views und Transformationen enthält. Die Reihenfolge dieser Elemente ist beliebig. Jedes wichtige Element der DSL besitzt ggf. weitere Attribute (`Attributes`), die der Erweiterbarkeit dienen.

Listing 5.1: Plain-DSL - App

```
grammar plain.Plain with org.eclipse.xtext.xbase.Xbase
generate plain "http://plain-dsl.org"
App:
  'app' name=ID title=STRING? attr=Attributes? '{'
  ( models+=Model
  | viewmodels+=ViewModel
  | views+=View
  | transforms+=Transform)*
  '}';
```

Models (Listing 5.2) besitzen einen Namen, Properties und ggf. eine Invariante über alle Properties des Models. Eine Property hat einen Namen, einen Typ und optional einen Initialwert und eine Invariante.

Als Typen kommen alle durch den Benutzer definierten Modell-Typen infrage, ebenso wie primitive Java-Typen. Dies ist keine Einschränkung in Bezug auf Plattformunabhängigkeit: Die Zahlentypen `Integer` und `Double` stehen für die Information, ob eine Property eine Ganz- oder Kommazahl ist. Ebenso unabhängig sind die Typen `Boolean` und `String`. Die Typen `Password`, `Email`, `Phone` und `Text` existieren in der Java-Basisbibliothek nicht und stellen Platzhalter für den primitiven String-Typ dar. Diese Vorgehen wird auch in [Groenewegen2013] für Typen der Web-Domäne gewählt. `Date` wird als Name des Typen für Zeitpunkte verwendet (der als Entsprechung den Java-Typ `Date` besitzt). Der Typ `List<T>` wird als Typ für Sammlungen verwendet (und besitzt als Entsprechung den Java-Typ `List<T>`).

Listing 5.2: Plain-DSL - Model

```
Model:
  'model' name=ID attr=Attributes? '{' ('inv' inv=Expr)? properties+=Property* '}';
Property:
  'prop' name=ID ':' type=JvmTypeReference ('=' value=Expr)? attr=Attributes? ('inv' inv=Expr)?;
```

Ein ViewModel (Listing 5.3) besitzt einen Namen, Properties, Operationen und ggf. eine Invariante über alle Properties des ViewModels. Ein ViewModel referenziert ggf. ein Model, aus dem es seine Daten bezieht. Eine Operation besitzt einen Namen und eine optionale Implementierung.

Listing 5.3: Plain-DSL - ViewModel

```
ViewModel:
  'viewmodel' name=ID ('uses' model=[Model])? attr=Attributes? '{' ('inv' inv=Expr)?
  (properties+=Property | operations+=Operation)*
  '}';
Operation:
  'op' name=ID attr=Attributes? ('=' block=Expr)?;
```

Eine View (Listing 5.4) hat einen Namen, kann einen Titel haben und besteht aus View Items. Sie kann ein ViewModel referenzieren, sodass sie auf dessen Operationen und Properties zugreifen kann. View Items besitzen ein optionales Label und sind entweder nicht gebundenen (Panel), oder gebunden (Action, Table, Edit und Custom). Gebundene View Items besitzen eine Referenz auf eine Property bzw. Operation des ViewModels.

Listing 5.4: Plain-DSL - View und View Item

```
View:
  'view' name=ID (title=STRING)? ('uses' viewmodel=[ViewModel])? attr=Attributes?' {'
  items+=ViewItem*
  '}';
ViewItem:
  {ViewPanel} 'panel' label=STRING? attr=Attributes? '{' items+=ViewItem* '}'
| {ViewAction} 'action' (label=STRING ':')? bind=Path attr=Attributes?
| {ViewTable} 'table' (label=STRING ':')? bind=Path attr=Attributes?
| {ViewEdit} 'edit' (label=STRING ':')? bind=Path attr=Attributes?
| {ViewCustom} 'custom' type=ID (label=STRING ':')? bind=Path attr=Attributes?;
```

Eine Transformation (Listing 5.5) erfordert die spezifische Angabe von mindestens einem Attributwert, abhängig davon wird ein passender Ausdruck erwartet. Transformationen sind für die in Abschnitt 5.9 erwähnte Lokalisierung und die in Abschnitt 5.6 diskutierte GUI-Anpassung konzipiert.

Listing 5.5: Plain-DSL - Transform

```
Transform: 'transform' name=NAME 'for' attr=Attributes? '{' body=Expr '}';
```

Attribute (Listing 5.6) bestehen aus Schlüssel-Wert-Paaren, wobei der Schlüssel ein Identifier und der optionale Wert ein Ausdruck ist. Dieser Ausdruck wird in einem `Expr`-Objekt gepeichert und ist intern ein Xtend-Ausdruck. Dieser kann für boolsche Ausdrücke in Invarianten oder für String-Literale eingesetzt werden. Für Datenbindung steht der Typ `Path` zur Verfügung, intern wieder durch einen Xtend-Ausdruck umgesetzt. Über diesen Typ werden Properties und Operation des ViewModels referenziert.

Listing 5.6: Plain-DSL - Attributes und Exprs

```
Attributes: '[' entries+=Attribute (',' entries+=Attribute)* ']';
Attribute: key=ID ('=' value=Expr)?;
Expr: value=XExpression;
Path: value=XExpression;
```

Durch die Benutzung von Xtend-Ausdrücken über den Typ `XExpression` wird im Editor das Feature Autovervollständigung angeboten. Dazu muss in Xtext eine Java-Klassen-Entsprechung bereitgestellt werden: Models, ViewModels und Views werden in gleichnamige Klassen transformiert. Diese Klassen verfügen über Felder und Methoden, die den Properties und Operationen entsprechen. Diese Klassen werden nur während der Entwicklung verwendet, nicht etwa für den Android-Java-Generator.

Der Parser erzeugt aus einer Plain-DSL-Datei der angegebenen Grammatik ein `App`-Objekt, das die eingegebenen Daten enthält. Dieses Objekt stellt im Rahmen dieser Arbeit die plattformunabhängige Beschreibung der Anwendung dar. Der Validierer arbeiten auf diesem Objekten, um die Gültigkeit zu ermitteln. Generatoren greifen über ein Interface auf diese Daten zu, um Anwendungen der Zielplattformen zu erzeugen.

Da die Validierung und Generierung umfangreiche Aufgaben sind, ist hier eine testgetriebene Entwicklung vorteilhaft, die im Folgenden erläutert wird.

5.3 Testgetriebene Entwicklung mit Xtext

Ein Generator ist potentiell ein umfangreiches Programm, sodass es besonders wichtig ist, das Verhalten des Codes genau zu dokumentieren und zu spezifizieren. Dafür sind Unit-Tests ein wertvolles Hilfsmittel. Um allerdings Generator- und Validierer-Klassen in Xtext sinnvoll zu testen, müssen diverse Annotationen angegeben werden, wie in Listing 5.7. Dazu gehören ein spezieller JUnit-Test-Runner (1), eine Registrierung für Dependency-Injection (2), Hilfsklassen für die Validierung (3), die Generierung (4) und das Parsen (5).

Listing 5.7: Annotationen für das Testen von Generator und Validierer unter Xtext in Xtend

```
...Imports...
@RunWith(XtextRunner)                       // 1
@InjectWith(PlainInjectorProvider)          // 2
class Tests {
  @Inject ValidationTestHelper validator    // 3
  @Inject CompilationTestHelper compiler    // 4
  @Inject ParseHelper<App> parser           // 5
  ...Testfälle und Hilfsmethoden wie read, parse, validate...
}
```

5.3.1 Testgetriebene Entwicklung der Validierer

Um die Validierung zu testen, muss die Klasse `PlainValidator` getestet werden. Ein Test wäre beispielsweise, dass die Blog-Anwendung vom Validierer akzeptiert wird. Dies wird in Listing 5.8 dargestellt. Die `validate`-Methode erhält ein DSL-Dokument und eine Liste von Fehlercodes, die als Konstanten der Klasse `PlainValidator` definiert sind. Durch einen statischen Import müssen diese nicht voll qualifiziert referenziert werden. Die Validierung schlägt fehl, wenn der Validierer nicht dieselben Fehlercodes liefert.

In Xtend kann statt `f(x)` auch `x.f` geschrieben werden. Für Methoden mit mehreren Parametern ist dies auch möglich, indem der erste Parameter vor die Methode geschrieben wird: `g(x, y)` kann als `x.g(y)` geschrieben werden. Gestaltet man die Test-Methoden passend, kann Xtend als eine Art interne Test-DSL verwendet werden.

Listing 5.8: Testen des Validierers mit der Blog-Anwendung

```
@Test def validateBlog() {
  val code = read("Blog.plain")
```

```
  code.validate(#[]) // No errors expected
}
```

Das Testen der Funktionsweise des Validierers durch Negativtests ist wichtig, da man als Benutzer daran interessiert ist, dass der Validierer im Fehlerfall eine hilfreiche Meldung ausgibt. In Listing 5.9 werden Minimalbeispiele für fehlerhafte Plain-DSL-Programme mit dem erwarteten Fehler angegeben.

Listing 5.9: Negativtesten des Validierers mit Minimalbeispielen

```
@Test def validateDuplicateName() {
  val code = "app A { model M { prop p1 : int  prop p1 : int } }"
  code.validate(#[duplicateName])
}
@Test def validateInvalidProperty() {
  val code = "app A { viewmodel VM { prop p1 : int } view V { edit [p2] } }"
  code.validate(#[invalidPath])
}
@Test def validateInvalidPropertyType() {
  val code = "app A { viewmodel VM { prop p1 : int } view V { table [p1] } }"
  code.validate(#[invalidType])
}
```

5.3.2 Testgetriebene Entwicklung der Generatoren

Mithilfe von Unit-Tests lässt sich auch spezifizieren, was die Generatoren als Ausgabe produzieren. In Listing 5.10 soll der Android-Generator sowohl für das ViewModel als auch für die View eine Klasse mit passendem Namen erstellen. Die Hilfsmethode `generateAndroid` erstellt ein Android-Generator-Objekt und führt diesen auf dem angegebenen Dokument aus. Das Resultat ist eine Menge von Dateien und ihr Inhalt. Diese Menge wird im Beispiel daraufhin untersucht, ob bestimmte Dateien generiert wurden und ob diese Dateien bestimmte Informationen enthalten.

In Xtend wird dies mit Lambda-Ausdrücken realisiert, die in eckigen Klammern angegeben werden: `[x | x.name]` ist beispielsweise die Abbildung eines Elementes auf dessen Namen. Dies kann verkürzt als `[name]` geschrieben werden. Diese Technik wird mit der Schreibweise `x.f` kombiniert, um die nächsten Tests zu formulieren. Zudem können Lambda-Ausdrücke als Parameter statt in auch hinter die Parameterliste geschrieben werden: `m(x)[...]` anstelle von `m(x, [...])`. Wieder ist dieses Wissen in die Gestaltung der Test-Methoden eingeflossen, um eine interne Test-DSL zu schaffen.

Listing 5.10: Negativtesten des Validierers mit Minimalbeispielen

```
@Test def genAndroidClassesForViewModelAndView() {
  val code = "app A { viewmodel VM { prop p1 : int } view V { edit [p1] } }"
  code.generateAndroid [
    assertFile("viewmodel/VM.java") [ assertContains("class VM ") ]
    assertFile("view/View.java") [ assertContains("class View ") ]
  ]
}
```

Die Methode `generateHTML` funktioniert analog unter Verwendung des HTML-Generators. Das Beispiel in Listing 5.11 liest das Eingabedokument über die Methode `read` als String aus einer Datei. Das Generat wird daraufhin getestet, ob eine View in einen `div`-Container und ein Modell in eine JavaScript-Konstruktor-Funktion übersetzt wird. In Xtend können String-Literale auch mit `'''` begrenzt werden, was praktisch ist, wenn Anführungszeichen (`"`) als Bestandteil des Strings benötigt wird.

Listing 5.11: Testen des Generators über externe Dateien

```
@Test def genBlogHTML() {
  val code = read("Blog.plain")
  code.generateHTML [
    assertFile("Blog.html") [ assertContains('''<div id="Dashboard"''') ]
    assertFile("BlogHtml/model.js") [ assertContains("function Article(") ]
  ]
}
```

Auf diese Weise können die Abbildungen, die in den nächsten Abschnitten beschrieben werden, genau getestet werden.

5.4 Generierung der Web-Anwendung

Ziel des HTML-Generators ist es, aus einer plattformunabhängigen Beschreibung eine Web-Anwendung zu erstellen. Hier wird die Plain-DSL als Beschreibungssprache verwendet.

Es ist vorteilhaft, die Anwendung in eine HTML-Seite zu übersetzen, weil so Navigation zwischen verschiedenen Seiten vermieden wird. Navigation würde dazu führen, Daten beim Seitenübergang zu speichern und wiederherzustellen. Stattdessen werden alle Views in der HTML-Seite erzeugt, wobei zur Laufzeit nur die aktive View angezeigt wird und alle anderen Views verborgen werden.

Da jedoch eine einzelne HTML-Seite als Generat zu umfangreich ist, werden zusätzlich noch weitere Dateien in einem Unterordner generiert. Eine Web-Anwendung besteht damit aus den folgenden Teilen:

```
app id title attr { models viewmodels views transforms } ↦
```

Für ...	Art und Bezeichnung	Inhalt
Models	JS-Datei `models` im Unterordner	Übersetzung der Models
ViewModels	JS-Datei `viewmodels` im Unterordner	Übersetzung der ViewModels
App	HTML-Datei im Hauptordner	Übersetzung der Views, HTML-Boilerplate
Style	CSS-Datei `style` im Unterordner	Inhalt des Transforms, z.T. statisch
Lokalisierung	JS-Datei `localization` im Unterordner	Übersetzung des Transforms
statisch	JS-Datei `util` im Unterordner	gemeinsame Funktionen

Da die Zielplattform Datenbindung nicht direkt unterstützt, verwende ich hierfür die Javascript-Bibliothek knockout.js [KnockoutJS]. Auf diese Weise können Properties aus ViewModel-Objekten direkt an Textfelder und andere Controls gebunden werden.

5.4.1 Generierung der GUI mit HTML

Die Generierung einer HTML-Anwendung lässt sich durch Abbildungen der plattformunabhängigen Darstellung nach HTML beschrieben. Eine detaillierte Abbildung wäre zu umfangreich, da auf viele Framework-spezifische Implementierungsdetails eingegangen werden müsste. Daher wird nur eine grobe Abbildung beschrieben, in der Details ausgelassen und mit den Auslassungszeichen `...` gekennzeichnet werden. Zwischen diese Auslassungszeichen kann vermerkt sein, was ausgelassen wurde. Eine Auslassung, die mit `...ggf.` gekennzeichnet ist, steht dafür, dass an dieser Stelle nicht immer Code generiert wird. Der zu generierende Code kann dadurch reduziert werden, dass einige Details durch gemeinsame Funktionen gekapselt sind.

Der HTML-Dokument, das die GUI darstellt, besteht aus den Abschnitten `head` und `body`. Im `head`-Tag werden die CSS- und JS-Dateien eingebunden. Existiert ein Style-Transform-Objekt (siehe Abschnitt 5.6), wird dieses mit einigen statischen Anpassungen in die `style.css`-Datei generiert. Die CSS- und JS-Dateien werden der Übersichtlichkeit halber in ein eigenes Verzeichnis generiert. Das `body`-Tag enthält wie oben erwähnt alle Views.

Die Verbindung der ViewModels mit der Seite in JavaScript wird als letzter Eintrag des `body`-Tags durchgeführt. Ein weiterer statischer Bestandteil dieses Tags ist ein `div`-Container, um Benachrichtigungen anzuzeigen. Dieser Container ist anfangs versteckt, wird bei Bedarf eingeblendet und kann durch einen Klick geschlossen werden.

```
app id title attr { models viewmodels views transforms } ↦
<html>
  <head>...Titel der Anwendung, Einbinden von CSS- und JS-Dateien...</head>
  <body ...>
    <div ...>...versteckter Container für eine Benachrichtigungs-Box...</div>
    ...Alle Views...
    <script ...>...JavaScript-Setup-Code...</script>
  </body>
</html>
```

Eine View wird in einen `div`-Container übersetzt. Als Inhalt der View werden die ihr untergeordneten Items in den `div`-Container generiert.

```
view id title attr { items } ↦
<div class="panel ..." id="...id der View...">
  <div ...>...Titel...</div>
  <div ...>...untergeordnete View Items...</div>
</div>
```

Ein Panel gruppiert Items und wird ähnlich wie eine View in einen `div`-Container übersetzt. Das Label des Panels ist optional. Die untergeordneten View Items werden hintereinander in HTML übersetzt und nicht explizit angeordnet. Dadurch wird die Standard-Anordnung des Browsers für HTML-Seiten verwendet, die Inline-Elemente nebeneinander mit automatischem Umbruch anordnet.

```
panel title attr { items } ↦
<div class="panel" ...>
  ...ggf. Label des Panels als div...
  <div ...>...untergeordnete View Items...</div>
</div>
```

Ein Action-Element wird in einen Button übersetzt, welcher bei einem Klick die entsprechende Operation des ViewModels ausführt. Dazu wird der Name der Operation angegeben. Das HTML-Element `button` wird dem `a`-Element für Links vorgezogen, weil bei ersterem die `enable`-Eigenschaft gesetzt werden kann. Diese führt dazu, dass das Element nicht anwählbar ist und dies visuell deutlich wird.

```
action label clickOp attr ↦
<button class="btn ..." data-bind="click: ...clickOp...">...Label...</button>
```

Ein Table-Element wird in einen `div`-Container mit einer Tabelle übersetzt. Diese bezieht ihre Daten über Datenbindung und stellt die Einträge als Tabellen-Zeilen dar. Eine Tabelle, die nicht als `readonly` gekennzeichnet ist, enthält Buttons zum Hinzufügen, Löschen und Editieren von Einträgen.

```
table label bind [props="p1, ..."] ↦
<div class="panel">
  ...ggf. Label und Buttons zum Hinzufügen und Löschen...
  <div ...>
    <table>
      <tr>
        ...Spaltenbeschriftungen als th-Tag für die Namen der anzuzeigenden Properties...
        ...ggf. Spaltenbeschriftungen Löschen...
      </tr>
      <tr data-template="foreach: ...Bindung...">
        ...td-Tag für Werte der anzuzeigenden Properties...
        ...ggf. Spalten für einen Editier-Button und eine Löschen-Checkbox...
      </tr>
    </table>
  </div>
</div>
```

Um ein Edit-Element zur Bearbeitung einer Property umzusetzen, wird nach dem Property-Typ unterschieden. Die Abbildung ist in Tabelle 5.2 dargestellt.

```
edit label bind attr ↦
<div ...>
  ...ggf. Label...
  ...Eingabe-Control, das den Bindungstyp unterstützt...
<div>
```

Typ bzw. Typen	HTML-Control
boolean, Boolean	<checkbox data-bind="checked: ...Property-Pfad..."... />
int, Integer	<input type="number" data-bind="value: ...Property-Pfad..."... />
double, Double	<input type="number"... />
String	<input type="text"... />
Password	<input type="password"... />
Date	<input type="datetime"... />
Email	<input type="email"... />
Phone	<input type="phone"... />
Text	<textarea data-bind="value: ...Property-Pfad..."... />
$M \in Models$	<select data-bind="options: ...Liste von Objekten vom Typ M..."... />

Tabelle 5.2: Abbildung der Property-Typen nach HTML-Controls

5.4.2 Generierung der Logik in Javascript

Zunächst wird gezeigt, wie in JavaScript benutzerdefinierte Typen erstellt werden. Dazu stellt man eine sogenannte Konstruktor-Funktion bereit, die einige Felder zuweist und sich selbst zurückgibt. Beispielsweise lässt sich folgendermaßen ein zweidimensionaler Punkt mit X- und Y-Koordinate erstellen. Jeder Konstruktor-Aufruf `new Pos()` gibt einen neuen Punkt zurück, wie man es aus z.B. Java gewohnt ist.

```
function Pos() {
  this.x = 0;
  this.y = 0;
  return this;
}
```

Damit die so erstellten Objekten über Funktionen verfügen, die auf die Daten des Objektes zugreifen können, ist es in JavaScript vonnöten, die Referenz auf das Objekt in einer Variablen (z.B. `self`) zu speichern. Folgendes Beispiel illustriert das Problem:

```
function Counter() {
  this.x = 42;
  this.incr = function() { this.x++; };
  return this;
}
```

Der Code `this.x++;` greift nicht auf den Wert zu, der mit `this.x = 42;` initialisiert wurde. Der Grund dafür ist, dass `this` an die aktuelle Funktion gebunden ist und nicht das Counter-Objekt. Die Lösung ist, die Variable `self` zu verwenden:

```
function Counter() {
  var self = this;
  self.x = 42;
  self.incr = function() { self.x++; };
  return self;
}
```

Nachdem die Grundlagen geklärt wurden, kann die Interaktionslogik in Java-Script generiert werden. Diese besteht aus mehreren Teilen: der statischen Datei `util.js` für grundlegende JavaScript-Funktionen und jeweils einer Datei für Models und ViewModels.

Models werden in einfache JavaScript-Konstruktor-Funktionen übersetzt, die Strings, Zahlen, Wahrheitswerte, Arrays oder JavaScript-Objekte dieser Art enthalten. Funktionen oder Objekte anderen Typs werden nicht in diesen Objekten gespeichert, um eine möglichst einfache Persistenz zu ermöglichen.

```
model id attr { props } ↦
```

```
function ...Model-ID...(...Argumente...) {
  var self = this;
  ...Vereinfachte Übersetzung der Properties in Felder, etwa: self.prop = init;...
  return self;
}
```

ViewModels werden in JavaScript-Konstruktor-Funktionen übersetzt. Das ViewModel referenziert entweder ein bestimmtes Model-Objekt oder das Zustands-Modell-Objekt. Ein Model-Objekt kann als Konstruktorargument übergeben werden, andernfalls ist der Wert `undefined` und es wird ein neues Element erzeugt. Für ein ViewModel wird ebenfalls eine Funktion, die den Konstruktor aufruft, generiert. Diese wird für die generierten Listen-Funktionen verwendet.

```
viewmodel id attr { props ops } ↦
```

```
function ...ViewModel-ID...(m) {
  var self = this;
  self.model = ...Übergebener Wert m, neues Element oder das Zustands-Modell-Objekt...;
  ...Übersetzung der Properties in Properties des Datenbindungsframeworks...
  ...Übersetzung der Operationen...
  return self;
}
util.new...ViewModel-ID... = function(m) { return new ...ViewModel-ID...(m); };
```

Der ViewModel-Typ enthält Properties des Datenbindungsframeworks [KnockoutJS] und besitzt Funktionen mit einer Implementierung. Für Properties, die einen Listen-Typ ausweisen, werden weitere Funktionen generiert. Diese werden von der generierten HTML-GUI zum Hinzufügen, Löschen und Editieren von Einträgen benötigt.

```
prop id : List<E> = val attr ↦
```

```
...ggf. Singular-Property, sofern nicht bereits generiert...
self....Property-ID... = util.bindingArray(...);
self.add...Property-ID... = function() { ...neues Element einfügen... };
self.remove...Property-ID... = function() { ...neues Element einfügen... };
self.select...Property-ID... = function(x) {
  ...Zuweisen der Singular-Property...
  ...Anzeigen der View für den Typ...
};
```

Properties, die nicht von einem Listen-Typ sind, verweisen entweder auf eine gleichnamige Property des zugrundeliegenden Models oder werden nur im ViewModel benötigt. Bei dem Verweis auf die Model-Property wird an diese gebunden (`util.binding(...)`), andernfalls wird eine neue Property des Datenbindungsframeworks erzeugt (`ko.observable(...Initialwert...)`).

```
prop id : E = val attr ↦
```

```
self....Property-ID... = ...Bindung an Model oder neue Property...;
```

Wie in Abschnitt 4.5 erläutert, wurden Konventionen bzgl. der Operationen eingeführt: Eine Operation `showXY` wird in eine Navigations-Funktion umgesetzt, welche die View `XY` anzeigt. Eine Operation `saveState` speichert den Zustand der Anwendung lokal ab. Dazu wird eine JSON-Repräsentation des Zustand-Modells in den Local Storage geschrieben. Beim Start der Anwendung wird ein vorhandenes Zustands-Modell-Objekt verwendet. Ansonsten wird neues Objekt erzeugt, das wie in Abschnitt 5.4.3 beschrieben, initialisiert ist. Besitzt eine Operation eine Implementierung, so wird diese in JavaScript umgewandelt und als Funktionsrumpf verwendet, genaues zum Funktionsumfang findet sich im Abschnitt 4.6, die Umsetzung findet sich in Abschnitt 5.4.4.

```
op id = impl ↦
```

```
self....Operation-ID... = function() { ...Übersetzung der Implementation/Konvention... }
```

5.4.3 Anmerkung zu undefinierten Werten

Bei Datenbindung ist in den Frameworks WPF und JavaFX darauf zu achten, dass Werte nicht immer definiert sind: Als Beispiel wird hier eine Tabelle aller Artikel mit den folgenden Spalten modelliert: Titel, Kategorie und Autor. Es reicht in der Tabelle nicht, an die Kategorie eines Artikel zu binden, weil dieses kein primitives Datum ist, sondern ein Modell-Objekt, dass sich nicht ohne Weiteres darstellen lässt. Um die Kategorie als Wert in einer Spalte anzuzeigen, kann der Name der Kategorie angezeigt werden.

Der Pfad für die Kategorie-Property wäre dann sinngemäß `article.category.name`. Das Problem ist, dass im Falle einer undefinierten Kategorie (die den Wert `null` hat) der Zugriff auf den Namen fehlschlägt. Eine Möglichkeit wäre, den Safe Navigation Operator `?.` zu verwenden, der in einigen Programmiersprachen wie Xtend verfügbar ist.
`article?.category?.name` verhält sich wie der folgende Ausdruck:

```
if(article != null && article.category != null) article.category.name else null
```

Der Operator ist aber nicht in JavaScript oder Java verfügbar, ebenso ist es nicht sinnvoll entsprechenden Code an allen Stellen zu generieren, an denen solche Bindungen vorkommen, da es potentiell sehr viele Stellen sind.

Aus der Darstellung des Problem für andere Frameworks, die Datenbindung unterstützen wird deutlich, dass es sich lohnt, das Problem undefinierter Werte konzeptuell zu lösen:

Das initiale Objektgeflecht sollte vollständig initialisiert werden. So hat der Benutzer eine sinnvolle Bearbeitungsgrundlage beim ersten Start der Anwendung. Hierfür muss das Model, das als Zustand gekennzeichnet ist (im Folgenden: Zustands-Model), mit sinnvollen Standardwerten für Properties versehen werden, ebenso alle Modelle, die dafür benötigt werden und alle verwendeten ViewModels.

Ein naiver Ansatz, um ein Model oder ViewModel zu initialisieren, würde allen Properties ein neues Objekt des Property-Typs zuweisen. Dies würde im Falle von rekursiven Typen allerdings zu einer Endlosschleife führen. Zur besseren Illustration dieses Problems werden Beispiele aus der Blog-App aus Abschnitt 4.4.2 verwendet.

Das Zustands-Model `Blog` enthält eine Liste von Artikeln, Autoren bzw. Kategorien. Nach der Initialisierung enthalten diese Listen ein Element des jeweiligen Elementtyps. So können die Properties der Modellobjekte Referenzen auf diese Objekte als Startwert besitzen. Beispielsweise wird das Element der Artikel-Liste so initialisiert, dass das Element der Autor-Liste als Autor und das Element der Kategorie-Liste als Kategorie verwendet wird. Ein interessanter Fall ist das Model-Element Kategorie, welches einen Namen besitzt und eine Ober-Kategorie referenziert. Das Element der Kategorie-Liste wird so initialisiert, dass es seine eigene Ober-Kategorie ist.

Dieser Ansatz ermöglicht eine Verwendung des Null-Value-Patterns [Eilebrecht2013, S. 186]. D.h. `null` bzw. `undefined` wird gar nicht als Wert für eine Referenz benötigt. Dies erfordert aber ein Umdenken in der Modellierung der Daten. Ein Beispiel illustriert dies: Anstatt einem Artikel keine Kategorie zuzuordnen, wird die Standard-Kategorie „Allgemein“ gewählt. Ebenso wird mit dem Standardwert für Autor verfahren: „Unbekannter Autor“. Bei der Car-Share-Anwendung hat das Null-Value-Objekt für Benutzer den Namen „Niemand“. Dieser Benutzer wird für verfügbare Autos verwendet.

5.4.4 JavaScript-Funktionsumfang

Für prototypische Implementierungen sind die in Abschnitt 4.6 erwähnten Funktionen in JavaScript umgesetzt. Für die Plain-DSL liegt die Implementierung einer ViewModel-Operation als Xtend-Syntax-Baum vor. Eine Baum-Übersetzung generiert JavaScript-Code und ist in Tabelle 5.3 dargestellt. Platzhalter-Variablen wie `block` und `expr` müssen rekursiv abgebildet werden.

Xtend-Konstrukt	JavaScript-Code	
`var $item = expr`	`var item = expr;`	
`$item = expr`	`item = expr;`	
`$item`	`item`	
`prop1.prop2`	`self.prop1().prop2()`	
`prop1.prop2 = expr`	`self.prop1().prop2(expr)`	
`prop1 = expr`	`self.prop1(expr)`	
`prop1`	`self.prop1()`	
`expr binop expr`	`expr binop expr`	binop $\in$ {+, -, *, /, %, ||, &&}
`unaryop expr`	`unaryop expr`	unaryop $\in$ {-, !}
`if(cond) block else block2`	`if(cond) block else block2`	
`_func(param1, ...)`	`util.func(param1, ...)`	
`operation()`	`self.operation();`	

`for($item : list) block`	`var arr = list();`
	`for(var i = 0; i < arr.length; i++){`
	`  var item = arr[i];`
	`  block;`
	`}`	

Tabelle 5.3: Übersetzung des Xtend-Codes in JavaScript

Die gezeigte Übersetzung zeigt die Verwendung der Properties des Datenbindungsframework: Diese sind Funktionen, die entweder ohne Argumente zum Lesen oder mit einem Argument zum Schreiben aufgerufen werden.

Die Hilfsfunktionen `_alert`, `_md5`, `_dateBefore`, `_validMail` und `_sendMail` sind in JavaScript implementiert. Die Übersetzung der Invariante auf Property-, Model- und ViewModel-Ebene konnte im Rahmen dieser Arbeit nicht fertiggestellt werden, nähere Informationen dazu finden sich in Abschnitt 7.2.

5.5 Generierung der Android-Anwendung

Die Entwicklung von Android-Anwendungen besteht aus der Beschreibung der GUI in XML und der Implementierung der Logik in Java. Zusätzlich müssen noch andere Artefakte wie beispielsweise Ressourcendateien definiert werden.

Das MVVM-Architekturmuster aus Abschnitt 2.1 führt zu einer Reduktion des Java-Codes, der sonst viele Zeilen für Adapter-Views und Callbacks vereinnahmen würde. Dazu ist allerdings eine externe Programmbibliothek erforderlich. Ich verwende hier die Programmbibliothek Robobinding [Robobinding], weil diese eine deklarative Datenbindungssyntax in der View-Beschreibung erlaubt; andere MVVM-Bibliotheken sind weniger ausgereift und unzureichend in Entwicklungsumgebungen integriert.

Robobinding ermöglicht es, eine View an ein ViewModel zu binden. Dabei können Properties des ViewModels in einigen Controls der View referenziert werden. Die `get`-Methoden der Properties werden beim Erstellen der View aufgerufen, um einen Wert anzuzeigen. Bei Änderung durch den Benutzer werden `set`-Methoden aufgerufen, um neue Werte zu speichern. Der Entwickler kann Properties als abhängig kennzeichnen. Bei der Änderung einer Property werden alle abhängigen Properties erneut ausgelesen.

Die die Abbildung der plattformunabhängigen Darstellung als Plain-DSL-Dokument in die Android-XML- und Java-Dateien wird in diesem Abschnitt beschrieben. Hier soll nur eine grobe Abbildung beschrieben werden, daher werden Implementierungsdetails ausgelassen. Die generierte Anwendung besteht aus mehreren Teilen, wie folgt dargestellt.

`app id title attr { models viewmodels views transforms }` ↦

Für jede/s/n ...	Art und Bezeichnung	Inhalt
Model	Java-Datei im Unter-Package `model`	Klasse für das Model
ViewModel	Java-Datei im Unter-Package `viewmodel`	Klasse für das ViewModel
View	Java-Dateien im Unter-Package `view`	Activity-Klasse
View	Java-Dateien im Unter-Package `view`	spezifisches ViewModel
View	Android-XML-Datei im Resourcen-Ordner	GUI-XML-Beschreibung
App	Java-Datei ApplicationWithModel	Erzeugung des Initialmodells
App	XML-Datei AndroidManifest	Aufzählung aller Views, Boilerplate
Style	Style-Ressource im Unterordner	Inhalt des Transforms
Lokalisierung	String-Ressource	Übersetzung des Transforms, z.T. statisch
statisch	Java-Dateien	Basisklassen und gemeinsame Funktionen
statisch	Ressourcen-Dateien	Layout für Spinner- und Listenelemente

5.5.1 Generierung der Android-GUI mit XML

Eine View ist ein komplexes Element, was die Übersetzung betrifft, sie erfordert die Übersetzung in drei Dateien: eine XML-Beschreibung (Listing 5.14), eine Android-Activity-Klasse (Listing 5.13) und eine View-spezifische ViewModel-Klasse (Listing 5.12).

`view id title attr { items }` ↦ Code in Listing 5.12, Listing 5.13 und Listing 5.14

Die Activity-Klasse wird in das Unter-Package `view` generiert und ist in Listing 5.12 dargestellt. Das verwendete Datenbindungsframework verbindet eine View mit seinem ViewModel über statische Hilfsmethoden der Klasse `Binder`. Diese erhält eine Referenz auf die XML-Beschreibung und ein ViewModel-Objekt und gibt ein Android-Control-Objekt zurück. Das zurückgegebene Objekt wird als Inhalt der View verwendet. In der Basisklasse `ActivityBase` ist diese Logik gekapselt, sodass in der Unterklasse die Referenz auf die XML-Beschreibung und ein ViewModel-Objekt ausreichen.

Ebenfalls in der Basisklasse `ActivityBase` gekapselt ist der Umgang mit zahlreichen Eigenheiten der Android-Plattform: Dazu zählt das Erhalten des Zustandes beim Drehen des Bildschirms und beim Schließen von Views. Eine Auflistung der Eigenheiten findet sich in Abschnitt 5.5.5.

Listing 5.12: Übersetzung der View, Teil 1: Activity-Klasse in Java

```
...
class ...View-ID... extends ActivityBase... { ...
  protected int getLayout() { return R.layout....View-ID...; }
  protected Object createInitialVM() { return new ...View-spezifisches ViewModel...(); }
  protected Viewable viewable(String binding) {
    if(binding.equals(...))
      return new Viewable... { ...spezifische Datenansicht für Typ der Bindung... }; ...
  }
}
```

Alle Bindungen innerhalb der View, die sich auf Properties von Properties beziehen, müssen in eigene Properties übersetzt werden. Ein Beispiel aus der Blog-App (Listing 8.10) veranschaulicht dies: Die Bindung `article.title` kann in die Methoden `getArticleTitle` und `setArticleTitle` übersetzt werden. Die `get`-Methode hängt von der Property `article` ab und besitzt diese Implementierung: `return getArticle().getTitle();`. Properties dieser Art werden für das verwendete Datenbindungsframework benötigt und befinden sich in einer ViewModel-Klasse, die spezifisch für diese View und von der referenzierten ViewModel-Klasse erbt (Listing 5.13). Im Beispiel ist dies die Klasse `ArticleViewSpecificBlogVM`, die von `BlogVM` erbt.

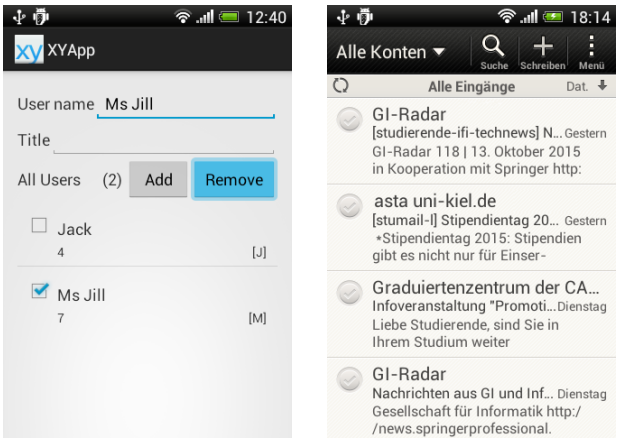


Abbildung 5.1: Tabellenzeilen-Anzeige auf Android-Geräten, inspiriert von Mailprogrammen

Listing 5.13: Übersetzung der View, Teil 2: Spezifisches ViewModel in Java

```
...
@PresentationModel
class ...View-spezifisches ViewModel... extends ...ViewModel-ID... {
  ...Konstruktor...
  ...Properties für View-spezifische Datenbindungen...
}
```

Eine View wird in einen Container übersetzt, der sein Kind-Element mittels Scrollbalken vollständig anzeigen kann. Das Kind-Element zeigt die seine Kind-Elemente wiederum untereinander an.

Listing 5.14: Übersetzung der View, Teil 3: GUI in Android-XML

```
<ScrollViewer ...XML-Namespace und weitere Attribute..>
  <LinearLayout a:orientation="vertical" ...>
  ...untergeordnete Items...
  </LinearLayout>
</ScrollViewer>
```

Ein Panel wird in ein Container umgesetzt (Listing 5.14), in welchem ein optionales Label und Kind-Elemente eingefügt werden. Dieser Container ordnet die Elemente ebenfalls vertikal untereinander gestapelt an. Dies ist dadurch begründet, dass Benutzer von Mobilgeräten durch die kleineren Bildschirme daran gewöhnt sind, den Inhalt des Fensters vertikal durchzuscrollen. Zudem kann nicht garantiert werden, dass die Elemente genug Platz finden, wenn sie horizontal angeordnet wären. Aus demselben Grund werden auch Labels von Eingabe-Elementen über dem Control angezeigt.

```
panel title attr { items } ↦
```

```
<LinearLayout a:orientation="vertical" ...>
  ...ggf. Label: <TextView a:text="...Label..." />...
  ...untergeordnete Items...
</LinearLayout>
```

Ein Action-Element wird in einen Button übersetzt, der die angegebene Operation am ViewModel beim Klick aufruft. Da es bei der GUI-Programmierung üblich ist, Navigation über Buttons zu realisieren, jedoch bei Android die Navigation über den Back Key natürlicher ist, muss ein Zurück-Button nicht generiert werden. Dies wird in Abschnitt 5.5.2 diskutiert.

```
action label op attr ↦
```

```
<Button bind:onClick="...Operation..." a:text="...Label..." />
```

Ein Table-Element wird in einen Container übersetzt, der einen weiteren Container und die eigentliche Liste enthält. In dem untergeordneten Container befinden sich ein Label und Knöpfe zum Bearbeiten der

Liste. Das Control `ExpandedListView` stellt eine nicht-scrollbare Liste dar, die alle Elemente anzeigt und muss verwendet werden, weil in Android scrollbare Listen nicht innerhalb von scrollbaren Containern verwendet werden dürfen, wie etwa dem Wurzelelement `ScrollViewer`.

`table label bind attr` $\mapsto$

```
<LinearLayout a:orientation="vertical" ...>
  <LinearLayout a:orientation="horizontal" ...>
    <TextView a:text="...Bindung..._label" ... />
    <TextView bind:text="title...Bindung..." ... />
    ...ggf. Knöpfe zum Bearbeiten:
       <Button bind:onClick="add...Bindung..." ... />
       <Button bind:onClick="remove...Bindung..." ... />
    ...
  </LinearLayout>
  <c.ExpandedListView bind:source="...Bindung..."
   ...Angabe, ob einfache oder komplexe Einträge angezeigt werden... />
</LinearLayout>
```

Bei dem Table-Element ergibt sich eine plattformspezifische Einschränkung: Da sich auf einem Smartphone mit kleinem Bildschrim keine Tabelle mit vielen Spalten anzeigen lässt, werden Einträge der Tabelle dargestellt, indem maximal drei Eigenschaften angezeigt werden. Die Darstellung ist von Einträgen in Mail-Programmen für Smartphones inspiriert und in Abbildung 5.1 dargestellt (man beachte das rechtsbündige Datum im Mailprogramm). Diese drei Eigenschaften werden folgendermaßen angezeigt: Die erste linksbündig in der ersten Reihe, die nächste linksbündig in der zweiten Reihe und die dritte ebenfalls in der zweiten, aber rechtsbündig. Es kann die Android-interne Verkürzung verwendet werden, die ein gewisses Suffix eines Textes mit „...“ ersetzt.

Ein Edit-Element wird in einen Container übersetzt, der ein Label und ein Control zum Bearbeiten enthält. Dazu wird anhand des Property-Typen unterschieden, welches Control verwendet wird. Dies ist in Tabelle 5.5 dargestellt. Die Idee ist, wie bei den HTML-Controls, eine Check Box für boolsche Werte, eine Auswahl für Modell-Typen und Text-Felder verschiedener Ausprägung für andere Typen zu verwenden.

`edit label bind attr` $\mapsto$

```
<LinearLayout a:orientation="vertical" ...>
  ...ggf. Label: <TextView a:text="..." />...
  ...Eingabe-Control, das den Bindungstyp unterstützt...
</LinearLayout>
```

Typ bzw. Typen	Android-Control
`boolean`, Boolean	`<CheckBox bind:checked="${...Property-Pfad...}" ... />`
`int`, Integer	`<EditText a:type="number|numberSigned" bind:text="${...Property-Pfad...}" ... />`
`double`, Double	`<EditText a:type="number|numberSigned|numberDecimal" ... />`
String	`<EditText a:type="text" ... />`
Password	`<EditText a:type="textPassword" ... />`
Date	`<EditText a:type="datetime" ... />`
Email	`<EditText a:type="textEmailAddress" ... />`
Phone	`<EditText a:type="phone" ... />`
Text	`<EditText a:type="textMultiLine|textCapSentences|textAutoCorrect" ... />`
$M \in Models$	`<Spinner bind:source="{...Liste von Objekten vom Typ M...}" bind:selectedItem="select...Property-Pfad..."... bind:selectedIndex="indexOf...Property-Pfad..."... />`

Tabelle 5.5: Abbildung der Property-Typen nach Android-Controls

5.5.2 Umgang mit Zurück-Buttons

Bei der Modellierung der Anwendung wird Navigation üblicherweise über Buttons realisiert. Für Android-Benutzer hingegen ist die Navigation zur vorherigen Ansicht über den Back Key natürlicher.

Daher sollte ein Zurück-Button für die Android-GUI nicht generiert werden. Die Information, ob ein Button ein Zurück-Button ist könnte beim Generieren über den Aufbau eines Navigations-Graphen erfolgen. Da dies aber sehr aufwendig ist, wird hier die vereinfachende Konvention getroffen, dass es sich um einen Zurück-Button handelt, wenn das Label mit „Back“ beginnt.

Eine Alternative wäre, den Namen der Navigationsoperation zu verändern: Für das Blog-Beispiel müsste für die Operation `showArticles` eine zweite Operation `backToArticles` zur Verfügung stehen, die beispielsweise in der `Article`-View verwendet werden könnte. Im schlimmsten Fall würde dies aber zu einer Duplizierung der Navigations-Operationen führen, nur um einen Zurück-Button in der GUI zu verbergen. Eine bessere Lösung würde den Button ausblenden, wenn er dasselbe Ziel wie der Back Key hat (z.B. indem dies zur Laufzeit geprüft wird).

Auch Buttons zum Speichern des Zustandes können verborgen werden, da sie unter Android nicht benötigt werden.

5.5.3 Generierung der Logik in Java

Ein Model-Element wird in eine Klasse im Unter-Package `model` übersetzt. Diese ist als serialisierbar gekennzeichnet, um sie mit dem Java-Serialisierungsmechanismus persistieren zu können. Die Properties des Models werden in Felder mit `get`- und `set`-Methoden übersetzt, wobei Properties mit primitivem Typen im Konstruktor auf Initialwerte gesetzt werden.

`model id attr { props } ↦`

```
...
public class ...Model-ID... extends Serializable {
  ...Felder für die Properties...
  public ...Model-ID...() { ...Initialisierung der Properties... }
  ...get- und set-Methoden der Properties...
}
```

ViewModels werden in Klassen im Unter-Package `viewmodel` übersetzt. Sie benutzen das Datenbindungs-Framework Robobinding, besitzen eine dafür benötigte Annotation (`@PresentationModel`) und implementieren ein benötigtes Interface.

`viewmodel id attr { props ops } ↦`

```
...
@PresentationModel
public class ...ViewModel-ID... implements ...Robobinding-Interface... {
  ...Felder für die Properties...
  public ...ViewModel-ID...(...) { ...Initialisierung der Properties... }
  ...get- und set-Methoden der Properties...
  ...Übersetzung der Operationen...
}
```

Hier ist die Implementierung der `set`-Methode interessant. Sie ändert das entsprechende Feld und ruft die Methode `propertyChanged` aus der Basisklasse auf. Dieser Benachrichtigungsmechanismus stammt aus dem Datenbindungs-Framework und ist in ähnlicher Form auch in WPF vorhanden. Bei Änderungen kann die View über neue Werte benachrichtigt werden, die dann abgerufen und angezeigt werden.

`prop id : type = val attr ↦`

```
public ...Typ... get...Property-ID...() { return this....Property-ID...; }
public void set...Property-ID...(...Typ... value) {
  this....Property-ID... = value;
  propertyChanged("...Property-ID...");
}
```

Für Listen-wertige Properties benötigt die View noch die Information, wie viele Elemente enthalten sind und Methoden zum Hinzufügen und Löschen von Elementen.

```
prop id : List<T> = val attr ↦
```

```
...Übersetzung der Property wie oben, aber Folgendes zusätzlich...
public void getTitle...Property-ID...() { return "(" + ...Property-ID....size() + ")"; }
public void add...Property-ID...() {
  add(T.class, ...Property-ID..., ...ggf. Property des Modells...);
}
public void remove...Property-ID...() {
  remove(...Property-ID..., ...ggf. Property des Modells...);
}
```

Eine Operation wird in eine Methode ohne Rückgabewert und Parameter übersetzt. Das ViewModel verfügt über alle Informationen in Form von Properties. Allerdings konnte im Rahmen der Arbeit eine generelle Übersetzung von Operations-Implementierungen nicht fertiggestellt werden. Nur ein Spezialfall von Operationen wird übersetzt: Navigationsoperationen.

```
op showXY ↦
```

```
public void showXY() { util.show(XYView.class); }
```

Der allgemeine Fall wird derzeit so übersetzt, dass eine Benachrichtigung über den Aufruf der Operation erscheint. Näheres zur Übersetzung findet sich im nächsten Abschnitt.

```
op id = impl ↦
```

```
public void ...Operation-ID...() {
  util.alert("Operation ...Operation-ID... was called.")
  ...eigentlich: Übersetzung der Implemetation...
}
```

5.5.4 Android-Funktionsumfang

Im Rahmen dieser Masterarbeit konnte die Xtend-zu-Java-Übersetzung nicht mehr fertiggestellt werden, da sie nicht ausreichend getestet werden konnte. Eine angegebene Implementierung der Operationen wird also vom Android-Generator ignoriert.

Xtend wurde so entwickelt, dass es in Java umgesetzt wird, daher ist die Funktionalität, Xtend in Java zu übersetzen, bereits in der Xtend-Bibliothek verfügbar [Xtend]. Dies umfasst sogar die hier nicht verwendeten Lambda-Ausdrücke und Funktionen höherer Ordnung. Dennoch reicht diese Übersetzung allein nicht aus, da der Umgang mit Properties, lokalen Variablen und Hilfsfunktionsaufrufen dort nicht festgelegt ist.

Diese Übersetzung (Tabelle 5.6) wird auch für die Methoden im View-spezifischen ViewModel verwendet. Die Übersetzung einer Bindung `prop1.prop2` wird in eine `getProp1Prop2`-Methode mit der folgenden Implementierung übersetzt: `return this.getProp1().getProp2();`.

Xtend-Konstrukt	Java-Code	
`var $item = expr`	`T item = expr;`	
`$item = expr`	`item = expr;`	
`$item`	`item`	
`prop1.prop2`	`this.getProp1().getProp2()`	
`prop1.prop2 = expr`	`this.getProp1().setProp2(expr)`	
`prop1 = expr`	`self.setProp1(expr)`	
`prop1`	`self.getProp1()`	
`expr binop expr`	`expr binop expr`	binop ∈ {+, -, *, /, %, ||, &&}
`unaryop expr`	`unaryop expr`	unaryop ∈ {-, !}
`if(cond) block else block2`	`if(cond) block else block2`	

Xtend-Konstrukt	Java-Code
`_func(param1, ...)`	`util.other("func", param1, ...)`
`operation()`	`this.operation();`

`for($item : list) block`	`List<T> coll = this.getList();`
	`for(int i = 0; i < coll.size(); i++) {`
	`  final T item = coll.get(i);`
	`  block;`
	`}`

Tabelle 5.6: Übersetzung des Xtend-Codes in Java

5.5.5 Eigenheiten der Android-Plattform

Neben der Generierung der Models, ViewModels und der View muss bei Android auf die unten stehenden Aspekte geachtet werden. Einige Probleme konnten durch statisch generierten Code der Basisklasse der Activities `ActivityBase` gelöst werden. Die Einzelheiten über die Eigenheiten finden sich in Standardwerken zur Android-Programmierung wie etwa [Ableson2012].

- **Der Navigations-Stack** wird dazu verwendet, mit dem Back Key die vorherige Ansicht wiederherzustellen. Beispielsweise beendet der Back Key direkt nach dem Start der Anwendung diese. Nach einer Navigation wird diese durch den Back-Key rückgängig gemacht. Dies ist bei der Generierung von Views zu beachten, weil Navigation zu anderen Ansichten für gewöhnlich über Buttons funktioniert. Da aber die Navigation zu einem vorherigen Fenster dieses unter Android nicht öffnet, sondern nur das aktuelle schließt, muss hier entsprechende Logik implementiert werden [Ableson2012, S. 547]. Im Rahmen dieser Arbeit wird dies über eine im Abschnitt 5.5.2 getroffene Konvention geregelt.
- **Diverse Persistenzmechanismen** sind verfügbar, wobei der einfachste (die sog. `SharedPreferences`) hier verwendet wurde [Ableson2012, S. 131].
- **Der Lebenszyklus** einer Android-Anwendung ist im Vergleich zu anderen GUI-Anwendungen unintuitiv. Beim Drehen des Bildschirms wird die aktuelle Ansicht beispielsweise neu erstellt [Ableson2012, S. 73]. Außerdem ist es möglich, jederzeit zu einer anderen Anwendung zu wechseln. Weil nicht sichergestellt ist, dass der Benutzer wieder zur Anwendung zurückkehrt, sollten die Daten daher gespeichert werden.
- **Das Sicherheitsmodell** erfordert, dass die Anwendungen alle benötigten Berechtigungen im Manifest auflistet. Hier müssen entsprechende Informationen generiert werden.

5.6 GUI-Anpassung

Eine visuelle Anpassung der Controls ist bei der Entwicklung von GUIs notwendig, wenn der Standard-Stil nicht dem gewünschten Aussehen der zu entwickelnden Anwendung entspricht. Beispielsweise können Eigenschaften von Buttons oder Textfeldern vorgenommen und das Layout von Controls beeinflusst werden.

Eine globale Anpassung aller Controls ist beispielsweise praktisch, wenn die Farbgebung eines bestimmten Unternehmens in der Anwendung eingehalten werden soll. In Beschreibungssprachen wie XAML können Styles und Templates verwendet werden, um Einfluss darauf zu nehmen, wie Steuerelemente aussehen sollen. In JavaFX und HTML hingegen wird CSS zur Anpassung verwendet.

Es stellt sich aber die Frage, wie eine plattformunabhängige GUI-Anpassung aussieht. Auf den ersten Blick könnten einige Elemente wie Farben, Abstände und Schriftarten für mehrere Plattformen ähnlich behandelt werden. Aber auch bei diesen Elementen kann es plattformspezifische Eigenheiten geben:

1. Es stehen nicht auf allen Plattformen dieselben Schriftarten zur Verfügung; das Referenzieren über Namen ist damit unzuverlässig. Nicht alle GUI-Frameworks haben die Möglichkeit, eigene Schriftarten einzubetten.
2. Abstände und Größen werden oft in Einheiten wie Pixel, Punkt oder Zentimeter angegeben. Ein Problem ist, dass nicht alle Plattformen alle Einheiten unterstützen, sodass eine Umrechnung erforderlich sein kann.
3. Farbangaben wie `rgb(20, 30, 150)` sind plattformunabhängig. Einige Zielplattformen erfordern aber ein bestimmtes Format, sodass sie beispielsweise als Hexadezimal-Darstellung `#141E96` umgerechnet werden muss. Bei halbtransparenten Farben muss der Alphakanal berücksichtigt werden, weil beispielsweise `#141E96` als Zahl der vollständig durchsichtigen Angabe `#00141E96` entspricht. Eine Komponenten-Notation wie `argb(128, 0, 255, 255)` sollte ebenfalls zur Verfügung stehen. Die Verwendung von Farbbezeichnern (z.B. `red`) ist ebenfalls möglich, erfordert aber ggf. eine Umrechnung in eine Hexadezimal- oder Komponenten-Darstellung, wenn der Bezeichner nicht auf der Zielplattform unterstützt wird.

Da also nicht alle Zielplattformen abstrahiert werden können, zumal die Menge der Zielplattformen nicht begrenzt sein sollte, ergibt sich, dass die GUI-Anpassung für jede Zielplattform separat vorgenommen werden sollte.

Dies hat den Nachteil, dass eine verhältnismäßig einfache Anpassung, wie ein anwendungsweites Farbschema, für jede Zielplattform umgesetzt werden muss. Das führt zu mehrfachem manuellem Programmieraufwand, sofern der Generator für die Zielplattform keine Kurzschreibweise unterstützt.

Der große Vorteil ist, dass eine sehr genaue Anpassung möglich ist, da man auf alle Eigenschaften der jeweiligen Zielplattform zugreifen kann. Beispielsweise können alle Eigenschaften von Buttons unter Android angepasst werden, inklusive solcher, die es auf anderen Plattformen nicht gibt.

In der Plain-DSL wird diese Anpassung mit dem `transform`-Element umgesetzt: Als Attribut wird dann `style` und die Zielplattform eingetragen `[style, platform="Android"]` (bzw. `platform="HTML"`). Als Ausdruck wird ein String erwartet, der Android-XML (bzw. CSS) ist. Ein Beispiel zeigt Listing 5.15, wobei wieder die mit `'''` abgegrenzten Strings aus Xtend verwendet werden.

Listing 5.15: Plattformspezifische Themes mit CSS und Android-XML in der Plain-DSL

```
transform css for [style, platform="HTML"] {
  '''
  .button { background-color: lightgreen }
  .label { color: black; }
  '''
}
transform androidTheme for [style, platform="Android"] {
  '''
```

```
  <style name="AppTheme" parent="AppBaseTheme">
    <item name="android:windowBackground">#50A050</item>
    <item name="android:colorBackground">#50A050</item>
  </style>
  '''
}
```

5.7 Generator-Anpassung

Für die Plain-DSL ist vorgesehen, auch andere Generatoren als die beiden hier entwickelten verwenden zu können. Dazu wird im `app`-Knoten das Attribut `platforms` verwendet, das eine kommaseparierte Liste von Plattformbezeichnern enthält. Sollen für das Todo-App-Beispiel aus Listing 8.9 die beiden Generatoren `HTMLGenerator` und `AndroidGenerator` verwendet werden, kann dies folgendermaßen formuliert werden:
`app Todo [platforms="HTML, Android"] { ... }`

Sollen die Generatoren angepasst werden, ist dies über Vererbung möglich: Man erstellt eine Unterklasse der entsprechenden Generator-Klasse, die angepasst werden soll.

Beispielsweise soll der Android-Generator so angepasst werden, dass ein Custom Control der Plain-DSL in ein Stadtkarten-Control unter Android umgesetzt wird.
Genauer gesagt soll das View Item der Plain-DSL `custom chart [todo.location]` in das entsprechende Android-XML-Fragment übersetzt werden, beispielsweise in ein hier nicht näher beschriebenes Map Control:
`<c.MapControl c.onLoaded="bindTodoLocation"... />`

Dafür sind folgende Schritte vonnöten:

- Erstelle eine Unterklasse `AndroidWithMapsGenerator` der Klasse `AndroidGenerator<View, ViewItem>` im selben Package wie die Oberklasse.
- Erstelle einen parametrisierten Konstruktor, der den Konstruktor der Basisklasse aufruft.
- Überschreibe die Methode `otherToXml(ViewItem item)` und füge dort den spezifischen Markup-Code ein, der überprüft, ob die Property `type` dem anzupassenden Custom-Control-Typ `chart` entspricht. Wenn dies der Fall ist, füge den Android-XML-Code ein, der für dieses Control generiert werden soll. Ist dies nicht der Fall, rufe die Methode der Basisklasse auf (`super.otherToXml(item)`).
- Erstelle `otherToJava(ViewItem item)` auf ähnliche Weise und füge dort den Code für die benötigten Methoden ein. Die Daten können aus dem ViewModel `getViewModel()` bezogen und verändert werden.
- Verwende in `app` nun den neuen Generator, indem er in `platforms` referenziert wird:
  `app Todo [platforms="HTML, AndroidWithMaps"] { ... }`

Andere Anpassungen erfordern die Einbeziehung des Java-Generators für das ViewModel. Auch für diesen kann eine Unterklasse erstellt werden, die im Konstruktor des abgeleiteten Android-Generators über `javaGenerator` gesetzt werden kann.

5.8 Sprachunabhängigkeit

Wie bereits in Abschnitt 4.3 erwähnt, ist es wünschenswert die Übersetzung der DSL in plattformspezifische GUIs nicht auf die hier entwickelte DSL zu beschränken, sondern einen Rahmen zu geben, sodass auch andere Beschreibungen verwendbar sind.

Dazu führe ich eine Strategie [Gamma1996, S. 315] ein, welche als Zugriffsschicht auf das Modell dient. Diese liefert die erforderlichen Informationen in einer sprachunabhängigen Form an den Generator. Diese

Zugriffsschicht wird als Interface mit Namen `ModelAccessStrategy` umgesetzt und enthält Operationen zur Ermittlung der folgenden Informationen:

- alle Models, ViewModels und Views
- die Start-View und das Model, das den Zustand darstellt
- den Titel und die View Items einer View
- die ID und das Label von View Items und Views
- die Bindung eines View Items
- die Unterscheidung, ob ein View Item ein Panel-, Action-, Edit- oder Table-Element ist
- die View Items eines Panel-Elementes
- weitere Eigenschaften eines View Items, beispielsweise `readonly`, `enabled`, `source`, ...
- Lokalisierungs-Informationen (als String-Abbildungen)
- Style-Informationen (als plattformspezifische Strings)

Der Nachteil einer solchen Indirektionsschicht ist, dass bei der Entwicklung Methoden für den Zugriff auf die Daten implementiert und verwendet werden müssen. Alternativ könnte man direkt auf diese zuzugreifen. Dies führt allerdings zu einer sehr hohen Kopplung zwischen Generatoren und zugrundeliegender DSL, sodass bei Änderungen an einem Teil der andere mit geändert werden muss.

Der Vorteil ist allerdings, dass verschiedene DSLs als Eingabe für den Generator verwendet werden können. Dies ist im Rahmen dieser Arbeit mit zwei verschiedenen DSLs geschehen: der im Abschnitt 1.3 erwähnten Gear-DSL und der in dieser Arbeit entwickelten Plain-DSL. Die DSLs enthalten sehr ähnliche Informationen, welche über das Strategie-Interface vereinheitlicht werden.

Der Typ einer View ist für dieses Interface ein generischer Typparameter, etwa `TView`. Ebenso ist der Typ von View Items `TViewItem`. So kann die Austauschbarkeit des zugrundeliegenden Modells typsicher gestaltet werden: Beispielsweise implementiert die Klasse, die die Daten der Plain-DSL liefert das Interface `ModelAccessStrategy<View, ViewItem>`. Der Typparameter `TView` wird an den konkreten View-Typ der DSL `plain.View` gebunden. Ebenso wird `TViewItem` an `plain.ViewItem` gebunden, den Obertyp aller View Items.

Die vorgestellten Generatoren sind generische Klassen, die dieselben Typparameter besitzen: Der Android-Generator ist die generische Klasse `AndroidGenerator<View, ViewItem>`, der seine Daten aus einem Objekt bezieht, welches das generische Interface `ModelAccessStrategy<View, ViewItem>` implementiert (HTML-Generator analog). Der Android-Generator kennt also keine konkreten DSL-Elemente, sondern greift auf ein Strategie-Objekt zu, das ihm die Informationen liefert. Ein großer Vorteil dieses Vorgehens ist, dass die Anpassung eines Generators sich auf eine konkrete DSL beziehen kann. Die Methoden des Generators erhalten nämlich `TView`- bzw. `TViewItem`-Objekte als Parameter. Die in Abschnitt 5.7 vorgestellte Anpassung bezieht sich beispielsweise auf die Plain-DSL.

Eine andere Möglichkeit, Generatoren unabhängig vom zugrundeliegenden Modell zu entwickeln, ist die Verwendung eines expliziten Domänenmodells, das in [Stahl2014, S. 7] diskutiert wird. Der Aufwand, ein solches Modell zu erstellen ist relativ hoch und ist erst sinnvoll, wenn eine größere Menge an Artefakten das Modell der Anwendung bilden („hoher Fan-In-Faktor“) und die Langlebigkeit der Generatoren sichergestellt werden soll („hohe Nachhaltigkeit“).

5.9 Lokalisierung

Unter Lokalisierung einer Anwendung versteht man das Übersetzen von Strings der Anwendung. So kann die Anwendung für Systeme mit anderen Sprachen in der passenden Sprache dargestellt werden. Die eben genannten Strings sind beispielsweise Labels von Views und View Items sowie die Namen aller Properties und Model-Elemente. Auch der Titel der Anwendung und die Titel der Views können lokalisiert werden, diese werden hier vereinfacht wie Labels behandelt.

Dies wird in der Plain-DSL über die Transform-Elemente formuliert. Für eine Sprache wird eine Abbildung über Schlüssel-Wert-Paare definiert. Diese Werte müssen zur Laufzeit abhängig von der eingestellten Sprache des Benutzers angezeigt werden.

```
transform german for [language=de] {
  #[
    "Add" -> "Hinzufügen",
    "Remove" -> "Entfernen",
    "Todo.title" -> "Titel",
    "Todo.note" -> "Notiz"
  ]
}
transform spanish for [language=es] { ... }
```

Dies wird für den HTML-Generator so realisiert, dass an allen Stellen, an denen Labels oder Property-Namen stehen, der Funktionsaufruf einer Lokalisierungsfunktion steht. Die Funktion erhält als Schlüssel das ursprüngliche Label oder Namen der Property und liefert den Wert, welcher in der Lokalisierungstabelle der aktuellen Sprache angegeben ist. Dazu gibt es zur Laufzeit für jede Sprache eine Lokalisierungstabelle als JavaScript-Objekt.

Unter Android werden alle Strings der Anwendung in einer String-Ressourcen-Datei beschrieben. Jeder Sprache ist eine String-Datei zugeordnet, in welcher sich die übersetzen Strings befinden. Die aktuell verwendete Sprache bestimmt, welche dieser Dateien geladen wird. Unter Android muss also für jede angegebene Sprache die entsprechende Datei generiert werden.

6 Evaluation von DSL und Generatoren

In diesem Abschnitt geht es darum, die entwickelte DSL und die dazugehörigen Generatoren zu evaluieren. Dazu werden die aus den Fallbeispielen generierten Anwendungen aus Abschnitt 4.4 mit ihren Referenzimplementierungen verglichen. Es folgt eine Darstellung des Umfangs der generierten und manuell implementierten Anwendungen.

6.1 Generierte Todo App

In Abbildung 6.1 ist die die generierte Todo-Anwendung aus Abschnitt 4.4.1 für Android zu sehen, daneben die Referenzimplementierung. Die Unterschiede sind die Folgenden:

- Die generierte Anwendung enthält kein Anwendungs-Icon und besitzt dadurch bedingt eine kleinere ActionBar. Es wurde auf Icons gänzlich verzichtet, da hierfür der Generierungsprozess um einen zusätzlichen Schritt erweitert werden müsste. Wie in Abschnitt 5.1.2 erwähnt, kann Xtend nur String-basierte Artefakte generieren. Eine Lösung ist, eine Liste von verwendeten Icons zu erzeugen und diese Icons vor dem Kompilieren in den entsprechenden Ordner zu kopieren. In der Implementierung des Android-Generators sind Teile davon vorhanden aber auskommentiert und ungetestet.
- Der visuelle Stil ist anders eingestellt. Beispielsweise können TextBoxes entweder als Kästen oder nur mit einer einfachen Linie an der Unterseite dargestellt werden.

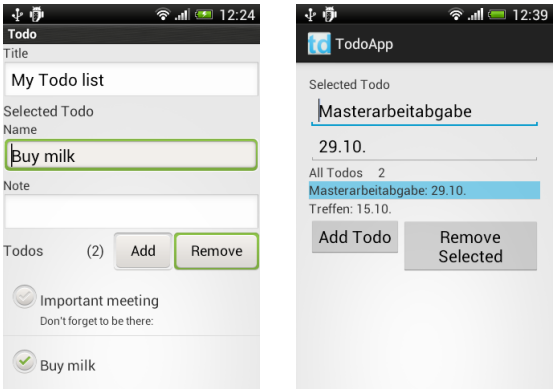


Abbildung 6.1: Die Todo-Anwendung für Android: generiert und Referenzimplementierung

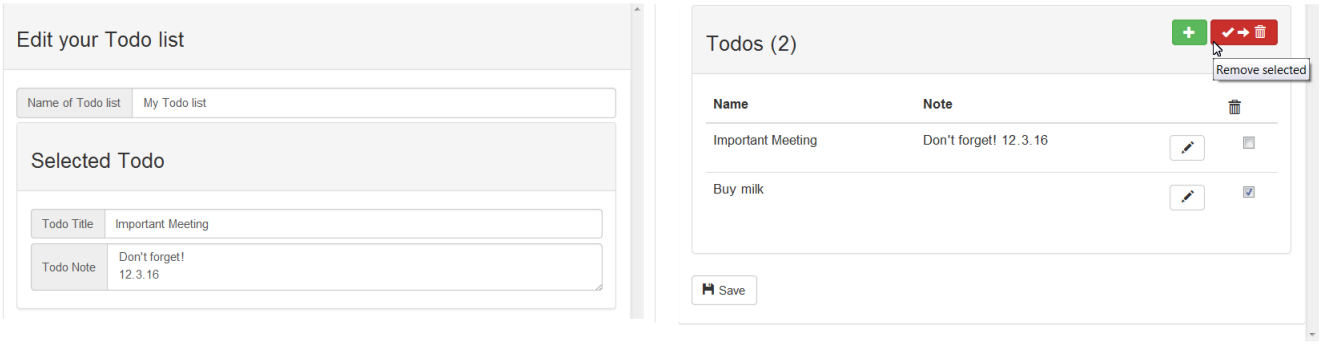


Abbildung 6.2: Die generierte Todo-Anwendung im Browser, Einzel-Seite abschnittsweise nebeneinander dargestellt

- In der Referenzimplementierung existieren keine Labels für die TextBoxes. Anstelle dieser wird der sogenannte Platzhaltertext verwendet. Dieser wird angezeigt, wenn die TextBox leer ist.
- Der Listen-Titel wurde in der Referenzimplementierung ausgelassen, weil er unwesentlich zur Funktionalität der Anwendung beiträgt.
- Das Löschen von Einträgen wird in der Referenzimplementierung so realisiert, dass der ausgewählte Eintrag entfernt werden kann. In der generierten Anwendung hingegen müssen die zu löschenden Einträge erst angewählt und dann über einen Button entfernt werden. So soll vermieden werden, dass ein Eintrag versehentlich gelöscht wird. Durch die zwei Schritte ist kein Bestätigungs-Dialog erforderlich.
- Die Einträge enthalten in beiden Varianten zwei Informationen, den Namen und die Notiz. In der Referenzimplementierung sind diese hintereinander angezeigt, was platzsparender, aber dafür schwerer anzuwählen ist.

Die generierte Web-Anwendung ist in Abbildung 6.2 zu sehen und entspricht der generierten Android-Anwendung mit Ausnahme des Buttons zum Speichern und einigen visuellen Unterschieden. Bei Browser-Anwendungen muss dies explizit getan werden, wohingegen es bei Android-Anwendungen üblich ist, den Zustand sogar beim Anwendungswechsel zu sichern.

6.2 Generierte Blog App

In Abbildung 6.3 ist die die generierte Blog-Anwendung aus Abschnitt 4.4.2 für Android zu sehen, daneben die Referenzimplementierung. Die Unterschiede sind die Folgenden:

- Der visuellen Stil ist wie zuvor anders eingestellt. Dies umfasst in dem gezeigten Beispiel auch die sogenannten Spinner-Controls, die hier zur Auswahl des Autors und der Kategorie verwendet werden.
- Das Label für den Titel des Artikels wird neben der TextBox dargestellt. Dies diente in der Referenzimplementierung als Test. Da nicht bekannt ist, wie breit dieses Label ist, kann nicht sichergestellt werden, dass die TextBox vollständig angezeigt werden kann. Daher wird in der generierten Anwendung das Label stets über dem Eingabe-Control angezeigt. In der generierten Artikel-Ansicht wird dies auch für CheckBoxes eingehalten. Dies führt zu einem konsistenten Umgang mit Labels. Das Auge des Benutzers muss in der generierten Ansicht nur den linken Rand scannen, um das richtige Label finden. In der Referenzimplementierung ist dies bei der CheckBox nicht der Fall.

Die generierte Web-Anwendung ist in Abbildung 6.4 zu sehen und entspricht in großen Teilen der generierten Android-Anwendung. Abbildung 6.5 zeigt die Blog-App auf einem Android-Tablet.

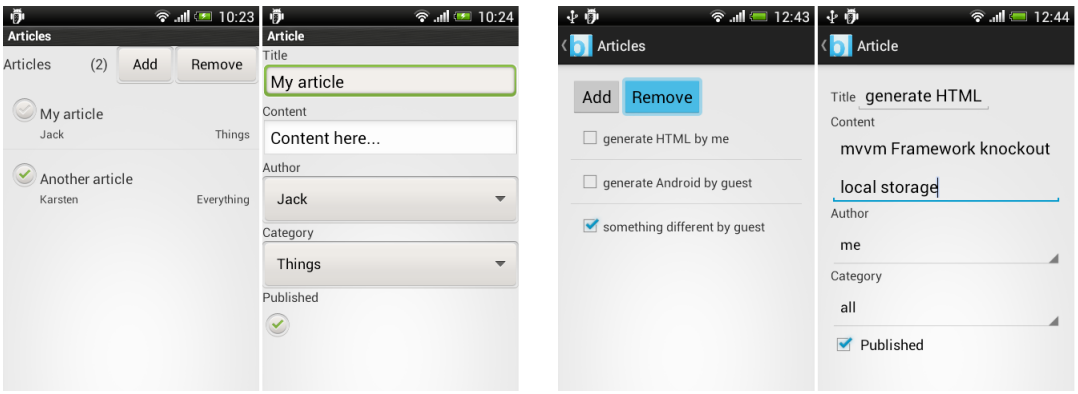


Abbildung 6.3: Die Blog-Anwendung für Android: generiert und Referenzimplementierung

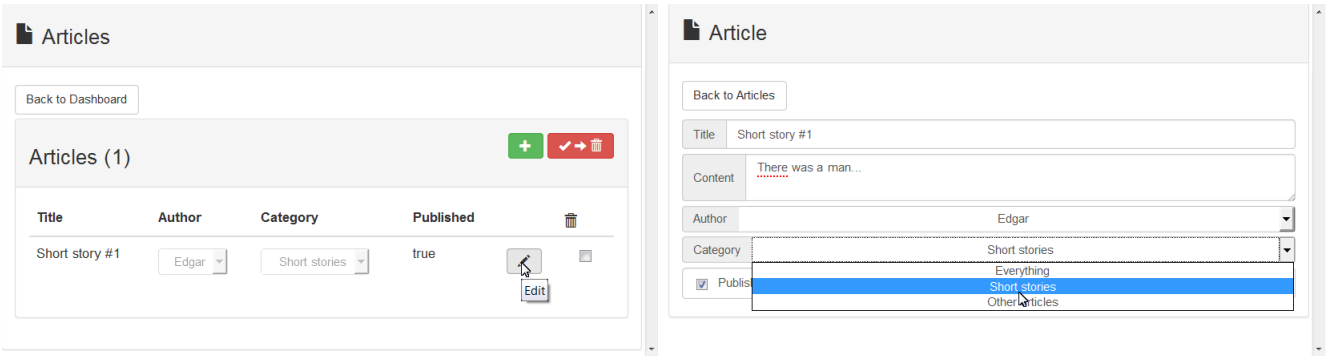


Abbildung 6.4: Die generierte Blog-Anwendung im Browser mit zwei dargestellten Ansichten

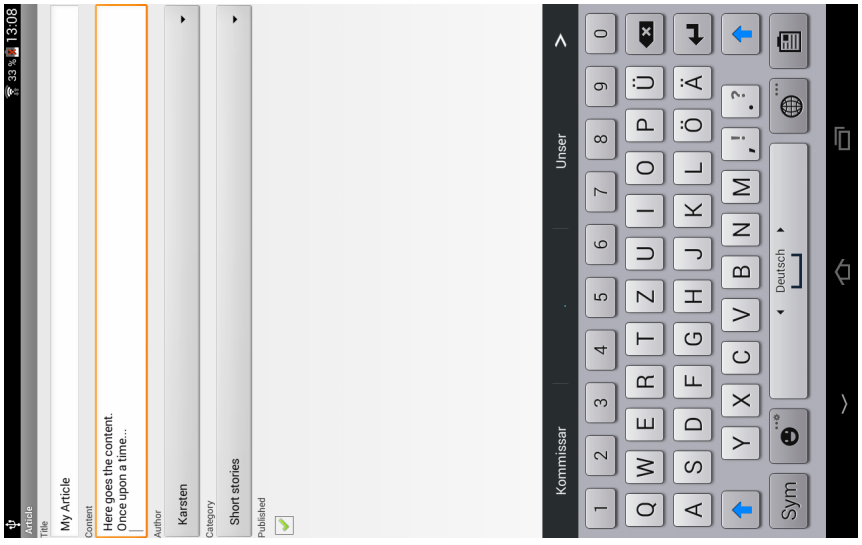


Abbildung 6.5: Die generierte Blog-Anwendung auf einem Android-Tablet

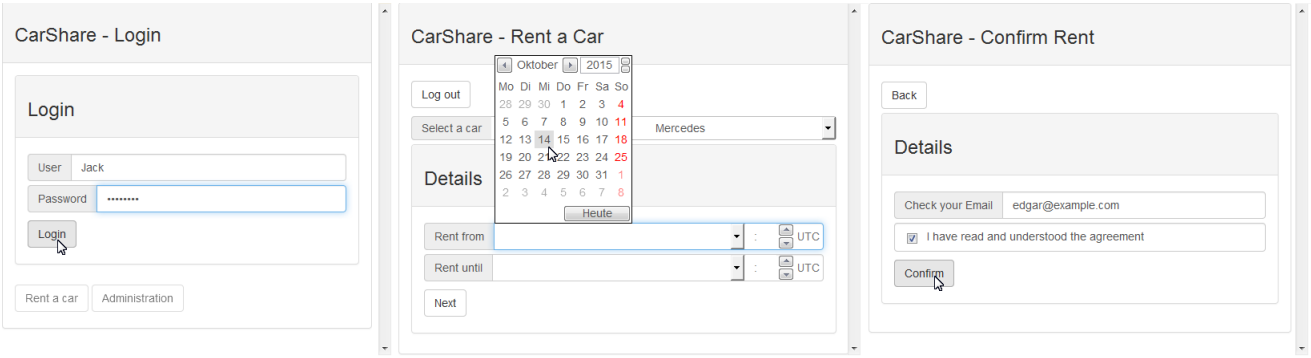


Abbildung 6.6: Die Car-Share-Anwendung im Browser mit drei dargestellten Ansichten

6.3 Generierte Car Share App

Die Car Share App aus Abschnitt 4.4.3 stützt sich auf die implementierten Operationen im ViewModel, um einen Navigationsfluss zu ermöglichen.

Es existiert keine Referenzimplementierung für die Car Share App, weil diese sehr umfangreich wäre. Denn der Umfang der Android-Referenzimplementierung der Blog-App ist in Tabelle 6.1 dargestellt und sehr hoch. Die Car Share App hätte nicht nur einen größeren Umfang, sondern auch viel manuell zu programmierende Anwendungslogik in Java.

6.4 Umfang der generierten Anwendungen

Um einen Überblick über den Umfang der generierten Dateien zu geben, wird hier kurz in Tabelle 6.1 dargestellt, welche Dateien generiert werden. Um abzuschätzen, wie groß die Eingabe-Datei und Ausgabe-Dateien sind, werden pro Datei alle Zeichen gezählt, die kein White-Space sind. Zu White Space zählen Leerzeichen, Umbrüche und Tabulatorzeichen. Alternativen wären die Metriken Lines Of Code oder Non Commenting Source Statements, die jedoch entweder formatierungsabhängig oder nur für Java- und JavaScript-Dateien anwendbar sind. Nicht gemessen würde also die GUI-Beschreibung, die den eigentlich Fokus erhalten sollte. In der Tabelle werden generierte statische bzw. gemeinsame Dateien nicht mitgezählt. Diese Dateien ließen sich im Fall von Android in eine eigene Programmbibliothek verpacken; im Fall von HTML sind dies eine statische JavaScript und eine CSS-Datei.

Umfang/Fallbeispiel	Todo App	Blog App	Car Share App
Plain-DSL-Dokument:	1: 369	1: 1775	1: 2979
Android-Referenzimplementierung:			
Java	5: 5225	19: 14424	–
Layout-XML	1: 2383	7: 10205	–
Ressourcen-XML	6: 2309	4: 5167	–
Android-Generat:			
Java	6: 7004	20: 26359	–
Layout-XML	1: 2579	7: 10498	–
Ressourcen-XML	3: 1874	3: 4742	–
HTML-Generat:			
HTML	1: 3686	1: 11756	1: 10783
JavaScript	3: 1678	3: 5123	3: 6361

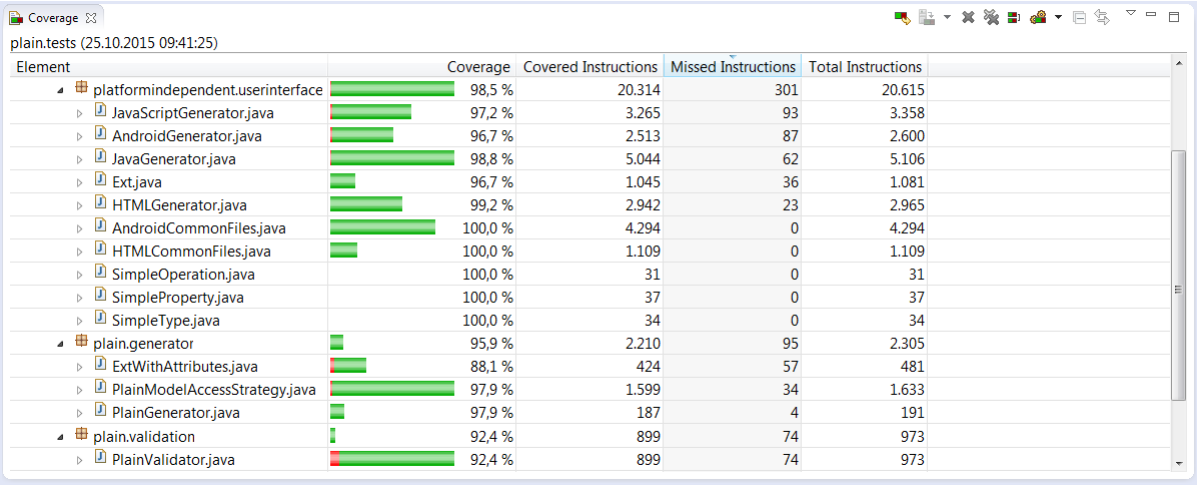


Abbildung 6.7: Darstellung der Testabdeckung in der Eclipse IDE

Umfang/Fallbeispiel	Todo App	Blog App	Car Share App

Tabelle 6.1: Umfang der Eingabe-DSL und generierten Dateien, angegeben als „Dateienanzahl: deren Inhalt als Nicht-White-Space-Zeichen-Anzahl“

Aus diesen Werten wird bereits deutlich, wie umfangreich eine vollständige HTML- und Android-Anwendung ist. Der manuelle Implementierungsaufwand ist damit sehr hoch, auch wenn Tools einen Teil dieses Codes generieren können. Dem gegenüber steht das DSL-Dokument, in welchem die Anwendung kurz beschrieben steht.

6.5 Testabdeckung

Generatoren sollten gründlich getestet werden, wie in Abschnitt 2.5 erläutert. Eine Kennzahl für die Gründlichkeit von Tests ist die sogenannte Testabdeckung (Coverage). Dabei wird gezählt, welche Instruktionen nach dem Durchlauf aller Tests ausgeführt wurden und diese Zahl in Relation mit allen Instruktionen des Programms gesetzt. Es ergibt sich eine Prozentzahl, die angibt, wie viel Code während des Tests ausgeführt wurde. Dies wird als getesteter Code angenommen.

Diese Zahl allein sagt allerdings nichts über die Qualität der Unit-Tests aus, nur darüber, welchen Anteil des Codes die Tests aufrufen. In den Unit-Tests muss immer noch das Ergebnis von getesteter Funktionalität auf seine Richtigkeit hin überprüft werden.

Die Darstellung der Testabdeckungsanalyse für die Generatoren zeigt Abbildung 6.7. Das verwendete Tool ist das Plugin EclEmma für die Eclipse IDE [EclEmma].

Die Darstellung zeigt die Analyse auf Java-Dateien, obwohl Xtend zur Implementierung verwendet wurde. Da aber Xtend in Java übersetzt wird, können auf diesen Dateien Analysen angewendet werden, wobei das Ergebnis auf Xtend übertragbar ist.

Die Generator-Klassen für die Android-Plattform heißen `AndroidGenerator` und `JavaGenerator`; für das Web `HTMLGenerator` und `JavaScriptGenerator`. Die `Ext`-Klassen enthalten die gemeinsame Funktionalität für die Generatoren oder Hilfsmethoden. Alle Klassen, die dem Package `plain` untergeordnet sind, beziehen sich auf die konkreten Klassen der Plain-DSL, alle anderen verwenden das Strategie-Interface, das in Abschnitt 5.8 diskutiert wurde.

7 Schluss

In diesem Abschnitt geht es darum, die gesammelten Erkenntnisse und offenen Punkte darzustellen. Dazu wird resümiert, welchen Impact Plattformunabhängigkeit auf eine DSL hat und wie der entwickelte DSL-Ansatz zu bewerten ist.

7.1 Impact von Plattformunabhängigkeit

Es ist nicht möglich, eine DSL zu erstellen, die gänzlich plattformunabhängig ist, da man nicht alle Plattformen mit ihren Eigenschaften überblicken kann. Daher muss eine Einschränkung der Zielplattformen vorgenommen werden. Ich habe Web- und Mobil-Plattformen als mögliche Zielplattformen gewählt und diese wiederum mit HTML und Android eingeschränkt und konkretisiert. Gründe für die vorgenommene Wahl wurden in Abschnitt 1.4 diskutiert.

Anschließend ist die Wahl der Domäne der DSL zu treffen: In dieser Arbeit liegt der Fokus auf GUIs. GUIs existieren allerdings immer in einem Kontext. Im Falle des hier betrachteten MVVM-Architekturmusters ist der Kontext eine MVVM-Anwendung. Der Fokus liegt auf den Views dieser Anwendung.

Die offensichtlichste Einschränkung, die durch Plattformunabhängigkeit entsteht, ist der Verzicht auf konkrete Framework-Klassen und -Funktionalitäten. Allerdings würden ohne Grundlage nur sehr generische Controls zur Verfügung stehen, etwa Buttons und TextBoxes. Dieses Vorgehen wird allerdings in anderen Ansätzen gewählt (siehe Abbildung 1.4). Vor allem bleibt die Frage offen, wie aus dieser GUI-Beschreibung eine gesamte Anwendung entstehen soll. Durch Hinzunahme des MVVM-Architekturmusters, und damit auch von ViewModels und Datenbindung, lässt sich der Werkzeugkasten vergrößern: Man benötigt kein großes Spektrum aufwendiger Controls, sondern nur die Möglichkeit, die Bearbeitung einer ViewModel-Property in der View anzugeben. Dies ist die Motivation hinter dem `edit`-Element der Plain-DSL, welches ein großes Spektrum an Controls konkreter Frameworks verbirgt. Der Typ einer Property entscheidet, welches Control für die Bearbeitung gewählt werden soll.

Aber diese Entscheidung muss erstens definiert werden, z.B. so, wie es in Tabelle 5.2 und 5.5 beschrieben wurde. Zweitens ist die Menge der Property-Typen nicht beschränkt und somit der Definitionsbereich dieser Abbildung beliebig groß. Es muss also Einschränkungen geben, damit nur ViewModel-Properties bearbeitet werden können, die einen bearbeitbaren Typen aufweisen.

Die Bearbeitung von primitiven Typen ist in vielen GUI-Frameworks möglich. Für den Fall, dass ein primitiver Typ nicht bearbeitbar ist, lässt sich eine Lösung finden: Angenommen, ein GUI-Framework stellt kein Control für die Bearbeitung von Zeitpunkten (Typ `Date`) dar, dann kann dieser wie ein zusammengesetzter Typ behandelt werden. Der generierte Code für die Zielplattform, die das angenommene GUI-Framework verwendet, könnte für die Bearbeitung eines Zeitpunktes die Bearbeitung von Tag, Monat, Jahr und Uhrzeit

verwenden. Dazu gibt es die Möglichkeit, mehrere Eingabe-Controls nebeneinander anzuzeigen oder über einen Button eine Ansicht (bzw. Popup) zu öffnen, das diese Eingabefelder enthält und die Daten zu einem `Date`-Objekt zusammensetzt. Dieses Objekt kann dann der Property zugewiesen werden.

Für die Bearbeitung von nicht-primitiven Typen (z.B. Model-Typen) sind mehrere Vorgehensweisen möglich: Die oben illustrierte Bearbeitung eines Zeitpunktes ließe sich auf auf einen Model-Typ wie Adresse anwenden; dazu würden TextBoxes für die Bearbeitung der Bestandteile einer Adresse verwendet. In der Plain-DSL wurde eine andere Möglichkeit gewählt: Um eine Property zu bearbeiten, deren Typ ein Model-Typ ist, wähle ein Objekt aus einer Liste der möglichen Objekte aus.

Ein weiterer wichtiger Punkt ist die Anpassung von plattformunabhängigen GUIs: Wie in Abschnitt 5.6 erwähnt ist eine Erweiterung oder Anpassung, die nicht bei der Implementierung des Generators bedacht wurde, stets für jede Zielplattformen einzeln durchzuführen.

Plattformunabhängigkeit nachträglich in eine GUI-DSL zu integrieren, ist nur möglich, wenn die DSL so entworfen wurde, dass keine spezifischen Framework-Details angenommen wurden. Das heißt, dass die DSL abstrakte Elemente besitzt (wie etwa `edit`), sodass die DSL auch auf neuen Zielplattformen umgesetzt werden kann.

Zusammenfassend ist der Impact von Plattformunabhängigkeit auf eine DSL gering, wenn folgende Eigenschaften zutreffen:

- **Unbeeinflusst von spezifischen Framework-Details:** Dazu gehört es, Framework-Klassen nicht zu referenzieren und keine Eigenschaften dieser Klassen zu benutzen. Zudem ist auf spezifische Gerätefeatures wie GPS zu verzichten.
- **Geeignete zugrundeliegende Architektur:** Eine DSL, die nur wenige Controls anbietet und mit Callbacks arbeitet, mag plattformunabhängig aber ausdrucksschwach sein. Wird Datenbindung angenommen, ist die DSL ausdrucksstark, erfordert aber den folgenden Punkt:
- **Möglichkeit der Abbildung von Typen zu Eingabe-Controls:** Eine Typinformation einer zu bearbeitenden Property ist eine Information, die bei der GUI-Generierung benötigt wird. Eventuell muss die Datenbindung an Properties bestimmter Typen verhindert werden (indem das Modell dann ungültig ist), wenn diese nicht mit den oben genannten Mitteln umsetzbar ist.

7.2 Ausgelassene Aspekte

Auch wenn es vorteilhaft ist, Software modellgetrieben zu entwickeln, ist damit oft nur die Neuentwicklung abgedeckt. Die Frage nach der Neu- oder Weiterentwicklung von Software, die nicht modellgetrieben entwickelt wurde, wird in den betrachteten Arbeiten in Abschnitt 1.6.1 meist nicht adressiert. Auch diese Arbeit geht nicht auf Weiterentwicklung alter Software ein. Stattdessen sei hier die Arbeit von [Bačíková2013a] angeführt, die dieses Thema genauer erörtert: Das dort beschriebene Verfahren erstellt ein Domänenmodell und eine DSL aus einer bestehenden Anwendung. Dazu werden per Reflection die Klassen der Anwendung analysiert, um daraus Typen der Domäne abzuleiten. Aus diesen Typen wird eine DSL für die Domäne erstellt [Bačíková2013b], aus welcher die Anwendung neu erstellt werden kann.

Wie in Abschnitt 5.5.4 erwähnt wurden Invarianten im Rahmen dieser Arbeit nicht umgesetzt, weder in JavaScript für das Web, noch in Java für Android. In JavaScript ist die Validierung über das verwendete JS-Datenbindungsframework knockout.js [KnockoutJS] bereits vorgesehen und kann an eine Property angefügt werden. Unter Android hingegen sind nur einige Controls dafür ausgelegt, Validierungsinformationen anzuzeigen (über ein rotes Ausrufungszeichen in TextBoxes). Das umfasst nicht den Aufruf der Validierungsfunktion und die Interpretation des Ergebnisses (im Gegensatz zu WPF). Da diese unter Android als Methoden im ViewModel stehen, wäre eine Bindung an diese Controls über die GUI-Beschreibung eine sinnvolle und saubere Lösung. Dies würde allerdings eine Anpassung des Android-Datenbindungsframeworks Robobinding [Robobinding] erfordern.

Anstelle der Validierung existiert für die Web-Plattform die Generierung prototypischer ViewModel-Operationen. Damit ist ein Teil der Validierung umsetzbar: Beispielsweise ist die Validierung von Start- und End-Datum für die Ausleihe eines Autos in der Car-Share-App (Listing 8.11) damit umgesetzt (Zeile 33-38 und 86-90). Ein anderes Beispiel ist die Validierung der E-Mail-Adresse (Zeile 43-54 und 96-98).

Eine genaue Evaluation der Cross-Plattform-Frameworks (Abschnitt 1.7) mit dem in dieser Arbeit gesammelten Wissen würde ergeben, inwiefern sie diese Probleme zufriedenstellend lösen. Auf der einen Seite sind die dort erwähnten kommerziellen Werkzeuge eventuell für solche Unternehmen interessant, die Mobil-Anwendungen entwickeln und diese portieren. Auf der anderen Seite ist JavaFX (Abschnitt 3.3) besonders interessant, sobald das Mobile-Portierungs-Projekt [JavaFXMobile] ausgereift ist.

7.3 Fazit

Am Ende dieser Arbeit stehen viele Erkenntnisse, einige die Implementierung betreffend, andere auf konzeptueller Ebene.

Zunächst ist das Vorhaben, eine plattformunabhängige DSL zu schaffen, ein Mammutprojekt, das alleine nicht durchführbar ist. Daher wurde in Abschnitt 4.5 mit Konventionen versucht, diese Komplexität zu handhaben. Es entstand eine DSL mit dazugehörigen Generatoren, mit denen sich bereits fertige Prototypen von einfachen Anwendungen umsetzen ließen. Diese Fallbeispiele decken natürlich nur einen Teil der möglichen Anwendungen auf dem Markt ab, sodass der Fokus ähnlich dem von [Heitkötter2013, S. 8] ist: Anwendungen mit einem starken Fokus auf Daten-Bearbeitung.

Eine Anpassung möglichst vieler Stellen ist für die modellgetriebene Software-Entwicklung sinnvoll, damit das Modell bei Bedarf verfeinert werden kann und nicht auf Zielcode-Ebene gearbeitet werden muss. Ich habe mich dazu auf einige wichtige Stellen konzentriert: Operationen im ViewModel, Lokalisierung und Themes. Weitere Anpassungen können durch Unterklassenbildung der Generatoren umgesetzt werden (siehe Abschnitt 5.7).

Für die Entwicklung des HTML-Generators habe ich zunächst mit den zugrundeliegenden Bibliotheken Bootstrap und knockout.js die Referenzimplementierungen der Fallbeispiele Todo-App und Blog-App erstellt. Bei der Implementierung stellte sich heraus, welche Teile sich einfach generieren lassen und für welche sich ein höherer Aufwand ergibt. Es existieren keine Referenzimplementierungen dieser Anwendungen, weil sie in folgendem Entwicklungsprozess entstanden sind:

1. Implementiere das Grundgerüst der Anwendung manuell und teste es.
2. Implementiere den Generator, sodass dieses Grundgerüst generiert wird.
3. Führe den Generator aus und teste das Generat.
4. Implementiere ein Feature der DSL manuell und teste es.
5. Implementiere den Teil des Generators, sodass dieses Feature nun generiert wird.
6. Führe den Generator aus und teste das Generat.
7. Verschiebe benötigte Anwendungslogik für das Feature evtl. in eine Unterklasse, sodass weniger Code generiert werden muss (Refactoring mit anschließendem Test).
8. Für das nächste Feature, gehe zu Schritt 4.

Wenn die Generierung der Dateien für die Zielplattform während der Ausführung der Testfälle passiert, ist dieser Prozess besonders praktisch: Die Tests überprüfen, ob der generierte Code mit dem erwarteten übereinstimmt und der Entwickler prüft die Funktionsweise des Generats manuell. Dies ist erforderlich, wenn die Zielplattform das Web oder Android ist. Im Fall der Web-Anwendung muss ein Browser eingesetzt werden. Dieser sollte Script-Fehler darstellen und über einen JS-Debugger verfügen.

Die Entwicklung des Android-Generators orientierte sich stärker an den Referenzimplementierungen. Zunächst habe ich die Todo-App entwickelt, um die Datenbindungsmöglichkeiten für eine Nicht-Hallo-Welt-Anwendung zu ermitteln. Dabei ist eine Herausforderung der korrekte Umgang mit dem Android-Listen-Control. Der Klick auf einen Eintrag bindet diesen an eine Property, andersherum wird die Liste aktualisiert, wenn der Eintrag verändert wird. Die Blog-App ist dann mit dem daraus gewonnenen Wissen entstanden und hat zu weiteren Herausforderungen wie Navigation geführt. Diese konnten ebenfalls erfolgreich gelöst werden, abgesehen von dem vereinfachten Umgang mit der Zurück-Navigation (Abschnitt 5.5.2).

Zum Testen muss die Android-Anwendung auf dem Mobilgerät gestartet werden. Ein Emulator kam hier nicht zum Einsatz, weil ein solcher langsam läuft und ein Realgerät ein besseres Gefühl der Anwendung vermittelt.

Neben diesen die Implementierung betreffenden Erkenntnissen wurden einige konzeptionelle Ergebnisse am Ende dieser Arbeit bereits in Abschnitt 7.1 erläutert. So bleibt zum Schluss nur noch ein bewertender Rückblick auf die plattformunabhängige Programmierung:

- Der naheliegende Vorteil, nicht an eine konkrete Plattform gebunden zu sein, wird mit erhöhtem Programmieraufwand für die Generatoren aufgewogen.
- Der Aufwand, eine Zielplattform in den Generator zu integrieren, orientiert sich daran, wie viel Funktionalität bzw. wie viele Features auf dieser Plattform implementiert werden. Weitere Anpassung bzw. Funktionalität muss eventuell auf der Ebene der Zielsprachen vorgenommen oder hinzugefügt werden.
- Durch den vorgestellten Entwicklungsprozess des Generators für Grundgerüst und Features lässt sich ermitteln, ob ein Feature umsetzbar ist.
- Vermutlich lassen sich Konventionen, wie die im Abschnitt 4.5 getroffenen, nicht umgehen, da Plattformen vereinheitlicht werden müssen. Dies ist erstens in anderen Ansätzen ebenfalls zu erkennen und lässt sich zweitens auch auf MVVM beziehen:
  - Das Cross-Plattform-Framework Xamarin [Xamarin] basiert auf einer Vereinheitlichung verschiedener Plattformen, die notwendig ist, da ansonsten nur JavaScript als kleinster gemeinsamer Nenner verwendet werden könnte.
  - Auch MVVM ist eine Konvention, die immerhin so populär unter Entwicklern ist, dass MVVM- bzw. Datenbindungs-Frameworks für verschiedene Plattformen existieren.

Literaturverzeichnis

[Abeck2007] Sebastian Abeck, Philip Hoyer, Stefan Link, Thomas Schuster:
Modellgetriebene Entwicklung von grafischen Benutzerschnittstellen. Institut für Telematik, Fakultät für Informatik, Universität Karlsruhe. 2007.

[Abdali2015] Abdelmounaïm Abdali, Mohamed Lachgar:
Modeling And Generating The User Interface Of mobile Devices And Web Development With DSL. *Journal Of Theoretical and Applied Information Technology 10th February 2015. Vol.72 No.1*. S. 124ff. 2015.

[Ableson2012] W. Frank Ableson, Robi Sen, Chris King, C. Enrique Ortiz:
Android in Action - Third Edition. Manning. 2012.

[Achilleos2011] Achilleas Achilleos, Georgia M. Kapitsaki, George A. Papadopoulos:
A Model-Driven Framework for Developing Web Service Oriented Applications. Department of Computer Science, University of Cyprus. 2011.

[Bačíková2013a] Michaela Bačíková, Jaroslav Porubän:
UI Creation Patterns (using iTasks for DSL → GUI transformation). Technical University of Košice, Slovakia. 2013.

[Bačíková2013b] Michaela Bačíková, Jaroslav Porubän, Dominik Lakatoš:
Defining Domain Language of Graphical User Interfaces. *2nd Symposium on Languages, Applications and Technologies (SLATE'13)*. Dagstuhl Publishing. 2013.

[Botturi2012] Giulio Botturi, Davide Quaglia:
Multi-Platform Design of Smartphone Applications. Research report. Department of Informatics, University of Verona. 2012.

[Brown2012] Pete Brown:
Silverlight 5 in Action. Manning. 2012.

[Buschmann1996] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal:
Pattern-oriented Software Architecture: A System of Patterns. John Wiley & Sons. 1996.

[Clark2011] Tony Clark, Dean Kramer, Samia Oussena:
Platform Independent, Higher-Order, Statically Checked Mobile Applications. *International Journal Of Design, Analysis And Tools For Circuits And Systems, Vol. 2, No. 1, August 2011*. S. 14-29. 2011.

[Deridder2004] Dirk Deridder, Sofie Goderis:
A Declarative DSL Approach to UI Specification - Making UI's Programming Language Independent. Programming Technology Lab, Vrije Universität Brüssel. 2004.

[Eilebrecht2013] Karl Eilebrecht, Gernot Starke:
Patterns kompakt - Entwurfsmuster für effektive Software-Entwicklung. 4. Auflage. Springer. 2013.

[Ghosh2011] Debasish Ghosh:
DSLs in Action. Manning. 2011.

[Fischer2011] Peter Fischer, Peter Hofer:
Lexikon der Informatik. 15. überarbeitete Auflage. Springer. 2011.

[Gamma1996] Erich Gamma, Richard Helm, Ralph E. Johnson, John Vlissides:
Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. 1996.

[Groenewegen2013] Danny M. Groenewegen, Eelco Visser:
Integration of data validation and user interface concerns in a DSL for web applications. *Software & Systems Modeling vol. 12 nr. 1*. S. 35-52. Springer-Verlag. 2013.

[Hanus2009] Michael Hanus, Christof Kluß:
Declarative Programming of User Interfaces. *Proc. of the 11th International Symposium on Practical Aspects of Declarative Languages, PADL 2009*. S. 16-30. Springer. 2009.

[Heitkötter2013] Henning Heitkötter, Tim A. Majchrzak, Herbert Kuchen:
Cross-Platform Model-Driven Development of Mobile Applications with MD 2. *Proceedings of the 28th Annual ACM Symposium on Applied Computing (SAC 2013)*. S. 526-533. 2013.

[LeGoaer2013] Olivier Le Goaer, Sacha Waltham:
Yet another DSL for cross-platforms mobile development. *Proceedings of the First Workshop on the Globalization of Domain Specific Languages (GlobalDSL'13)*. S. 28-33. ACM New York. 2013.

[McConnell2004] Steve McConnell:
Code Complete - Deutsche Ausgabe der Second Edition. Microsoft Press. 2004.

[Ribeiro2014] André Ribeiro, Alberto Rodrigues da Silva:
XIS-Mobile: A DSL for Mobile Applications. *Proceedings of 29th Symposium on Applied Computing (SAC'14)*. S. 1316-1323. ACM New York. 2014.

[Rosén2013] Kristoffer Rosén:
A Domain-Specific Language for Cross-Platform Smartphone Application Development. Masterarbeit. Department of Computer Science, Faculty of Engineering LTH, Lund University. 2013.

[Stahl2007] Thomas Stahl, Markus Völter, Sven Efftinge, Arno Haase:
Modelgetriebene Software-Entwicklung - Techniken, Engineering, Management. dpunkt.verlag. 2007.

[Stahl2014] Thomas Stahl, Christoph Gutmann:
Meta-Architektur: Das richtige Setup für Ihr MDD-Projekt. Objekt Spektrum November/Dezember 2014, Nr. 6. Verlag SIGS DATACOM GmbH. 2014.

Online-Quellen

[AlphaAnywhere] *The Top Platform for Mobile Enterprise App Development - Alpha Software*.
http://www.alphasoftware.com. Abgerufen am 29.06.2015.

[AndroidUIPatterns] *Android UI Patterns - Design Patterns for Responsive Android Design*.
http://www.androiduipatterns.com/2011/11/design-patterns-for-responsive-android.html.
Abgerufen am 31.08.2015.

[Appcelerator] *Appcelerator - Mobile App Development Platform.*
http://www.appcelerator.com. Abgerufen am 29.06.2015.

[Appmethod] *Appmethod - Cross-Platform App Development for Android, iOS, Windows and Mac OS X.*
http://www.appmethod.com/de. Abgerufen am 29.06.2015.

[BMGear] *b+m Informatik AG - Lösungen - Engineering Solutions - b+m gear.*
http://bmiag.de/loesungen/engineering-solutions/b-m-gear. Abgerufen am 11.10.2015.

[Bootstrap] *Components - Bootstrap.*
http://getbootstrap.com/components. Abgerufen am 20.10.2015.

[EclEmma] *EclEmma - Java Code Coverage for Eclipse.*
http://eclemma.org/. Abgerufen am 25.10.2015.

[FsLexYacc] *FsLexYacc - FsLex and FsYacc tools.*
https://github.com/fsprojects/FsLexYacc. Abgerufen am 29.06.2015.

[GTK] *The GTK+ Project.*
http://www.gtk.org. Abgerufen am 3.09.2015.

[GTKSharp] *GtkSharp - Mono.*
http://www.mono-project.com/docs/gui/gtksharp. Abgerufen am 14.08.2015.

[HaXe] *Haxe - The Cross-platform Toolkit.*
http://haxe.org. Abgerufen am 29.06.2015.

[JavaFX] *6 Using FXML to Create a User Interface (Release 8).*
https://docs.oracle.com/javase/8/javafx/get-started-tutorial/fxml_tutorial.htm. Abgerufen am 11.08.2015.

[JavaFXCSS] *Skinning JavaFX Applications with CSS - JavaFX 2 Tutorials and Documentation.*
https://docs.oracle.com/javafx/2/css_tutorial/jfxpub-css_tutorial.htm. Abgerufen am 27.08.2015.

[JavaFXMobile] *Writing JavaFX Applications for Mobile Devices (Java Magazine - March/April 2015, digital-only publication).*
http://www.oraclejavamagazine-digital.com/javamagazine/march_april_2015?pg=66. Abgerufen am 29.08.2015.

[JQuery] *jQuery.*
http://jquery.com. Abgerufen am 1.09.2015.

[KnockoutJS] *Knockout - Introduction.*
http://knockoutjs.com/documentation/introduction.html. Abgerufen am 1.09.2015.

[MarketShareICD] *IDC: Smartphone OS Market Share 2015, 2014, 2013, and 2012.*
http://www.idc.com/prodserv/smartphone-os-market-share.jsp. Abgerufen am 29.07.2015.

[MVVMCross] *MvvmCross – Official MvvmCross news and updates.*
http://mvvmcross.com. Abgerufen am 14.08.2015.

[PhoneGap] *PhoneGap - a free and open source framework to create mobile apps.*
http://phonegap.com. Abgerufen am 29.06.2015.

[Qt] *Qt - the leading independent technology for cross-platform development.*
http://www.qt.io. Abgerufen am 29.06.2015.

[Robobinding] *RoboBinding - A data-binding Presentation Model (MVVM) framework for the Android platform..*
https://github.com/RoboBinding/RoboBinding#-robobinding. Abgerufen am 5.10.2015.

[Sencha] *Design, Develop, and Manage Enterprise Web Applications with Sencha*.
http://www.sencha.com. Abgerufen am 29.06.2015.

[SenchaGTX] *GXT - Java Framework for Building Web Apps Using Google Web Toolkit - Sencha*.
https://www.sencha.com/products/gxt. Abgerufen am 1.10.2015.

[UDK] *UDK Licensing Resources*.
https://www.unrealengine.com/previous-versions/udk-licensing-resources. Abgerufen am 10.08.2015.

[Unity] *Unity - Game Engine - Develop once, publish everywhere*.
https://unity3d.com. Abgerufen am 29.06.2015.

[Vaadin] *Vaadin – User Interface Components for business apps*.
https://vaadin.com/home. Abgerufen am 12.10.2015.

[Xamarin] *Mobile App Development & App Creation Software - Xamarin*.
http://xamarin.com. Abgerufen am 29.06.2015.

[XamarinCustomers] *Xamarin customers are building amazing native apps with Xamarin products - Xamarin*.
http://xamarin.com/customers. Abgerufen am 30.07.2015.

[Rails] *Ruby on Rails*.
http://rubyonrails.org/. Abgerufen am 18.10.2015.

[Xtend] *Xtend - Modernized Java*.
http://www.eclipse.org/xtend. Abgerufen am 17.10.2015.

[XtendTemplates] *Xtend - Expressions*.
http://www.eclipse.org/xtend/documentation/203_xtend_expressions.html#templates. Abgerufen am 22.10.2015.

[Xtext] *Xtext - Language Development Made Easy*.
http://eclipse.org/Xtext. Abgerufen am 29.06.2015.

[XtextGrammar] *Xtext - The Grammar Language*.
https://www.eclipse.org/Xtext/documentation/301_grammarlanguage.html#parser-rules. Abgerufen am 11.09.2015.

8 Anhang

8.1 XAML-Beispiele

Listing 8.1: Eine einfache GUI mit Buttons und Datenbindung in XAML

```
<Window xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  x:Class="ExampleNamespace.ExampleWindow">
  <StackPanel>
    <Button Click="firstClick">First Button</Button>
    <Button Click="secondClick">Second Button</Button>
    <TextBox Text="{Binding Title}" />
    <TextBox Text="{Binding Title}" />
  </StackPanel>
</Window>
```

Listing 8.2: Code-Behind-Datei des Datenbindungs-Beispiels für WPF in C#

```
using System.Windows;
using System.Windows.Controls;
namespace ExampleNamespace {
  public partial class ExampleWindow : Window {
    public ExampleWindow() {
      InitializeComponent();
      DataContext = this;
    }
    private void firstClick(Object sender, RoutedEventArgs e)
    { MessageBox.Show("You've clicked the first Button."); }
    private void secondClick(Object sender, RoutedEventArgs e)
    { MessageBox.Show("You've clicked the second Button."); }
  }
}
```

Listing 8.3: Templates für CultureInfo-Objekte in XAML

```
<Window xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:sys="clr-namespace:System.Globalization;assembly=mscorlib"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml" Title="CultureInfo-Template">
  <ListBox>
    <ListBox.ItemTemplate>
      <DataTemplate DataType="sys:CultureInfo">
        <StackPanel Orientation="Horizontal">
          <TextBlock Text="{Binding DisplayName}" />
          <TextBlock Text=", Code: " />
          <TextBlock Text="{Binding Name}" />
```

```
          <StackPanel.ToolTip>
            <ListView>
              <ListView.View>
                <GridView>
                  <GridViewColumn Header="English Name"
                    DisplayMemberBinding="{Binding EnglishName}" />
                  <GridViewColumn Header="LCID"
                    DisplayMemberBinding="{Binding LCID}" />
                  <GridViewColumn Header="Currency"
                    DisplayMemberBinding="{Binding NumberFormat.CurrencySymbol}" />
                  <GridViewColumn Header="NaN"
                    DisplayMemberBinding="{Binding NumberFormat.NaNSymbol}" />
                </GridView>
              </ListView.View>
              <ListViewItem Content="{Binding}" />
            </ListView>
          </StackPanel.ToolTip>
        </StackPanel>
      </DataTemplate>
    </ListBox.ItemTemplate>
    <x:Static Member="sys:CultureInfo.InvariantCulture" />
    <x:Static Member="sys:CultureInfo.CurrentUICulture" />
  </ListBox>
</Window>
```

8.2 JavaFX-Beispiele

Listing 8.4: Die GUI-Beschreibung der Datenbindungs-Anwendung unter JavaFX in FXML

```
<?import javafx.scene.layout.*?>
<?import javafx.scene.control.*?>
<?import javafx.geometry.*?>
<FlowPane xmlns:fx="http://javafx.com/fxml/1" fx:controller="application.Main"
  orientation="VERTICAL">
  <padding><Insets top="6" left="6" /></padding>
  <Label text="Input" />
  <TextField fx:id="test" />
  <Label text="Output" />
  <Label text="${test.text}" />
  <Label text="Output Length" />
  <Label fx:id="lengthOutput" />
</FlowPane>
```

Listing 8.5: Die Controller-Klasse zur Datenbindungs-Anwendung in JavaFX

```
package application;
import java.net.*;
import java.util.*;
import javafx.application.*;
import javafx.beans.property.*;
import javafx.beans.value.*;
import javafx.fxml.*;
import javafx.stage.*;
import javafx.scene.*;
import javafx.scene.control.*;
public class Main extends Application implements Initializable {
  @FXML TextField input;
  @FXML Label lengthOutput;
  SimpleStringProperty text = new SimpleStringProperty("Test");
  @Override
  public void initialize(URL location, ResourceBundle resources) {
    input.textProperty().bindBidirectional(text);
    lengthOutput.textProperty().bind(SimpleStringProperty.stringExpression(text.length()));
  }
  @Override
  public void start(Stage primaryStage) throws Exception {
```

```
    Scene scene = new Scene(FXMLLoader.load(getClass().getResource("MainPage.fxml")));
    scene.getStylesheets().add(getClass().getResource("application.css").toExternalForm());
    primaryStage.setScene(scene);
    primaryStage.setTitle("Databinding in JavaFX");
    primaryStage.show();
  }
  public static void main(String[] args) { launch(args); }
}
```

8.3 Android-Beispiele

Listing 8.6: Die Beschreibung einer einfachen GUI mit Text und Button unter Android in Android-XML

```
<LinearLayout
  xmlns:a="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  tools:context="app.android.StartActivity"
  a:orientation="vertical"
  a:layout_width="match_parent" a:layout_height="match_parent">
  <TextView a:id="@+id/textview" a:text="Click the button to show current time!"
    a:textSize="22sp" a:layout_width="match_parent" a:layout_height="wrap_content" />
  <Button a:onClick="updateTime" a:text="Click me!" a:textSize="22sp"
    a:layout_width="match_parent" a:layout_height="wrap_content" />
</LinearLayout>
```

Listing 8.7: Die StartActivity-Klasse für die einfache GUI unter Android in Java

```
package app.android;
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.widget.TextView;
import java.util.Date;
public class StartActivity extends Activity {
  private TextView textview;
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_start);
    textview = (TextView)findViewById(R.layout.textview);
  }
  public void updateTime(View view) {
    textview.setText(new Date().toString());
  }
}
```

8.4 HTML-Beispiele

Listing 8.8: Ein Counter in JavaScript

```
<html>
  <head>
    <title>Counter in JavaScript mit Bootstrap</title>
    <link href="http://getbootstrap.com/dist/css/bootstrap.css"
      rel="stylesheet" type="text/css"></link>
    <script src="http://code.jquery.com/jquery.js" type="text/javascript"></script>
    <script src="http://getbootstrap.com/dist/js/bootstrap.js"
      type="text/javascript"></script>
    <script>
      var count = 0;
      function refresh() { $("#counter").html("Counter: " + count); }
      function decr()    { count--; refresh(); }
      function reset()   { count = 0; refresh(); }
```

```
      function incr()    { count++; refresh(); }
    </script>
  </head>
  <body onload="reset()">
    <center>
      <h2 id="counter"></h2>
      <div class="btn-group" role="group">
        <a class="btn btn-default" onclick="decr()">-1</a>
        <a class="btn btn-default" onclick="reset()">=0</a>
        <a class="btn btn-default" onclick="incr()">+1</a>
      </div>
    </center>
  </body>
</html>
```

8.5 Plain-DSL-Beispiele

Listing 8.9: Die Todo-App in der Plain-DSL

```
app Todo {
  model Todo {
    prop name : String = "Important"
    prop note : Text = "Don't forget..."
  }
  model Todos [state] {
    prop title : String = "My Todo list"
    prop todos : List<Todo>
  }
  viewmodel TodosVM uses Todos {
    prop todo : Todo
    op saveState
  }
  view TodoList "Edit your Todo list" uses TodosVM [start] {
    edit "Title for your list" : [title]
    panel "Selected Todo" {
      edit [todo.name]
      edit [todo.note]
    }
    table [todos]
    action [saveState]
  }
}
```

Listing 8.10: Die Blog-App in der Plain-DSL

```
app Blog {
  model Article {
    prop title : String = "Unnamed Article"
    prop content : Text = "Text goes here"
    prop author : Author
    prop published : boolean = false
    prop category : Category
  }
  model Author {
    prop name : String = "Unnamed Author"
    prop password : Password = ""
  }
  model Category {
    prop name : String = "Everything"
    prop parent : Category
  }
  model Blog [state] {
    prop title : String = "My Blog"
    prop articles : List<Article>
    prop authors : List<Author>
    prop categories : List<Category>
```

```
  }
  viewmodel BlogVM uses Blog {
    prop article : Article
    prop author : Author
    prop category : Category
    prop generated : Text = "Not generated"
    op generate = {
      generated = "<h1>" + title + "</h1>"
      for ($a : articles)
        if ($a.published)
          generated = generated + "<h1>" + $a.title + "</h1>" + $a.content
      _alert ("Done!")
    }
    op showAuthors
    op showArticles
    op showCategories
    op showDashboard
    op saveState
  }
  view Dashboard "Dashboard" uses BlogVM [start] {
    edit "Blog title" : [title]
    panel "Navigation" {
      action "Articles" : [showArticles]
      action "Categories" : [showCategories]
      action "Authors" : [showAuthors]
    }
    panel "Actions" {
      action "Save" : [saveState]
      action "Generate Blog" : [generate]
      edit "Preview" : [generated] [readonly]
    }
  }
  view Articles "Articles" uses BlogVM {
    action "Back to Dashboard" : [showDashboard]
    table [articles] [props = "title, author, category, published"]
  }
  view Article "Article" uses BlogVM {
    action "Back to Articles" : [showArticles]
    edit [article.title]
    edit [article.content]
    edit [article.author]
    edit [article.category]
    edit [article.published]
  }
  view Authors "Authors" uses BlogVM {
    action "Back to Dashboard" : [showDashboard]
    table [authors] [props = "name"]
  }
  view Author "Author" uses BlogVM {
    action "Back to Authors" : [showAuthors]
    edit [author.name]
    edit [author.password]
  }
  view Categories "Categories" uses BlogVM {
    action "Back to Dashboard" : [showDashboard]
    table [categories]
  }
  view Category "Category" uses BlogVM {
    action "Back to Categories" : [showCategories]
    edit [category.name]
    edit [category.parent]
  }
}
```

Listing 8.11: Die CarShare-App in der Plain-DSL

```
app CarShare {
  model User {
```

```
    prop name : String = "Nobody"
    prop password : Password = ""
    prop email : EMail = ""
    prop isAdmin : boolean = true
  }
  model Car {
    prop name : String = "Unspecified car"
    prop available : boolean = true
    prop currentlyUsedBy : User
  }
  model Cars [state] {
    prop cars : List<Car>
    prop users : List<User>
  }
  viewmodel LoginAndUserPerspective uses Cars {
    prop loggedInUser : User
    prop userName : String = ""
    prop userPassword : Password = ""
    prop loggedInAsUser : boolean = false
    prop loggedInAsAdmin : boolean = false
    op login = {
      loggedInAsUser = false
      loggedInAsAdmin = false
      for($u : users)
        if($u.name == userName && $u.password == userPassword) {
          loggedInUser = $u
          loggedInAsAdmin = loggedInUser.isAdmin
          loggedInAsUser = true
        }
    }
    prop hasDatesMessage : boolean = false
    op rentNext = {
      hasDatesMessage = _dateBefore(endRent, startRent)
      if (!hasDatesMessage)
        showConfirmRent()
    }
    prop car : Car
    prop startRent : Date
    prop endRent : Date
    prop agreed : boolean = false
    op confirm = {
      var $m = loggedInUser.email
      if(_validMail($m)) {
        var $text = car.name + ": " + startRent + " - " + endRent
        _sendMail("admin@carshare.com", "Request", $text)
        _sendMail($m, "CarShare - Rent request", $text)
        _alert("Your request was sent, you will receive an email shortly.")
        showLogin()
      } else {
        _alert("Check your email address.")
      }
    }
    op showLogin
    op showAdministration
    op showRent
    op showConfirmRent
    op saveState
  }
  viewmodel AdminPerspective uses Cars {
    prop car : Car
    prop user : User
    op showLogin
    op saveState
    op showAdministration
    op resetPassword = {
      user.password = _md5(user.email + user.password)
      _sendMail (user.email, "CarShare - Password reset", "Password reset: " + user.password)
      _alert ("User was reset and notified.")
```

```
    }
  }
  view Login "CarShare - Login" uses LoginAndUserPerspective [start] {
    panel "Login" {
      edit "User" : [userName]
      edit "Password" : [userPassword]
      action [login]
    }
    action "Rent a car" : [showRent] [enabled = loggedInAsUser]
    action "Administration" : [showAdministration] [enabled = loggedInAsAdmin]
  }
  view Rent "CarShare - Rent a Car" uses LoginAndUserPerspective {
    action "Log out" : [showLogin]
    edit "Select a car" : [car] [source = cars]
    panel "Details" [visible = car.available] {
      edit "Rent from" : [startRent]
      edit "Rent until" : [endRent]
      action "Next" : [rentNext]
      panel "Start date must be before end date" [visible = hasDatesMessage] {
      }
    }
  }
  view ConfirmRent "CarShare - Confirm Rent" uses LoginAndUserPerspective {
    action "Back" : [showRent]
    panel "Details" {
      edit "Check your Email" : [loggedInUser.email]
      edit "I have read and understood the agreement" : [agreed]
      action "Confirm" : [confirm] [enabled = agreed]
    }
  }
  view Administration "Car Share - Administration" uses AdminPerspective {
    action "Log out" : [showLogin]
    action [saveState]
    table "All Cars" : [cars] [props = "name, currentlyUsedBy, available"]
    table "Registered users" : [users] [props = "name, email"]
  }
  view Car "Car Share - Edit Car" uses AdminPerspective {
    action "Back to Administration" : [showAdministration]
    edit [car.name]
    edit "Is available" : [car.available]
    edit "Used by" : [car.currentlyUsedBy]
  }
  view User "Car Share - Edit User" uses AdminPerspective {
    action "Back to Administration" : [showAdministration]
    edit [user.name]
    edit [user.email]
    edit [user.isAdmin]
    edit [user.password]
    action [resetPassword]
  }
}
```

Eigenständigkeitserklärung

Ich versichere hiermit, die Masterarbeit im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel benutzt zu haben, insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.
Ich bin außerdem mit der Einstellung meiner Arbeit in die Bibliothek einverstanden.

.....
Datum, Ort, Unterschrift Karsten Pietrzyk