

Polo Territoriale di Como

Scuola di Ingegneria dell'Informazione



Corso di Laurea Magistrale in Ingegneria Informatica

**Cross-Platform Mobile Application Generation with
a Model-Driven Approach Based on IFML and
Cross-Compilation**

Relatore:

Prof. Marco Brambilla

Tesi di Laurea di:

Emilijan Sekulovski 780049

Anno Accademico 2013-2014

Abstract

Today there are so many mobile devices running on different platforms and platform versions, with even bigger number of different displays (differing in screen size, aspect ratio, PPI, resolution and various technology). Therefore, developers need tools that will help them build applications faster while keeping them consistent throughout different devices. The main motivation of this thesis is to completely automate the process of developing cross-platform mobile applications by using the *Interaction Flow Modeling Language* (IFML). With the introduction of IFML, we are moving into the field of *Model-Driven Development* (MDD), where MDD applications are (semi)automatically generated from the models, allowing for more flexibility, faster prototyping, validation in the early phases of a project and shorter time to market. In this thesis we choose a cross-platform mobile development tool, propose a basic mapping between IFML and the chosen tool, develop an application generation prototype, followed by a more extensive elaboration on a concrete example. At the end we give conclusion on the advantages and disadvantages of both the tools and the languages used, along with a future work proposition.

Astratto

Al giorno d'oggi abbiamo accesso a una svariata quantità di dispositivi mobili operanti con diversi sistemi operativi, con diverse versioni degli stessi sistemi operativi e con un numero ancora più svariato di display (differenti tra loro per dimensioni, proporzioni, PPI, risoluzione e tecnologia realizzativa). A questo proposito, i programmatori necessitano di strumenti finalizzati a creare, nel minor tempo possibile, nuove applicazioni supportate da questa varietà di dispositivi. La motivazione principale di questa tesi è quella di automatizzare completamente il processo di sviluppo di tali applicazioni mobili attraverso l'utilizzo del linguaggio IFML (Interaction Flow Modeling Language). Con l'introduzione dell'IFML, ci stiamo muovendo nel campo dello sviluppo MDD (Model-Driven Development), nel quale le applicazioni sono generate (semi)automaticamente partendo da modelli pre-strutturati, consentendo una maggiore flessibilità, una veloce prototipazione, una convalida in fase iniziale e un breve tempo di inserimento sul mercato. Per cui, attraverso questa tesi, partendo da uno strumento di sviluppo cross-platform, proponiamo una mappatura di base tra tale strumento e l'IFML, sviluppiamo un prototipo di generazione dell'applicazione per poi concludere con una elaborazione più dettagliata applicata a un esempio concreto. Il lavoro si conclude con una riflessione sui vantaggi e gli svantaggi degli strumenti e dei linguaggi utilizzati, unitamente a una serie di ipotesi su linee di sviluppo future.

List of Abbreviations

API	Application Programming Interface
BPMN	Business Process Model and Notation
CSS	Cascading Style Sheet
DOM	Document Object Model
DSL	Domain-Specific Language
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
ITPG	IFML to Titanium Project Generator
Java EE	Java Enterprise Edition
JS	JavaScript
M2T	Model to Text
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
MVC	Model View Controller
NFC	Near Field Communication
OMG	Object Management Group
PDA	Personal Digital Assistant
PIM	Platform Independent Model
PX	Pixel
RIA	Rich Internet Application
SDK	Software Development Kit
SME	Small and Medium-sized Enterprises
TSS	Titanium Style Sheet
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Contents

1. Introduction	6
2. Mobile Application Design Principles	9
2.1. Application Structure.....	9
2.2. General Structure.....	10
3. The Interaction Flow Modeling Language (IFML).....	11
3.1. IFML Main Concepts	12
3.2. IFML Example	13
3.3. The AutoMobile Project	13
4. Cross-Platform Mobile Development Tools Overview	15
4.1. Choosing the Right Tool	18
4.2. Why Titanium?.....	21
5. Going from IFML to Titanium	23
5.1. Mapping IFML to Titanium	25
5.1.1. Visual Constraints and Heuristics.....	25
5.1.2. Component Mapping Proposal	26
5.2. Mapping Implementation	30
5.2.1. Constructing the Data Model.....	33
5.2.2. Window Element	34
5.2.3. Form, List and Details Elements	36
5.2.4. Action, View Element, Select and Submit Events	36
6. Case Study: Book Library Example	38
6.1. Application Structure in IFML.....	38
6.2. Example Break Down.....	41
7. Related Work.....	43
8. Conclusion and Future Work.....	48
9. Bibliography	50

Table of Figures

Figure 1 Global population owning a smartphone	7
Figure 2 Mobile devices in GSMArena’s database	7
Figure 3 Common app structure overview	10
Figure 4 IFML email model	13
Figure 5 DeveloperEconomics cross-platform tools research	15
Figure 6 PhoneGap Build cloud platform.....	20
Figure 7 Titanium global architecture	21
Figure 8 IFML to Titanium transformation example	24
Figure 9 IFML to Titanium transformation process	24
Figure 10 Example of vertical and horizontal layout in action.....	25
Figure 11 ITPG process	31
Figure 12 ITPG web interface	32
Figure 13 ITPGData folder structure.....	32
Figure 14 Transformation classes developed for this thesis	33
Figure 15 Titanium project structure	35
Figure 16 IFML model for simple Book Library App	39
Figure 17 Part of the printed log from the ITPG tool	40
Figure 18 Initial MobiA prototype [26].....	43
Figure 19 Collaborative development process [22].....	44
Figure 20 Web-based designer application screenshot [28]	45
Figure 21 Snapshot of the open source IFML modeling tool [7]	46
Figure 22 MD ² Eclipse environment	46

1. Introduction

In 1983, the DynaTAC 8000x was the first mobile phone to be commercially available. From 1983 up until 2014, worldwide mobile phone subscriptions grew from zero to over 7 billion, penetrating 100% of the global population. Today most of these phones are smartphones, and with that they become less of phones, and more of cameras or navigation devices.

This extraordinary change in the way people communicate and access information is happening all the time. Each day mobile devices are used more extensively. It seems like everything is evolving so fast, that is hard to keep up with the ever-changing mobile industry. But if we just take a small step back and try to see the big picture, the only important thing we can see is information. Information is the one thing that matters, because people need to communicate. In order to live, survive and prosper, people must exchange information. The only thing that changes with time is the medium through which people communicate.

The first known symbols with the purpose of communication are cave paintings like the ones in the Caves of Nerja located in Málaga, Spain, dating older than 40,000 years [1]. Later people used all sorts of tools (devices) as a medium to communicate and exchange information, like fire, smoke signals, drums and horns, as well as many other visual, auditory and ancillary methods. Of course all this transformed into today's exciting world of communication and information consumption. Tools have evolved, and instead of choosing whether to use fire or smoke signal, we get to choose *Apple* or *Samsung*, *Tablet* or *Smartphone*, *iOS* or *Android*. Therefore, we need to bring consistent information and allow seamless way of communication throughout all modern mobile devices.

Nonetheless, this devices still stay phones and allow people to call each other, but at the same time they become less of that. Today, millions of apps are available on the *App Store* and *Google Play*, giving us the possibility to learn, communicate, create, entertain and share things as never before. From 2009 to the end of 2013 we see an enormous jump from 5% to 22% of the global population owning a smartphone [2] (shown in Figure 1).

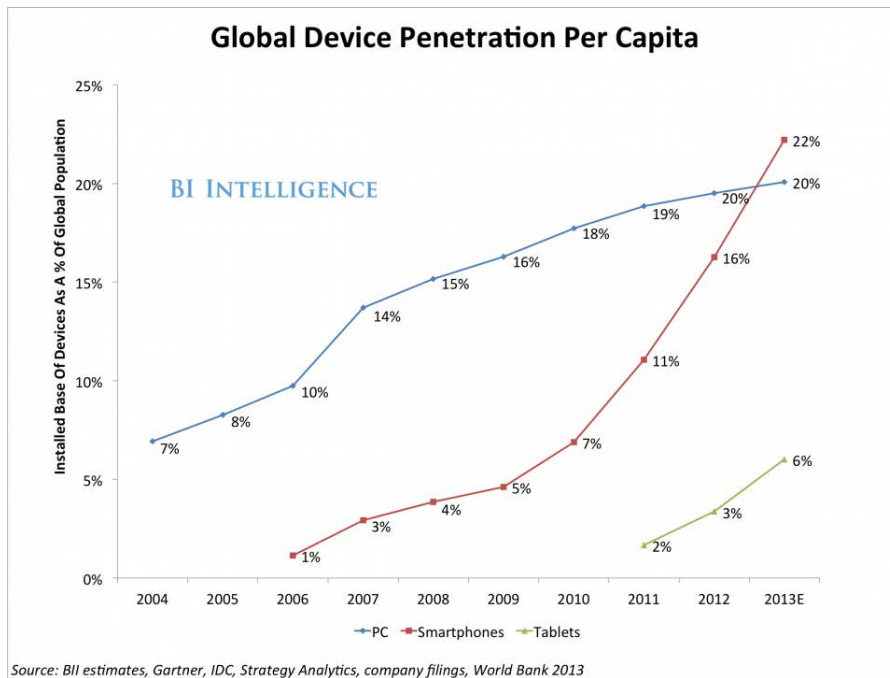


Figure 1 Global population owning a smartphone

In 2013 solely 1 billion smartphones were shipped, an increase of 38.4% comparing to 2012. Tablets are enjoying 51.6% sales growth in 2013 compared to 2012 [3]. As of today, GSMarena.com has 6424 different mobile devices in its database [4] (shown in Figure 2). This number represents one thing: **diversity**. Diversity is what makes our life complicated.

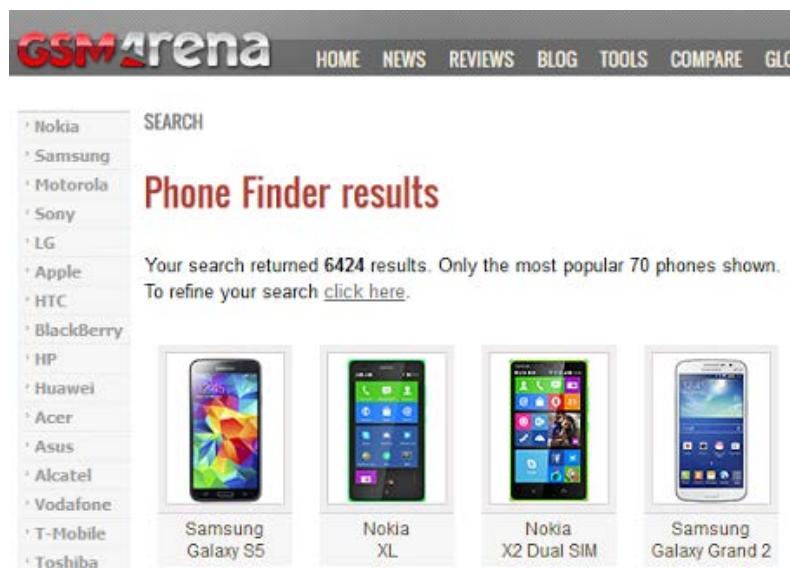


Figure 2 Mobile devices in GSMarena’s database

Today there are so many devices running on different operating systems (platforms) with different platform versions, and on top of that the number of different displays they use (differing in screen size, aspect ratio, PPI, resolution and technology) just makes them more

diverse than ever. Therefore, we can say that we have a vast range of different devices, and with this taken into consideration, we see the problem that lies there: **consistency**.

Keeping app's user interface (UI) and user experience (UX) consistent in a cross-platform environment and throughout different displays out there represents a huge challenge. Fundamentally, we have two words that do not quite fit together: diversity and consistency. The bigger the diversity, the harder to keep the consistency. And what matters the most while keeping this cross-platform consistency are time and cost. How long it will take, and how much it will cost? Every application developer (whether an individual or a company) out there tries to keep these two values as low as possible.

Having that in mind, developers need tools that will help them build apps faster while keeping the apps consistent throughout different devices. Fortunately, there are many innovating tools that are focused on designing cross-platform mobile development patterns and principles. We will compare several tools used for cross-platform mobile development and choose to proceed this thesis research with one of them.

The main motivation of this thesis is to automate the process of developing cross-platform mobile applications even further. To achieve this, the *Interaction Flow Modeling Language* (IFML) is used. A modeling language is used to create models for the system to be developed. It provides a conceptual model consisting of formally defined graphical and textual representations. By introducing IFML, we are moving into the field of *Model-Driven Development* (MDD). Usually, in MDD applications are (semi)automatically generated from the models, allowing for more flexibility, faster prototyping, validation in the early phases of a project and shorter time to market.

A basic mapping proposal between IFML and the chosen tool will be placed, followed by a more extensive elaboration on a concrete example. Conclusion on all the pros and cons of this approach and the developed prototype, along with a future work proposition summarizes this work.

2. Mobile Application Design Principles

Mobile design is different, and not just because of the size of mobile devices. The physicality and specifications of mobile devices convey different design affordances and requirements, notably perceived affordances. Perceived affordances describe the relationships that users perceive within an environment. Mobile design should help users to distinct actions and invite them to use those actions [12].

Because mobile devices are lighter and more portable, it is more convenient to use them. Consequently, people start feeling this unique, emotional connection to them. Today, most of the mobile devices employ touch screens, making users rely on gestures and simple interface elements to interact with them. Their smaller dimensions often impose smaller and simpler content structures. Also, limited bandwidth and connectivity makes mobile devices require optimized designs with less data requirements [13]. Having constant access to mobile devices, people tend to use them more frequently: on the bus, while walking down the street, while watching TV etc. They are often used while doing something else, which means that they are used under difficult viewing conditions and among a variety of distractions.

The areas of the screen the user touches to activate something (hit areas) require adequate space for the user to accurately and confidently use an action. The average fingertip is between one to two centimeters wide, which roughly correlates to somewhere between 44px and 57px on a standard screen and 88px to 114px on a high-density (retina) screen. Nokia, Apple and Microsoft each recommend slightly different sizes to account for the nature of touchscreens. Therefore, there are no hard and fast rules regarding to hit areas and their size. Getting the interactions right on a mobile device is crucial for its usability and functionality. But to create a truly wonderful experience the appearance and the UI of the application needs to inspire, charm and engage users [14].

2.1. Application Structure

Applications come in many varieties that address very different needs. For example:

- Apps such as Calculator, Camera or Games are built around a single focused activity handled from a single screen. It is not very useful to create a Titanium Project Generator for this kind of apps.
- Apps such as Phone, Messages or Gallery whose main purpose is to switch between different tabs (views) without deeper navigation are also not very interesting for us.
- Apps such as Gmail, WhatsApp or Instagram that combine a broader set of data views with deeper navigation are the kind of apps we aim for.

2.2. General Structure

A typical mobile app consists of top level and detail/edit views. If the navigation hierarchy is deep and complex, category views connect top level and detail views. This shown in Figure 3 below [15].

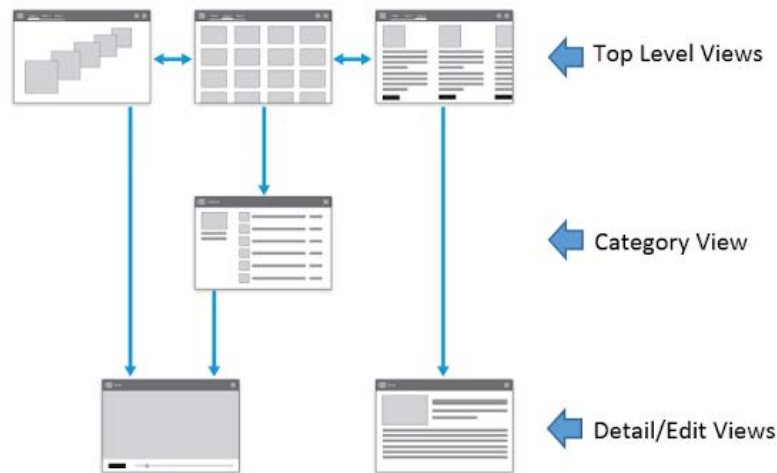


Figure 3 Common app structure overview

- **Top level views** - the top level of the app typically consists of the different views that your app supports. The views either show different representations of the same data or expose an altogether different functional facet of an app. It is practice to navigate between top level views either from the app's menu or by interacting with the current view elements. See Figure 3.
- **Category views** - category views allow to drill deeper into the app's data. It usually offers a grouped and more structured view of the data available in the top level views. See Figure 3.
- **Detail/edit view** - the detail/edit view is where data is consumed or created. This is often the last view in the hierarchy of an app structure. See Figure 3.

As mentioned above in subsection 2.1, apps with broader set of data and deeper navigation are the kind of apps that make sense to be modeled and developed in a cross-platform environment.

3. The Interaction Flow Modeling Language (IFML)

A modeling language can be used to express information, knowledge or systems in a structure that is defined by a consistent set of rules [5]. There are some general-purpose modeling languages, like UML, that are used in any sector or domain. On the other hand, there are also domain-specific languages, like IFML, which are focused on facilitating the definition of a specific software subsystem.

IFML is an OMG standard for expressing the content, user interaction and control behavior of the front-end of software applications. IFML supports the platform independent description of graphical user interfaces for applications accessed or deployed on systems such as desktop computers, laptop computers, PDAs, mobile phones, and tablets. The focus of the description is on the structure and behavior of the application, as perceived by the end user. The description of the structure and behavior of the business and data components of the application is limited to those aspects that have a direct influence on the user's experience.

With respect to the popular Model-View-Controller (MVC) model of an interactive application, the focus of IFML is on the view part. Furthermore, IFML describes how the view references or is depended on the model and control parts of the application.

IFML is a *Platform Independent Model (PIM)-level language* in *Model-driven Architecture (MDA)* parlance, and it perfectly fits into the *Abstract User Interface* level of the *Cameleon Reference Framework*. The *Business Process Model and Notation (BPMN)* language may be used in the context of IFML to provide *Context Independent Models*.

3.1. IFML Main Concepts


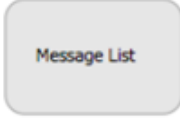




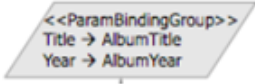
Concept	Meaning	IFML Notation	PSM Example
View Container	An element of the interface that comprises elements displaying content and supporting interaction and/or other ViewContainers.		Web page Window Pane.
View Component	An element of the interface that displays content or accepts input		An HTML list. A JavaScript image gallery. An input form.
Event	An occurrence that affects the state of the application		
Action	A piece of business logic triggered by an event		A database update. The sending of an email.
Navigation Flow	An input-output dependency. The source of the link has some output that is associated with the input of the target of the link		Sending and receiving of parameters in the HTTP request
Data Flow	Data passing between ViewComponents or Action as consequence of a previous user interaction.		
Parameter Binding Group	Set of ParameterBindings associated to an InteractionFlow (being it navigation or data flow)		

Table 1 IFML main concepts and notations

An IFML model consists of one or more top-level view containers. Each view container can be internally structured in a hierarchy of sub-containers. In case of mutually exclusive (XOR) containers, one could be the default container, displayed by default when the parent container is accessed.

A view container can contain view components, which denote the publication of content or interface elements for data entry (e.g., input forms). A view container and a view component can be associated with events, which can represent user's interaction or system-generated occurrences. An event can also cause the triggering of an action, which is executed prior to updating the state of the user interface. The main concepts and notations are listed in Table 1 above.

3.2. IFML Example

To illustrate how the main concepts are used in real case scenarios, an example model of an email system is shown in Figure 4 below.

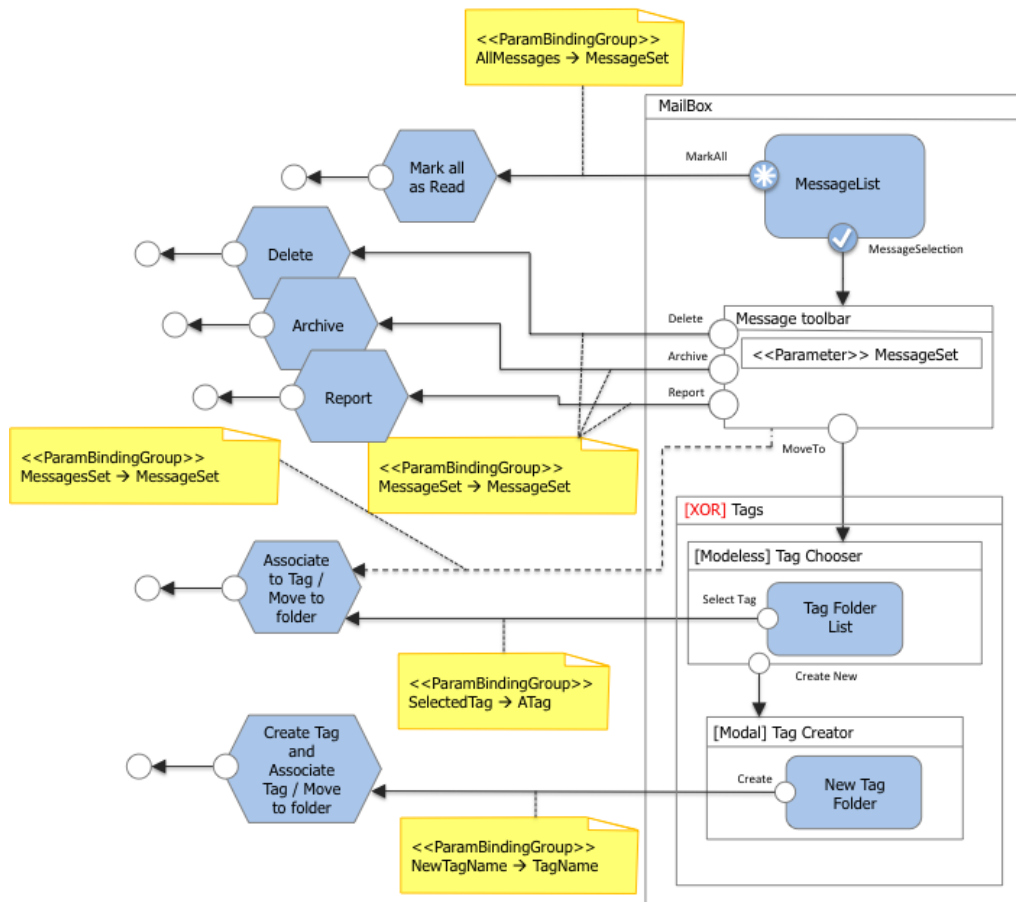


Figure 4 IFML email model

3.3. The AutoMobile Project

AutoMobile [6] is a project that includes the development of a mobile modeling language. This language is an extension of the IFML modeling language tailored especially for mobile applications. AutoMobile exploits the modern paradigm of Model-Driven Engineering and code generation to dramatically simplify multi-device development, reducing substantially cost and development times, so as to increase the profit of small and medium-sized enterprise (SME) solution providers and at the same time reduce the price and total cost of ownership for end-customers.

AutoMobile relies on modeling languages such as IFML and on tools like WebRatio. IFML, as described above, does not cover any mobile specific aspects. AutoMobile brings mobile specific view elements, such as Screen or Message. It also brings mobile specific events, such as *TouchEvent* or *SwipeEvent*. At the moment of writing, AutoMobile successfully completed its first review on October 10th 2014 and is on its way to the second phase, which is defining

the methodology. It will be a textbook with clear methodological guidelines on how to elicit, design, implement and deploy successfully mobile multi-channel and multi-device [7].

Now, we move on to the cross-compilation part of this thesis. We will review, evaluate and choose one out of the four most popular cross-platform mobile development tools on the market.

4. Cross-Platform Mobile Development Tools Overview

Cross-platform mobile development represents a technique used in developing mobile apps for multiple platforms (OS). Writing a single codebase for an app that will be deployed on multiple platforms sounds intriguing. This approach is also very challenging, because of two reasons. First, the most popular mobile platforms like *Android* and *iOS* are structurally very different from each other. This difference goes down to architectural level in some cases. Second, *Google* and *Apple* usually encourage developers to code in contradicting ways, making it difficult to have a single codebase efficient on both platforms. Therefore, developers face major problems when coding apps for multiple platforms.

Today there are many cross-platform development tools to choose from. All of them have their own unique approach. If we look at the research statistics (shown in Figure 5 below) from *DeveloperEconomics* [8] blog, we will see that the top five most used tools are *PhoneGap*, *Appcelerator (Titanium)*, *Adobe AIR*, *Sencha* and *Qt*.

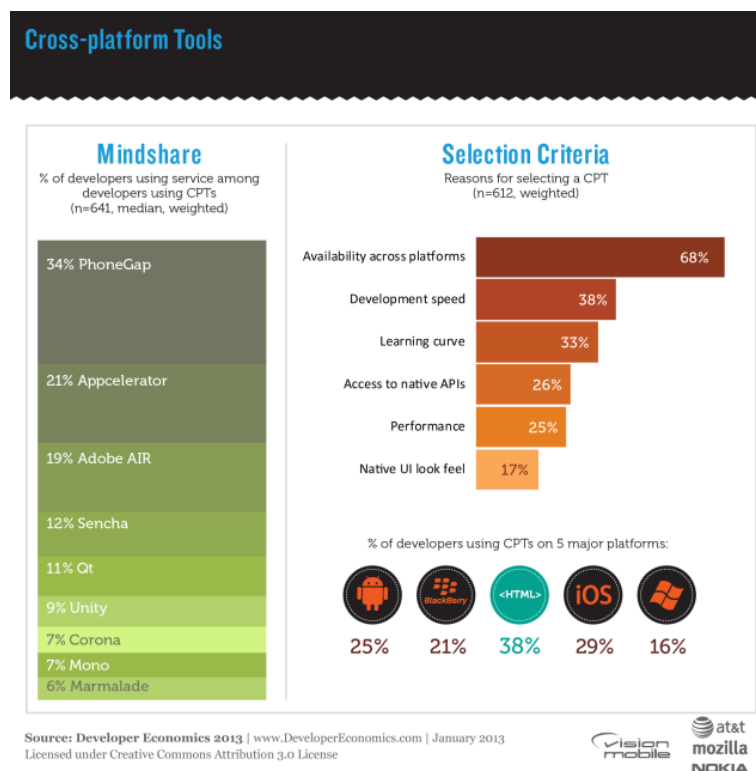


Figure 5 DeveloperEconomics cross-platform tools research

In the following part, we describe the first four most used tools, and we list their pros and cons.

- **PhoneGap** - it uses HTML, JavaScript (JS) and CSS for writing mobile applications, and it runs the apps in a “WebView” inside a native application container on the target platform. It is, conceptually, a web application packaged within a native application container where your JavaScript has access to native device APIs that normal web

applications would not. The name *PhoneGap* is quite possibly one of the more recognizable names in the space of cross-platform mobile development. To summarize, here are the most notable pros and cons of this development tool:

- Pros:
 - i. Use of common web technologies (HTML, JS & CSS) for writing apps.
 - ii. Reduces training time and learning curve for your team, resulting in quick-to-market stance.
 - iii. The apps install just like any other native app, through each platform's respective app store.
 - iv. There are no license costs, it is completely open source and free.
 - v. It has a plugin architecture, allowing access to new native device APIs through extendable modular way.
 - vi. It offers cloud build, no SDK required. You can build your cross-platform app on their online *PhoneGap Build*.
- Cons:
 - i. Getting support for things that do not come right out of the box.
 - ii. Writing custom plugins by yourself in order to utilize additional native APIs.
 - iii. Performance. Because of its "WebView" nested inside a native application container, native UI apps will always outperform hybrid solution like this.
- **Appcelerator Titanium** - this tool provides a unified (across devices) JavaScript API, coupled with native-platform-specific features. Developers write JavaScript and utilize a UI abstraction (the Alloy MVC framework) that results in the use of native UI components, greatly aiding UI performance compared to other hybrid options. Here are the pros and cons of this tool:
 - Pros:
 - i. Uses JS along with native-platform-specific features.
 - ii. Deployment with native UI elements is a performance win.
 - iii. Alloy framework normalizes UI across platforms.
 - iv. Use of JS normalizes code across platforms, allowing you to leverage existing skills on multiple target platforms.
 - v. Provides app analytics and marketplace for 3rd party components.
 - Cons:
 - i. Developers have to manage all targeted platform SDKs locally by themselves. SDK versions & build related issues can be a horrific time sink.
 - ii. Normalizing UI across platform, often means gaining skills that later cannot be transferred outside Titanium.
- **Adobe AIR** - this is a cross-operating-system runtime that lets developers combine HTML, JavaScript, Adobe Flash and Flex technologies, and ActionScript to deploy rich Internet applications (RIAs) on a broad range of devices including desktop computers,

netbooks, tablets, smartphones, and TVs. But, you can only use Flash and ActionScript to write Adobe AIR applications for mobile devices.

- Pros:
 - i. Possibility for creating more animated UI elements, but not with native approach.
 - ii. Developers praise its mature IDE.
- Cons:
 - i. Since Adobe purchased PhoneGap, it is the “elephant in the room”. With other words, it’s not the long term strategy of Adobe for cross-platform mobile development.
- **Sencha** - Sencha Touch is an HTML5 mobile application framework for building web applications that look and feel like native applications. Apps built with Sencha Touch can be used with PhoneGap or Sencha’s native packager.
 - Pros:
 - i. Sancha has built a range of interoperable products to ease the process for developers.
 - ii. It offers an MVC architecture along with a library for UI components.
 - Cons:
 - i. Same performance issues as PhoneGap. Developers should be experienced enough to write efficient JS and DOM structure.
 - ii. It’s community is much smaller than the one of PhoneGap.
 - iii. There is the need to write custom PhoneGap plugins for accessing additional or new native device APIs.

Summing up all these pros and cons, I narrowed my choice to two options:

- PhoneGap
- Appcelerator Titanium

Adobe AIR is the “elephant in the room” between the rests. It uses technologies like Adobe Flash and ActionScript which are being less and less used among developers. Sancha is very similar to PhoneGap, but Sancha’s community being much smaller made the difference between this two. There is much more forum discussion, support, examples and tutorials for PhoneGap than for Sancha.

4.1. Choosing the Right Tool

In the following table (Table 2) we list and compare the different features of PhoneGap and Titanium. With green and red color we mark the strong and weak sides of each tool:

Feature	PhoneGap	Titanium
Supported platforms	iOS, Android, Blackberry, Windows Phone 7, Symbian and Bada.	iOS and Android. Blackberry support is available only to Pro/Enterprise customers (paid).
Development technologies	HTML, CSS and JavaScript. It's possible to use jQuery Mobile and other web frameworks which rely on presence of DOM. This is probably one of the reasons why PhoneGap is so popular – most developers, who are not familiar with mobile development, can use PhoneGap easily if they have some experience in web development.	Maps JavaScript calls into native code at run time. Developers can't use jQuery Mobile or HTML for UI, but Titanium API overall is much closer to native API, comparing to PhoneGap API. That's why developers, who are familiar with native mobile apps development, can start developing using Titanium quickly.
Control over system	PhoneGap supports less features which are specific to only one platform.	Has much deeper integration with each mobile platform, so it allows fine control over the system, depending on the platform on which the app runs.
Code maintenance, code reuse and porting to other platforms	Aims to standardize code across all platforms. Code maintenance for several platforms is easier than with Titanium.	Some APIs are platform-specific. Because of this fine control, sometimes it's more difficult than with PhoneGap to make and support code which runs on all required platforms. Apps must be tested thoroughly on all platforms.
UI (look and feel)	Controls simulate look of native UI, but there's almost no way to get the real native look – it just feels different.	Native look, feel and UI controls (which can be customized, if necessary)
Performance	Performance is limited by HTML and JavaScript. It's noticeably slower than what you can get with native apps.	Performance is similar to what you can get with native apps (it's much smoother than with PhoneGap).
Development environment	Eclipse. Not as well-suited for a specific framework as Titanium Studio. On the other hand, the PhoneGap Build cloud package compiler makes thing a lot easier for the developers.	Have their own IDE based on Eclipse. It's called Titanium Studio. With introduction of the IDE, Titanium development has become much easier, there's integrated debugger and other convenient tools to help developers.

Table 2 Feature comparison between PhoneGap and Titanium [9]

Both of the tools have their strong and weak sides, and because of their weaknesses, apps for iOS and Android are still mostly implemented natively.

Each mobile application has its own specifics, so choice of technologies for its development must be based on project requirements and supported features of each tool.

When it comes to Titanium, there might be more difficulties in maintaining and testing the code compared to PhoneGap, as Titanium projects tend to have more platform-specific parts of code. Applications developed with Titanium make use of platform-specific native user interface elements for user interaction. Thus, with this approach it is possible to create a user interface that closely matches a native one. Judging from *a cross-platform code reuse aspect*, PhoneGap does a better job. It has 95-100% code and resources reused across different platforms, varying 5% only for the potential use/development of additional platform-specific plugins. However, Titanium stands at around 60-90% [10], depending mostly on the amount of platform-specific UI elements used in a project.

In Titanium there are platform-specific resources, like images, scripts and style sheets. Therefore, developers must use a technique called *code branching* for separating the platform-specific parts. Code branching is useful when the code is mostly the same across platforms. Usually, blocks of if-then-else statements are used to separate platform-specific sections of the code. Long blocks of if-then-else code are difficult to read and maintain. Also, excessive branching will slow down the app's execution. When using this technique, developers should group as much code as you can within a branch and defer loading as much as possible to mitigate the performance penalty of branching.

Branching is a result of the different UI elements in each platform. For example, *Titanium.UI.iOS.ANIMATION_CURVE_EASE_IN* defines an iOS-specific animation property. One platform's constants should not be used on another platform, because the code will throw an error. Differences like this are solved through code branching. Its best practice to query the platform value once, store it in a globally accessible variable and use it to branch your platform-specific code sections.

Example:

```
var isAndroid = (Ti.Platform.osname=='android') ? true : false;
if(isAndroid) {
    // do Android specific stuff
} else {
    // do iOS stuff
}
```

So, only one project is developed and maintained, which has all this little platform dependent if-then-else statements to handle the differences between the multiple OS and their presentation layer (native UI elements).

The main disadvantage of the code branching technique is that the user interface of a specific branch cannot be reused. Also the platform-specific features (for example: camera access, location services, local notifications) cannot be reused. These features are platform-specific and the way to access these features varies from platform to platform. This approach would be appropriate for simple applications but for sophisticated applications cross-compilation might be outweighed by native approach.

Finally, it all depends on the project, the way UI should feel and the way a user interacts with it. The more *native feel* is required, the more platform-specific code is needed to be branched. So the code can easily go from 100% cross-platform to 70%.

Looking at the necessary development environment (IDEs, SDKs, Libraries) to work with both tools, PhoneGap has a much faster and simple way of doing that. Developers can use any IDE (Eclipse, Netbeans, Dreamweaver, even Notepad), and when the coding is finished one should just upload the project to *PhoneGap Build* cloud platform. There is an option of connecting a project's GitHub repo directly with the cloud platform. Using this option will update, compile and package the new app version automatically for iOS, Android, Windows, Blackberry, WebOS and Symbian. These are platform-specific packages ready to be downloaded and installed immediately on any device. The Figure 6 below illustrates the cloud build environment for a basic *HelloWorld* app on the cloud platform. Only iOS package was not built because we needed an iOS developer account, which we currently don't have.

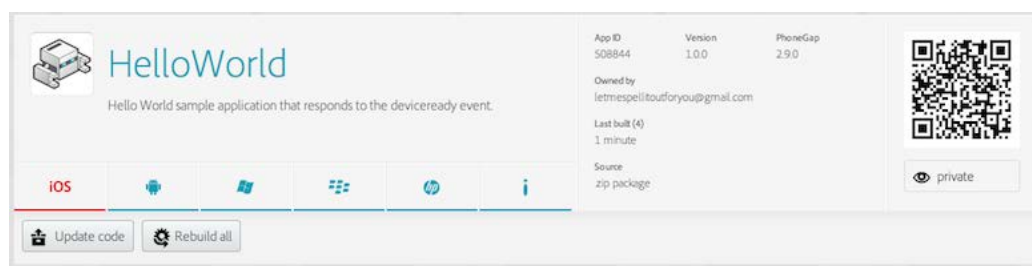


Figure 6 PhoneGap Build cloud platform

Titanium has its own Titanium Studio IDE. All titanium projects must be developed in this environment. They have a 3rd party plugins market integrated into the IDE. It also features a robust debugger and many other convenient tools at disposal for the developer. When it comes to SDKs, they have to be managed by the developers locally on their machine and reference them in the IDE. This method is less convenient than the cloud build platform provided by PhoneGap.

When it comes to performance, PhoneGap is much slower than Titanium. This is mainly due to the usage of non-native UI. Titanium compiles native UI and provides smoother interactions, much similar to the ones of native apps.

One more concern is the technology base of a developer. For developers who are only familiar with web development (HTML, JS, CSS) PhoneGap is a better choice. For developers with a bit of experience in native mobile development, Titanium is a better choice.

4.2. Why Titanium?

The decision comes from the project specification. Therefore, in our case, we would definitely go with Titanium. The winner feature is *native UI*. In a mobile application, the smooth native transitions and the responsiveness of the interface are still the most noticeable and most important parts for a good user experience. There is a noticeable difference in the user experience when it comes to PhoneGap apps. A developer working with PhoneGap must pay close attention to performance. This means that the knowledge of profiling tools as well as which web UI frameworks are mobile-friendly is essential for developing stable and user-friendly apps.

Titanium exists as a bridge between the native operating system and the app's code. The following graphic (Figure 7) illustrates its architecture.

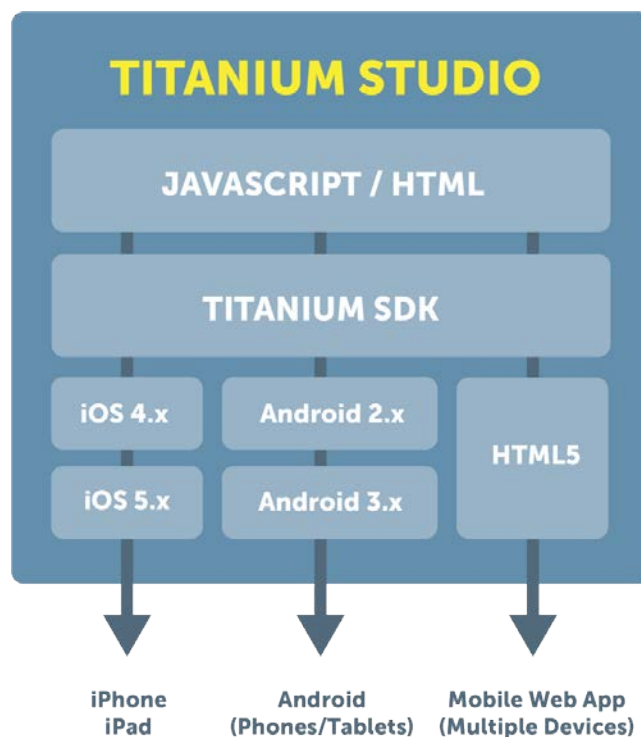


Figure 7 Titanium global architecture

At the bottom of the stack is the client operating system: Android, iOS, or the browser (for Mobile Web applications). At the top is the app and the built JavaScript. In between, there is the Titanium SDK and the APIs it exposes. The app is written in JavaScript, calling on the Titanium APIs to take actions like drawing buttons, opening windows, showing the camera, etc. The Titanium bridge (part of the SDK), which is called Kroll, translates those calls into their native equivalents. In other words, when a Titanium button is created, it's actually a proxy for a true native button. When the Titanium button is modified, for instance to change its label or add an event listener, Kroll applies corresponding changes to the native equivalent. When events occur in native-land, Kroll bubbles them up to the JavaScript code.

The goal of Titanium is to help developers leverage their JavaScript skills to build native mobile apps that run across multiple platforms. It gives developers the tools to build apps that look, feel, and perform native. Furthermore, Titanium apps fit well within the native ecosystem of each platform [11].

5. Going from IFML to Titanium

To move from an IFML model (like the email model shown in Figure 4) to a runnable Titanium code, a *Model-to-Text* (M2T) transformation is needed.

Many concepts, languages and tools have been proposed to help automate the derivation of text from models by using M2T transformations. These kind of transformations have been used for automating numerous software engineering tasks such as the generation of documentation or task lists.

Code-generation is by far the most important application of M2T transformations. Actually, the main goal of model-driven software engineering is to generate a running system out of the models. M2T transformations, in the area of model-driven software engineering, are mostly focused on code-generation that manages the transition from model level to code level. One of the major benefits of using models is that they can be used constructively to derive the system as well as analytically to better explore and verify the properties of the system [16].

When developing a model-based code generator, three questions are essential:

- *How much is generated?* This question should answer which parts of the code can be automatically generated from models. If only partial code-generation is possible, the second question here is: *which part?* It can be one layer (horizontal or vertical) completely generated while another may be completely manually developed. Moreover, it can be one layer partially generated and all the missing parts may be manually developed.
- *What is generated?* This question answers what kind of source code will be generated. Furthermore, the code must be generated in a way developers are able to read it and understand it.
- *How to generate?* When the requirements for the code-generation are specified, namely when the first two questions are answered, we have to decide how to implement these requirements.

A code-generator must support the following phases to be able to successfully transform models into text:

- *Load models* - models have to be deserialized from their XMI representation and their content has to be programmatically accessible for further processing.
- *Produce code* - collect the model information needed for generating the code. Usually, the object graph is traversed starting from the root element of a model breaking it down to its leaf elements.

The IFML model is the **model** and the cross-platform code for the Titanium application is the **text** for the M2T transformation used in this thesis. IFML has dozens of components to choose from, which should be mapped into some of the hundreds offered by Titanium.

We chose a set comprised of five components: **Window**, **List**, **Details**, **Form** and **View Element Event**. We generate the XML view and the JavaScript code (if necessary), for each

of these components. The only file we do not generate is the TSS file, which defines the style of the components, similar to the commonly known CSS files. This answers the first two questions from above (*how much and what we generate?*). Answering the third one is somewhat more complicated and requires more extensive elaboration.

In Figure 8 we show how a simple IFML model with a Window and a child List component is transformed into a Titanium View XML file with a Window node and a nested ListView node.

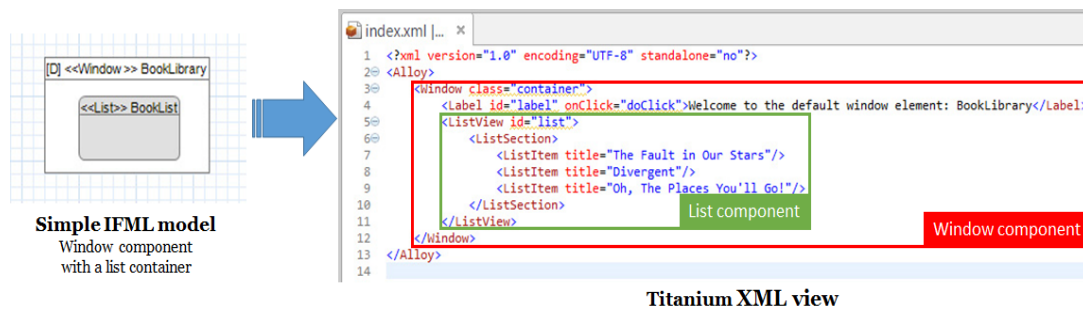


Figure 8 IFML to Titanium transformation example

A specific Java web tool has been developed, that handles the whole process of generating Titanium code from IFML models. This tool is referred to as **IFML to Titanium Project Generator (ITPG)**. The ITPG accepts an IFML model as an input and outputs a ready to run Titanium project. One is able to import this project into Titanium Studio and continue working on it as with any other Titanium project. The ITPG generates all the necessary code for the Titanium app as an output. It acts like a bridge between IFML and Titanium, as shown in Figure 9 below.



Figure 9 IFML to Titanium transformation process

The ITPG tool is the answer to the third aforementioned question (*how we generate?*). It's expressive power and limitations are more broadly explained in subsection 5.2 of this thesis.

5.1. Mapping IFML to Titanium

5.1.1. Visual Constraints and Heuristics

The first constraints came from the fact that IFML does not define the actual UI layout nor the style of the components. Because of that, we set and define some rules and heuristics in order to move forward.

The first thing we have to set is the UI layout. The main two containers in Titanium, the Window and the View, can employ one of three layout modes by setting their layout property to one of the following values:

- **absolute** - This is the default mode. You specify point coordinates on a grid relative to the parent container's top/left or bottom/right corner.
- **vertical** - This layout mode stacks child views vertically. The child's top property becomes an offset value. It describes the number of units from its previous sibling's bottom edge where the view will be positioned.
- **horizontal** - This layout mode lines up child views horizontally. The child's left property, similar to vertical, becomes an offset. This time, it's the position from the previous sibling's right edge.

In Figure 10 we show an example of vertical and horizontal layout.

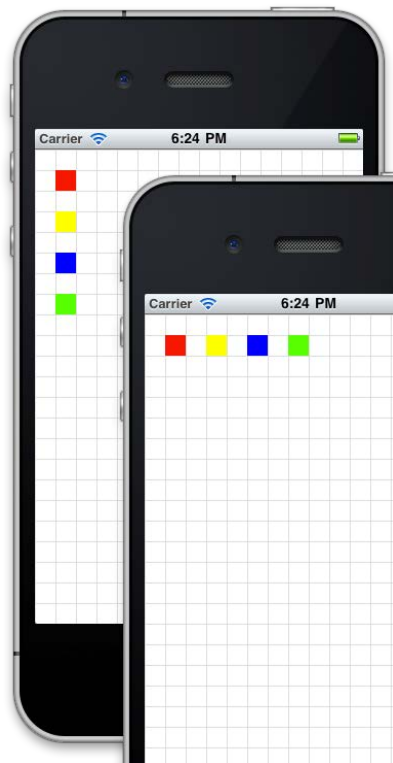


Figure 10 Example of vertical and horizontal layout in action

We set the layout of each component that contains subcomponents to **vertical**. Setting a component layout to vertical stacks all the subcomponents vertically, making them easily accessible and visible to the user. This is clearly the most simple and most useful layout to represent the UI of a generated mobile app.

Since IFML does not take care of the styles, we did not implement any styling to the UI components. That means we are not writing TSS style rules (similar to the familiar CSS, only in Titanium) to position and change the visual aspects of a component.

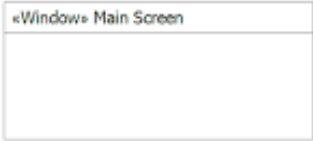
A heuristic conclusion on the complexity and the depth of navigation in a mobile app has to be made. Based on the general app structure mentioned earlier in subsection 2.2, as well as the fact that this is more of a research thesis, we choose to limit the depth of nested components in windows to one level. Despite the fact that we make this compromise, we keep in mind that cross-platform tools are best suitable for deep, complex, data-driven mobile apps. They ease the process of developing and maintaining this kind of apps in a cross-platform environment.



5.1.2. Component Mapping Proposal

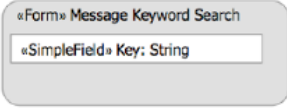
The layout of an app is mostly determined by the components used and by the way they are used. Titanium by default does not support all the components available in IFML and vice versa. Therefore, we need a mapping table of a subset of components along with certain rules about how the ITPG transforms the model components created in IFML into valid mobile application structure ready for immediate cross-platform development in Titanium Studio. Titanium Studio is the IDE for developing cross-platform Titanium apps.

Some of the IFML components should be mapped to Titanium top-level components, others to Optional Titanium Module Packages (such as Maps, Facebook, NFC etc.), and others to concrete Titanium UI elements (such as Buttons, Labels, Views, Windows etc.). Sometimes it would be necessary to map even platform specific components (e.g. Menus, Action Bars and Notification Managers) through platform specific code branching with conditional expressions, like *if-then-else* statements.

The following table represents our final mapping proposal for five most commonly used components, based on an extensive research on both IFML and Titanium. It contains a brief description of each pair of mapping components, as well as restrictions and concerns regarding the mapping process itself. Some of the components are clearly too broad to cover in this thesis, therefore we put notes and possible future work in the table itself.

IFML component	mapping	Titanium component
<p>Window</p>  <p>A View Container rendered as a Window.</p> <p>Usually an HTML page or a desktop window.</p> <p>Properties:</p> <ul style="list-style-type: none"> ● Name ● Is Land Mark ● Is Default ● Is Xor ● Is Modal ● Is New 	<p>Name -> Window Title</p> <p>Set the title of this window in your mobile app</p> <p>Is Land Mark -> Menu Item</p> <p>Create a link to this window directly from the menu of your mobile app</p> <p>Is Default -> Start Screen</p> <p>Set this window as the start screen of your mobile app (only one per app)</p> <p>NOTE:</p> <p><i>There are special views that manage windows (such as NavigationWindow, SplitWindow and TabGroup) but they will not be covered by this mapping.</i></p> <p>FUTURE WORK:</p> <p><i>“Is Xor” property can be used to create tabbed windows in the future.</i></p>	<p>Ti.UI.Window</p> <p>A window is a top-level container which can contain other views. Windows can be <i>opened</i> and <i>closed</i>. Opening a window causes the window and its child views to be added to the application render stack, on top of any previously opened windows. Closing a window removes the window and its children from the render stack.</p> <p>Windows <i>contain</i> other views, but in general they are not <i>contained</i> inside other views.</p>

<p>View Container</p>  <p>An element of the interface that comprises elements displaying content and supporting interaction and/or other view containers.</p> <p>Properties:</p> <ul style="list-style-type: none"> • Name • Is Land Mark • Is Default 	<p>Name -> View ID</p> <p>Set the ID of the View as the name</p> <p>NOTE:</p> <p><i>One suggestion is to allow the use of “Is Land Mark” & “Is Default” only in the case of Windows. When multiple View Containers are added to a single component we show them all together.</i></p>	<p>Ti.UI.View</p> <p>A View represents an empty drawing surface or container created by the method <u>Ti.UI.createView</u>.</p> <p>The View is the base type for all UI widgets in Titanium. Views can be added to Windows or nested inside other views. It’s the closest representation of the DIV element in HTML.</p>
<p>List</p>  <p>A View Container used to display a list of DataBinding instances. Table with rows of elements of the same kind.</p> <p>Properties:</p> <ul style="list-style-type: none"> • Name 	<p>Name -> Section Title</p> <p>Set the title of the first section in the ListView</p> <p>NOTE:</p> <p><i>There is also the TableView component that can also be mapped with an IFML List.</i></p> <p><i>We are choosing the ListView because it uses a data-oriented approach vs. the view-oriented approach of the TableView.</i></p> <p>FUTURE WORK:</p> <p><i>Creating a data set of items and adding it to the ListView section is left as potential future work, or its implementation can later be done inside Titanium Studio.</i></p>	<p>Ti.UI.ListView</p> <p>A list view is used to present information, organized into sections and items, in a vertically-scrolling view.</p> <p>A ListView object is a container for <u>ListSection</u> objects that are, in turn, containers for <u>ListItem</u> objects.</p> <p>List view is designed for performance. One side effect of the design is that the views cannot be directly manipulated (add children, set view properties and bind event callbacks) as in TableView.</p>

<p>Form + Simple Fields</p>  <p>A ViewComponent used to display a form that is composed of Fields. HTML form.</p> <p>Form Properties:</p> <ul style="list-style-type: none"> • Name <p>SimpleField Properties:</p> <ul style="list-style-type: none"> • Name 	<p>Form Name -> View ID</p> <p>Set the ID of the View as the form name</p> <p>SimpleField Name -> TextField ID</p> <p>Map each SimpleField into Titanium's own TextField component. The TextField ID and hint text will get the value of the SimpleField Name property.</p> <p>NOTE:</p> <p><i>There is the problem of mapping different type of fields (date, password, image, dropdown selection) in a form.</i></p>	<p>Ti.UI.View + Ti.UI.TextField + Ti.UI.Button</p> <p>The Form can be represented by a view in Titanium. To that view, we later add all the simple fields as text fields.</p> <p>By default we add a button to each form.</p> <p>A TextField is just a single line text field with a hint text.</p>
---	---	---

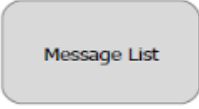
<p>View Component</p>  <p>An element of the interface that displays content or accepts input.</p> <p>For example: HTML list, image gallery, input field.</p> <p>Properties:</p> <ul style="list-style-type: none"> • Name <p>e.g. TextArea_AboutMe</p> <p>TextArea – the targeted component AboutMe – the ID of that component</p>	<p>Name Prefix -> Titanium Component</p> <p>The name prefix should select the titanium mapping component (e.g. Label, TextField, ImageView, Button, TextArea)</p> <p>Name -> Component ID</p> <p>Map the view component name into component's ID</p> <p>NOTE:</p> <p><i>Here we suggest some rule for the View Component Name property, like having a prefix which selects the Titanium component that it will map into. See example on the left.</i></p> <p>FUTURE WORK:</p> <p><i>Some better way of mapping can be used here, by extending IFML with more specific components for each of the Titanium components mentioned on the right.</i></p>	<p>Ti.UI.Label, Ti.UI.TextField, Ti.UI.ImageView, Ti.UI.Button, Ti.UI.TextArea</p> <p>A Label is actually just a normal text label, with optional background image.</p> <p>A TextField is just a single line text field.</p> <p>An ImageView is a view to display a single image or series of animated images.</p> <p>A Button is just like a normal view that has four states: normal, disabled, focused and selected. Usually an event listener is attached to it that performs some action.</p> <p>A TextArea represents a multiline text field that supports editing and scrolling.</p>
---	--	---

Table 3 IFML to Titanium component mapping proposal

5.2. Mapping Implementation

In this section we describe in details the ITPG tool which was created as a M2T transformation tool between the IFML models and Titanium code. As mentioned before, ITPG stands for *IFML to Titanium Project Generator*. It accepts an IFML model as an input, and generates a whole Titanium project as an output. There is a *Hello World* titanium project template which we duplicate and use as a starting point for each generated project. After that, the generator adds and modifies files according to the uploaded model. The project is then ready to be imported and built as-is in Titanium Studio. With that being said, we created a transformation process for the generator, comprised of the following steps:

1. A Titanium project template is copied and the configuration file (*tiapp.xml*) is updated.
2. Data Model is created based on the Form component.
3. IFML Windows are transformed into Titanium Windows, creating a separate XML, JS and TSS file for each of them.
4. View Element Events, children of an IFML Window component, are transformed into Titanium Menu items.
5. IFML Lists are transformed into Titanium ListView components.
6. IFML Details are transformed into Titanium View components with labels to display each attribute.
7. IFML Forms are transformed into Titanium View components with text fields and a save button.

This process is show in Figure 11 below.

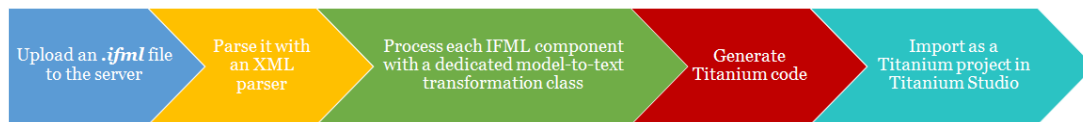


Figure 11 ITPG process

With that being said, ITPG has a limited expressive power at this moment, keeping in mind its purpose is to demonstrate capabilities of an end-to-end (IFML to Titanium) generation of cross-platform mobile applications. We summarize the limitations in the following list:

- ITPG supports models that contain only one list, one form and one details view.
- There must be only one default window in the IFML model, which is set as the start page of the mobile app.
- There can be endless empty windows.
- The data source is generated out of the form.
- The form comes with a button that adds the form input as a new item in the data source.
- After the item is added, the window with the form is closed and we show the default window again.
- If there is a *Select Event* on the list, a click on each item in the list is automatically added, which opens the details window showing all the information available for that item.
- Each *View Element Event* is added as a menu button to the window in which it belongs and it must be connected to another window.

All of these limitations come from the fact that this code generator was developed in order to prove that such mapping is possible. There are many cases that are not covered with this implementation, therefore the types of applications we can generate are fairly limited for now.

ITPG is developed in Java and can be run on any Java compatible server. It has a simple interface that offers to choose the project name and to upload your IFML model (.ifml file). This can be seen in Figure 12 below.

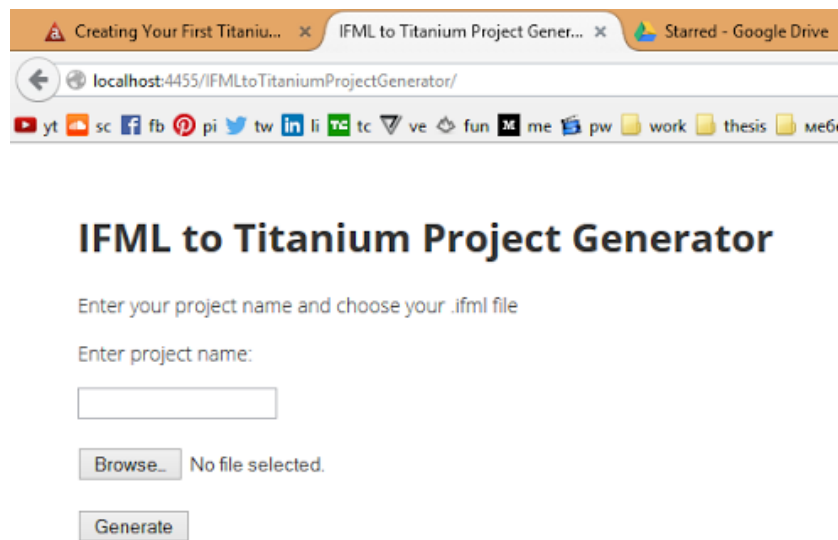


Figure 12 ITPG web interface

The “*IFML to Titanium Project Generator*” Java EE project offers not just the functionality to generate these kind of projects, but also introduces an easy and convenient way to do that through a web interface.

Locally we have one main directory where this generator operates. We’ve set it up to be located in **C:\ITPGData**. This directory structure can be seen in Figure 13 below.

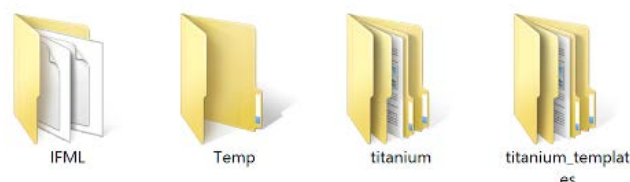


Figure 13 ITPGData folder structure

It contains the following directories:

- The **IFML** directory contains the ifml models uploaded by the user. (ex. **IFML/[ifml-model-name].ifml**)
- The **titanium** directory contains the Alloy Titanium projects generated for each ifml model. (ex. **titanium/[ifml-model-name]**)
- The *Hello World* titanium project template is located in the **titanium_templates** directory under the name of **alloyHello**.

- The *Temp* directory is used for some temporary files generated by the file upload control. The content of this directory can be deleted from time to time, but not the directory itself.

Each time a titanium project is generated from an ifml model, a copy of **alloyHello** will be placed in the *titanium/[ifml-model-name]* directory.

Once the project is copied, ITPG parses each IFML model with an XML parser. The model has its own tags, attributes and attribute values that we need to understand and process in order to perform all the necessary M2T transformations. The ITPG edits files, such as the *tiapp.xml*, *index.xml*, *index.js* and *model.js*. It also creates new files, such as XML, JS and TSS files for all the additional windows and their components.

The *Window* is the main component of any application view. Each window can contain multiple child components, for example, views, lists, forms, details and events. Therefore, we describe each of the transformations used in this thesis, starting with the one that maps the IFML window component into a Titanium one. Moreover, we show all the developed transformation classes for this thesis in Figure 14 below.

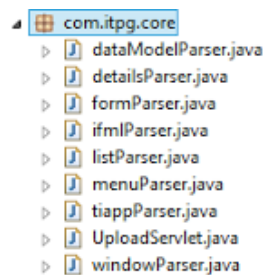


Figure 14 Transformation classes developed for this thesis

5.2.1. Constructing the Data Model

Before we look at the mapping implementation, we must explain how we handle data. Every application suitable for cross-platform development relies on data, and IFML's Eclipse plugin does not contain data components, which we find as a major limitation. That being the case, we presume that a form, by its nature, handles data. So, we try to extract a data model out of the form itself. Therefore, we developed a workaround where we first look for a form anywhere in the model and create a database based on it. The *dataModelParser* class handles this before mapping any other elements.

We iterate through each of the *SimpleField* element in the form. As a result of this class, there are two global variables that are later used to manipulate the views of the mobile app:

- *mainParam* (String) - this is the first *SimpleField* element, and it is used as display text for any element which has this datasource
- *otherParams* (String[]) - this is an array of all the remaining *SimpleField* elements in the form

The *SimpleField* element does not specify the type (string, integer, date, picture, or file) nor the format (there is no validation implemented). Thus, we set the type for all the *SimpleField* elements to *string* and no validation.

The last step is to create an alloy model, which inherits from the Backbone.Model class [17]. Models are specified with JavaScript files, which provide a table schema, adapter configuration and logic to extend the Backbone.Model class. Models are automatically defined and available in the controller scope as the name of the JS file. This JS file is already available in the *Hello World* template mentioned before. We just modify it by adding all of the form fields as columns of type *string*, as shown with the code below. Then we save the model and we use it later as our database. If there was validation to be implemented, it has to be added in this file, by extending the model with the *validate(attrs)* method.

```
config: {  
  columns: {"id": "integer", "title": "text", "author": "text", "year": "text", "description": "text"},  
  adapter: f
```

Later on we create alloy collections, which are ordered sets of models and inherit from the Backbone.Collection class [17]. Alloy collections are automatically defined and available in the controller scope as the name of the model [18]. Moreover, this data is only available in a single session of the app. In order to make it persistent between sessions, we must utilize some local storage or remote server storage. This is currently out of scope and there are many questions and parameters in order to make this work, hence they are not covered in this thesis.

5.2.2. Window Element

We use *element* and *component* as synonyms. In this part we use more *element*, since it's used in the IFML XML semantic.

In the Alloy framework, the view component represents the UI of the application, comprised of XML file (ex: index.xml) and Titanium style sheets file (ex: index.tss). The XML markup defines the structure of the view, while the TSS file contains the styling elements applied to the XML markup. It is similar to the relationship between HTML and CSS. There are also controllers (ex: index.js) that contain the application logic used to control the UI and communicate with the data model.

So, we create these three files for each of the IFML windows in model, and place them in the corresponding project directory (controllers, styles and views), as shown in Figure 15.

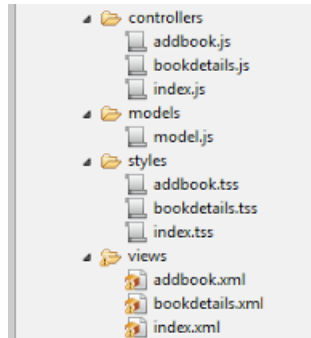


Figure 15 Titanium project structure

After this, each of the windows has a separate view in the app. But, in order to use them, there must be a defined navigation between them. There are multiple ways to navigate from one view to another, like: click on a list item, click on a button in the view, through a menu button or by navigating through a tabbed window component.

There are several different ways to manage windows in Titanium:

- simple window
- tabbed window
- split-window
- navigation window

Each of them have their own distinguished features and markup. From here comes the necessity to introduce certain rules on how to use IFML to build mobile apps. We must define ways how we build the IFML model itself, so the generator can recognize and map certain types of windows in Titanium. For this thesis we map all the IFML windows into simple Titanium windows, without navigation, split windows or tabs. We also used the *isDefault* attribute to mark the start screen of the app. Therefore, there has to be only one window in the IFML model with this attribute set to true. Note that not all of the proposed mapping options shown in Table 3.1 are developed in this thesis.

After parsing the IFML model with an XML parser, we iterate through all of its *interactionFlowModelElements*. These elements are the children of the root *interactionFlowModel* element which represents the whole IFML model. We implement a switch statement which takes the window elements and sends them to the window transformation class. The code chunk follows below.

```
switch (eElement.getAttribute("xsi:type")) {
case "ext:Window":
    result += "<br> processing window: " + eElement.getAttribute("name");
    result += windowParser.readFile(eElement.getAttribute("name"),
        nList.getLength(), projectName, eElement.getAttribute("isDefault"), nNode, nList);
    break;
default:
    result += "<br>-----no case for this element-----<br>";
    break;
}
```

The next step creates the necessary files for each of the window elements. The *Hello World* template contains an empty titanium window, named *empty.xml*. We copy this file with the name of the IFML window element into the */app/views* directory. The same procedure is done for the TSS and JS files, which are copied to the */app/styles* and */app/controllers* directories respectively. This directory structure was shown above in Figure 15.

5.2.3. Form, List and Details Elements

These elements have to be children of some window element. Therefore, we iterate through the window's children elements. Similarly as before, we implement a switch statement which propagates the form elements to the form transformation class (*formParser*), the list elements to the list transformation class (*listParser*), and the details elements to the details transformation class (*detailsParser*).

Since we already created the files for each window, we modify those files through each of the transformation classes. The classes add the necessary XML elements and JS code in order to have the elements as in the model. While implementing this transformations we noticed some limitations of IFML:

- List element does not specify what data to display in the list items. Therefore, we chose one attribute as the list item text, the default one from the data model.
- List element requires a data collection. We bind it to the alloy collection mentioned in subsection 5.2.1, since there is no list attribute to specify the data source.
- Form element fields have no type specified, so we set it to *string*. Ideally each form field should have its own type defined in the model.
- Details element does not specify what data to display, so we display all of the fields.

5.2.4. Action, View Element, Select and Submit Events

These are the four event types that we cover in this thesis. As an addition to the XML markup, we must also create JavaScript functions for each of the events. If we look at the *Select Event* for example, first we check if there is a select event defined in the model and then we store three global variables:

- *hasClick* - a global boolean variable that tells us if the list has a select event or not.
- *targetWindow* - a global string variable that contains the value of the *trgtInteractionFlowElement* attribute of the list's *NavigationFlow* element
- *clickFunction* - the name of the function in the JS file that should be called when clicking a list item

In the code below we can see how these three global variables are created while processing the list element with the *listParser* class.

```

hasClick = eee.getAttribute("xsi:type").equals("ext:SelectEvent");
if(hasClick) {
    NodeList selectEventChildren = eee.getChildNodes();
    for (int temp2 = 0; temp2 < selectEventChildren.getLength(); temp2++) {
        Node nNode2 = selectEventChildren.item(temp2);
        if (nNode2.getNodeType() == Node.ELEMENT_NODE) {
            Element eee2 = (Element) nNode2;
            if(eee2.getAttribute("xsi:type").equals("core:NavigationFlow")) {
                targetWindow = eee2.getAttribute("trgtInteractionFlowElement");
            }
        }
    }
    String targetWindowNumber = targetWindow.split("@interactionFlowModel/@interactionFlowModelElements.")[1].substring(0, 1);
    Element targetElement = (Element) fullIFML.item(Integer.parseInt(targetWindowNumber));
    clickFunction = targetElement.getAttribute("name").replace(" ", "").toLowerCase();
    listView.setAttribute("onItemClick", clickFunction);
}

```

If the event target element is a window, we presume that when this event is triggered, we need to open the target window in the app. The *parameter binding* element helps us understand which parameters should be transferred from the source window to the target window. To implement this we use the *args* input parameter when we open the new window. The JavaScript code below, which is generated by the list transformation class, illustrates that binding.

```

function bookdetails(e) {
    var selectedItem = e.section.getItemAt(e.itemIndex);
    var args = {
        title : selectedItem.properties.title,
        author : selectedItem.properties.author,
        year : selectedItem.properties.year,
        description : selectedItem.properties.description
    };
    var targetWindow = Alloy.createController("bookdetails", args).getView();
    targetWindow.open();
}

```

The biggest problem comes with the action event. The action event is triggered by an IFML action element. The only logical transformation would be to create a JS function from the action element, which as an output triggers an action event (save data, load data, calculate, activate/deactivate phone feature, open window, close window, etc.). Since there is no support in IFML for this kind of information, we presume that the form element needs to save data. Therefore, we create a new model with the information from the form, we add that to the alloy collection and we close the current window to return to the main window. This might not work in other scenarios. For example, when we don't want to save the data or when the form is located in the main window it will close the app once we call the *close()* method at the end of the JS function. An example of action event can be seen below.

```

var appModel = Alloy.Collections.model;
function save() {
    var model = Alloy.createModel('model', {
        title : $.titleInput.value,
        author : $.authorInput.value,
        year : $.yearInput.value,
        description : $.descriptionInput.value
    });
    appModel.add(model);
    model.save();
    $.addbook.close();
}

```

Now, we can go deeper into the example application developed for this thesis. At the same time we can explain why certain decisions were made, as well as pinpointing the major drawbacks of IFML when used for modeling cross-platform mobile applications.

6. Case Study: Book Library Example

6.1. Application Structure in IFML

As mentioned before, we use IFML for expressing the content and user interaction of the front-end of the example application. As a tool for creating this model we used the IFML IDE Eclipse plugin [19]. At the time of writing this thesis, the plugin was still in early beta stage, so not all the components and features were available to us.

We are going to model a Book Library App, which is the *Book Library* example project from Appcelerator Titanium: *Creating Your First Titanium App Example* [20]. We will use this project to practically test the developed ITPG.

All the decisions throughout the modeling process, as well as the discussions and considerations are strongly based on the context of this example. This is to avoid the possible ambiguities that might come along using a more extensive example.

As with most of the projects in the software domain, especially application development, we start by brainstorming ideas and concepts on a whiteboard. We choose the case of a *Book Library App* as the most simple and most convenient for elaborating the topic of this thesis. The functionality of this application is simplified in order to ease the process of mapping elements and code generation. By this, we are able to demonstrate how a cross-platform mobile application is generated out of a model-driven approach based on IFML and cross-compilation.

Using the general structure of mobile applications mentioned in subsection 2.2, we can say that the top level view of the app is the *Book list* window. This is the first window of the app and also the main window of our app. It contains the list of books, button to add a book and an action on each list item to open the each book details view. The other two views of the app are part of the detail/edit view according to the general structure presented in Figure 3. *Add book* window contains the form for creating a new book, and buttons to confirm and cancel the action. *Book* window contains the detailed data available for each book (title, author, year, description).

The next step is modeling the app's content and events according to IFML notation. We have three different windows as content of this app, for which we use the IFML *Window* component:

- The main (*Book list*) window is utilizing the IFML *List* component to display the list of books available.
- The *Add book* window contains the IFML *Form* component for creating a new book.
- The *Book* window uses the IFML *Details* component to show the detailed data available for each book.

There are three events that can be taken by the user in this example app:

- *Add book* - triggered by pressing a button on the main window. We model this event with the IFML *View Element Event* component. This event opens the *Add book* window.
- *View book details* - this event is triggered by an IFML *Select Event* component attached to the Book list.
- *Create book* - this event is triggered by pressing on a button located in the *Add book* window, firing the *Add book* action which is modeled by the IFML *Action* component. Moreover, the action has an IFML *Action Event* which returns the user back to the *Book list* window.

Note that all the styles for this application were created manually, with the goal to improve the look and feel of this example. The styles (TSS files) were not in the scope of the implementation. The final model of this example is shown in Figure 16 below.

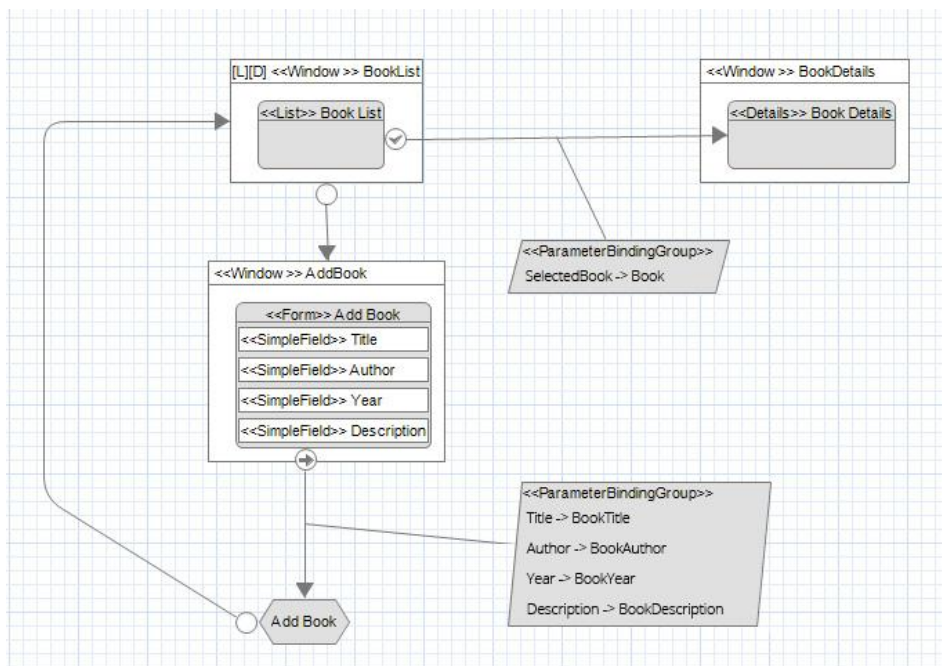


Figure 16 IFML model for simple Book Library App

After uploading the model of this example project to the ITPG tool, the generation produces a log like the one shown in Figure 17 below.


```

-----Window children-----

Child Element: Add Book
Element Tag: viewElements

Element Type: ext:Form
--

processing form: Add Book
FORM!!!!

Current Element: Title
Element Type:
-----no case for this element-----

Current Element: Add Book Event
Element Type: ext:SubmitEvent
-----no case for this element-----

Current Element: Title
Element Type: ext:SimpleField
Processing form element: Title

Current Element: Author
Element Type: ext:SimpleField
Processing form element: Author

Current Element: Year
Element Type: ext:SimpleField
Processing form element: Year

Current Element: Description
Element Type: ext:SimpleField
Processing form element: Description
JS Code: title : $.$titleInput.value,author :

Done adding form toADDBOOK.XML
-----

JS for form successfully modified!
----end Window children----

Current Element: Add Book
Element Type: core:Action
-----no case for this element-----

-----
IFML MODEL SUCCESSFULLY MAPPED
-----

```

Figure 17 Part of the printed log from the ITPG tool

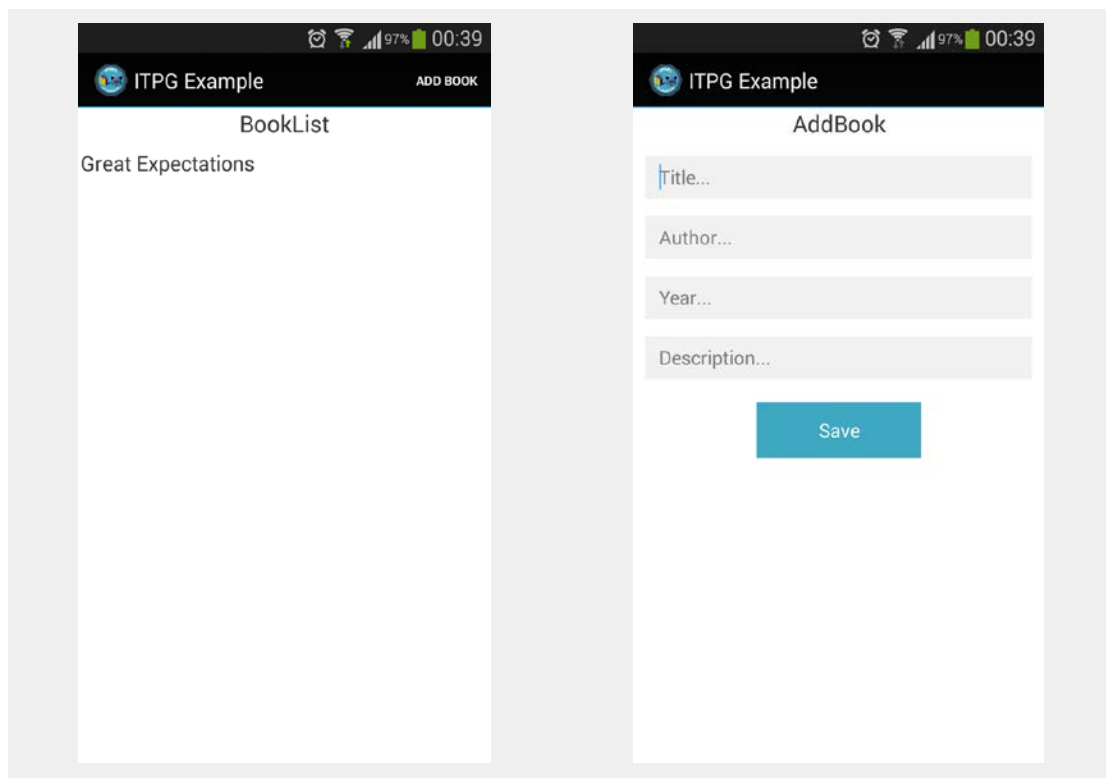
This log tells us that the generation was finished and that the whole IFML model was mapped successfully. Next, we import the generated project with the “*Existing Folder as New Project*” option in Titanium Studio. On the next screen it is important to check the “*Alloy*” and “*Mobile*” project types in order to be able to run this project properly. The last step is to run this project on a simulator or a connected device.

6.2. Example Break Down

The following screenshots are taken right after running the project on a Samsung Galaxy S3 device, without any modifications made in Titanium Studio. We'll shortly explain what is shown on these screenshots and what actions are available for the user.

When we first launch the app, we create the alloy collection as a database, which was generated by the data model transformation class.

The first screenshot (left) is the start screen of the mobile app. As modeled, the start screen is represented by the *Book List* window. There is only one sample book in the list, the one that was hardcoded in the list transformation class while mapping the IFML list component. On this window, there is also the *Add Book* button in the top right corner. Clicking this button takes us to the second screenshot (right), which represents the *Add Book* window. This window contains the form component with all the fields and a *Save* button.

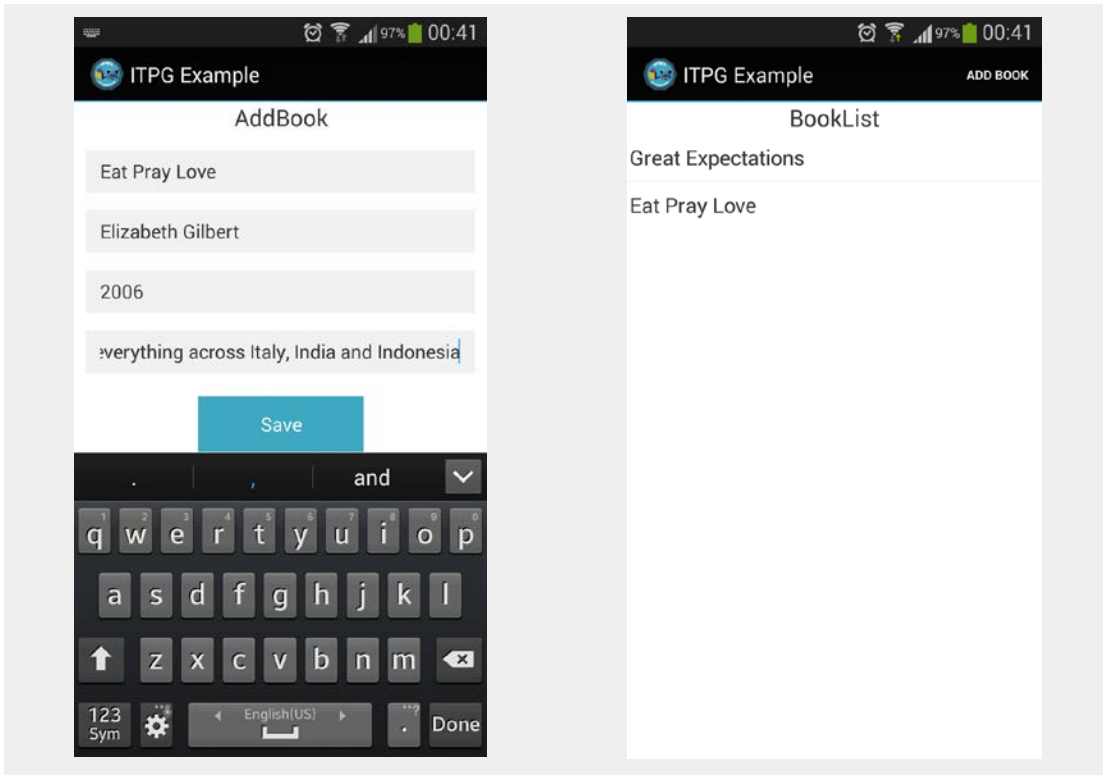


Book List window

Add Book window

On the next screenshot (left) we fill the form and press *Save*. The action creates a new alloy model with the data from the fields and appends it to the alloy collection that was created on the start. After that we close this window and we go back to the start screen (right).

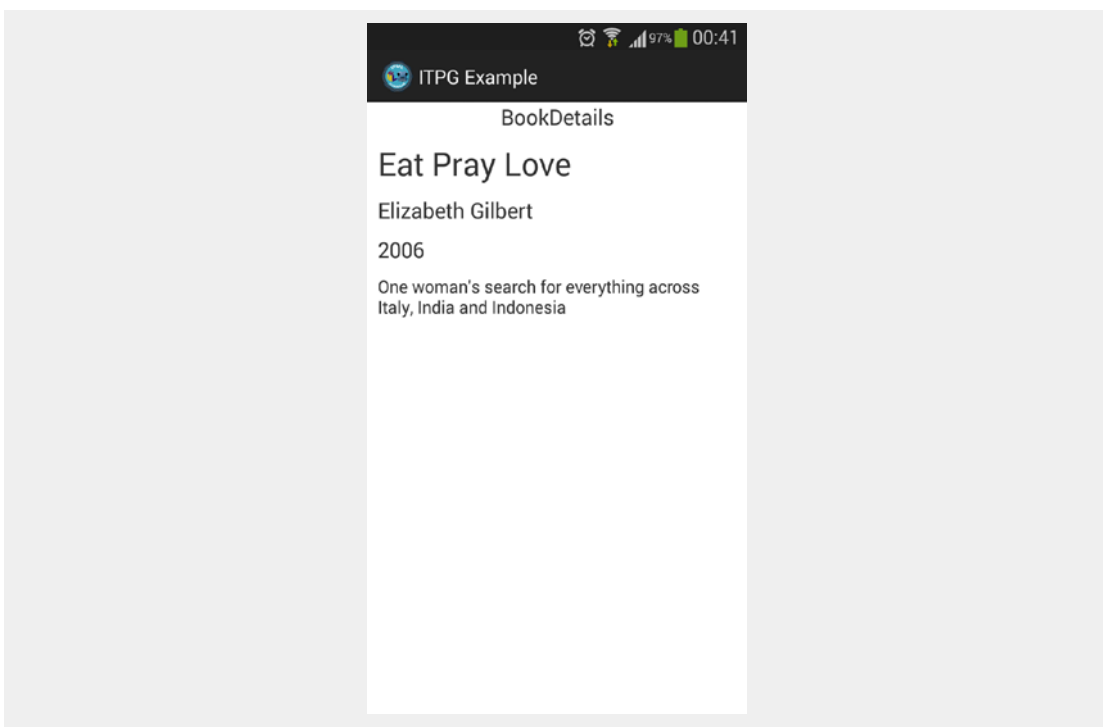
Now there are two books in the book list. One is the old one, and one is the book we just created with the form. From here there is only one more action that the user can take, and that is opening the details view.



Filled *Add Book* form

New book in the list

By clicking on an item in the list, the *Book Details* window opens, which is shown on the last screenshot. This window contains the details view and shows all the information about that book.



Book Details window

7. Related Work

There is already a significant number of research done towards model-driven approaches for mobile applications development. Some of the work is more abstract and in some sense more theoretical. However, there is a notable amount of work that proposes concrete solutions with practical implementation. We list the related work, starting from the theoretical one and continuing with the more concrete one:

(i) Florence T. Balagtas-Fernandez in his paper [26] proposes the MobiA tool, which is a prototype based on a conducted survey among non-expert users willing to create their own mobile applications. With regards to the participant's wishes for the tool, it has been designed to be graphical in nature, and features a drag-and-drop functionality. This tool is shown in Figure 18 below.

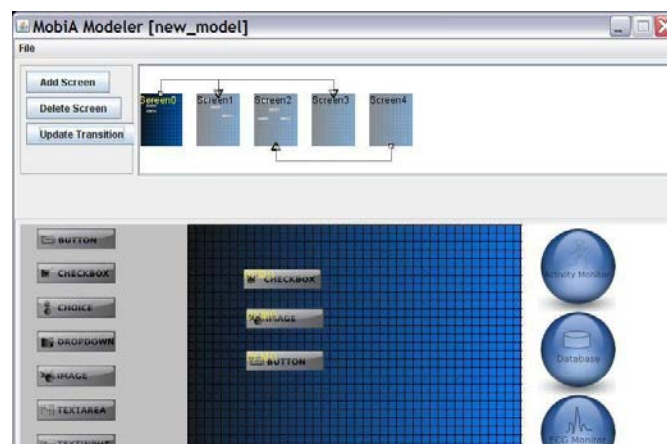


Figure 18 Initial MobiA prototype [26]

(ii) Mirco Franzago et al. in their paper [22] present the main principles and modelling languages of a collaborative modelling framework for mobile applications. The proposed modelling framework should enable designing apps at the right level of abstraction, and a multi-site, real-time modelling across different stakeholders at the same time. The proposed framework should also abstract the implementation-specific details to non-technical stakeholders in order to maximize reuse, and enable early validation through incremental prototypes and analysis. A schema of the collaborative development process is shown in Figure 19 below.

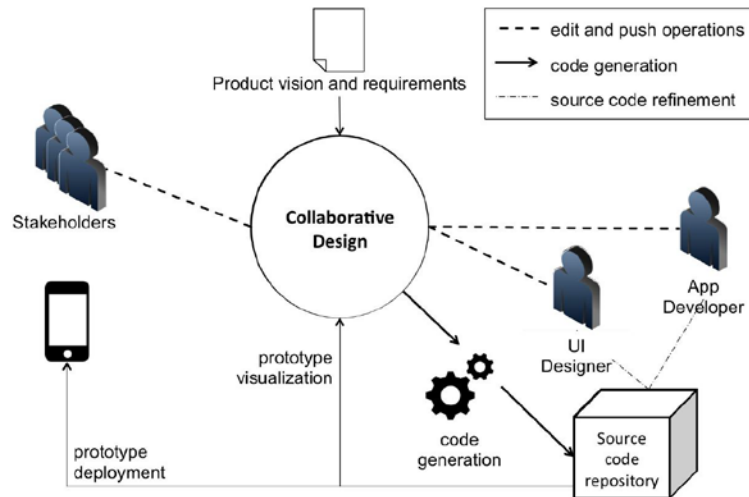


Figure 19 Collaborative development process [22]

(iii) In yet another paper [29], Abilio G. Parada et al. propose an MDE approach for Android applications development, which addresses modeling with standard UML notation. They conducted a case study in which an Android application was modeled in UML and code was generated from it, using the extension of GenCode. They only made a transformation from UML class and sequence diagrams to the target Android Java code, without consider any optimization in the generated code. As future work, they will consider the good practices for Android development, and thus generate more efficient code.

(iv) Chi-Kien Diep et al. in their paper [28] take advantage of web technology and model-driven approach to develop a web-based designer which provides friendly interface and highly extensible transformation architecture. The system provides an Online Integrated Development Environment for cross-platform graphical user interface, which allows native code project generation. Experiments with volunteers have shown that this solution can save up to 25-51% time to create the GUI of an application to three different platforms: Android, iOS and Windows Phone. The designer has limited functionalities, but there are intentions to support higher level of model abstraction and enhance user management to package the system into a commercial product. Figure 20 shows the main view of the designer application.

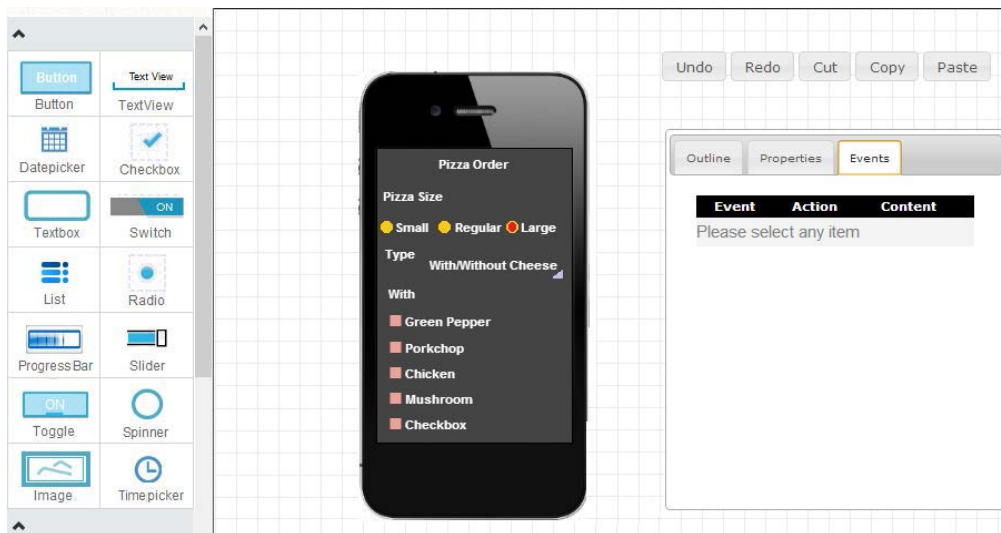


Figure 20 Web-based designer application screenshot [28]

(v) Glenn Cavarle et al. in their paper [25] present a solution named Dali which uses Smalltalk together with some Model Driven Engineering to allow multi-platform application design. When the application is mature enough, the idea is to finally generate a native target application. Dali provides a framework that can be used to design desktop as well as simple mobile applications. The set of available widgets and adapters remains to be expanded. Unfortunately, a Dali model provides the mean to generate target platform code, but generating a real application remains to be experimented;

(vi) Some work even focuses on performance optimization and power consumption of mobile applications, like the paper of Chris Thompson et al. [27]. Their approach addresses these challenges by enhancing model-driven engineering (MDE) tools to enable developers to quickly understand the consequences of architectural decisions. Their approach can help drawing conclusions long before implementation. This would significantly reduce production costs and time while substantially increasing battery life and overall system performance. From their experience, they could conclude the following points:

- By utilizing MDE it becomes possible to quantitatively compare design decisions and deliver some level of optimization with regards to power consumption,
- Developing applications for platforms such as Android require extensive testing as hardware configurations can greatly influence performance, and
- It is impossible to completely profile a system configuration because ultimate device performance and power consumption depends on user interaction, network traffic and other applications on the device.

(vii) Marco Brambilla et al. in their paper [7] present a mobile extension of OMG's standard IFML (Interaction Flow Modeling Language) for mobile application development. Their modeling on existing apps proves that the language is expressive enough to satisfy all the typical development needs on mobile. Moreover, they have real industrial experience, which gives positive feedback on the applicability, effectiveness and efficiency of this approach. Their planned future work covers the implementation of more refined code generators and the study of design patterns for model-driven mobile applications design. The developed modeling tool is shown in Figure 21 below.

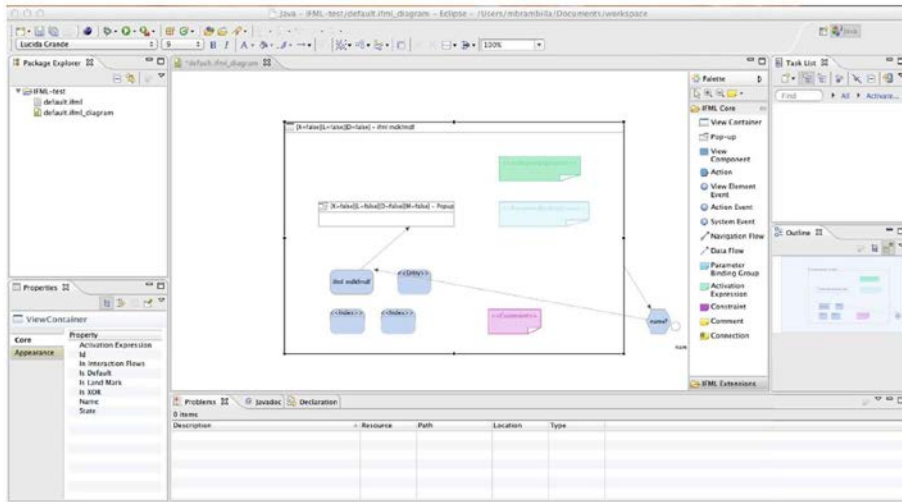


Figure 21 Snapshot of the open source IFML modeling tool [7]

(viii) Henning Heitkotter et al. in their paper [24] introduced the MD² model-driven framework for cross-platform development of mobile applications. While not being a general purpose tool that can be used for any kind of app, it has proven to be feasible for typical business apps even in its prototypic state. This framework runs as a set of plugins for Eclipse and it is based on a textual domain-specific language (DSL). The tool utilizes the MVC (model-view-controller) software architectural pattern, therefore you must write the model, the view and the controller in their DSL. The source code for both iOS and Android is automatically generated and ready to be installed. The Eclipse environment where the MVCs are created is shown in Figure 22 below.

```

package de.wuu.md2.liveApp.controllers

/*
 * Implement the controller here
 */

@main {
  appName "Live app"
  appVersion "1.0"
  startView address
  uninitialized startUpAction
}

contentProvider Customer customerContentProvider {
  providerType local
}

action CustomAction retrieveGPSPositionAction {
  call GPSupdateAction (street to customerContentProvider.address."street",
  city to customerContentProvider.address."city",
  postalCode to customerContentProvider.address.zipCode,
  "You are " + altitude + " meters above ground." to customerContentProvider.address."altitude")
}

action CustomAction startUpAction {
  bind action retrieveGPSPositionAction on address.retrieveGPS.onTouch
  map address.streetInput to customerContentProvider.address."street"
  map address.cityInput to customerContentProvider.address."city"
  map address.zipInput to customerContentProvider.address.zipCode
  map address.altitudeInput to customerContentProvider.address."altitude"
}

```

Figure 22 MD² Eclipse environment

The biggest difference between our approach and MD² is the input model. While MD² uses a textual modeling language, we use a graphical modeling language (IFML). Moreover, writing the MVC model for MD² will most probably be done by an experienced developer, while in our case a broader user base can create the IFML model.

Another very important difference is the output. As we can see, MD² generates final native apps by itself, while we generate a Titanium project and we deploy the final native apps through Titanium Studio. While first evaluation results of the MD² framework are promising, and most data-driven business apps can already be implemented, it is currently still limited with respect to certain functionalities (e.g. advanced device features). They are continuously working on MD² by refining it, extending it, and applying it to additional scenarios [23].

Similar approach can be found in other development tools, like Applause [21] for example. It is a toolkit for creating cross-platform mobile applications. It consists of a DSL to describe mobile applications and a number of code generators that will use these descriptions to generate native applications for the major mobile platforms (iOS, Android, Windows Phone). This toolkit is based on Eclipse and Xtext.

Both of these tools, the MD² and Applause, provide generated native apps as output. On the other hand, our ITPG tool generates only a Titanium project. This project needs to be imported in Titanium Studio and the native app deployed from there, hence the name of the thesis does not contain *development*. We only provide an easier and faster way to start a cross-platform mobile project. The real development environment stays within Titanium Studio, where the developers can modify, add or remove code if necessary.

This is a relatively new field of research, so limited number of tools and frameworks are available. However, it is an attractive topic and we can see new tools and papers emerging every day.

8. Conclusion and Future Work

As already discussed in the introduction, mobile phones are here to stay, and so are mobile applications. Cross-platform mobile development helps people build applications faster, with less resources and helps them increase their market reach. The main goal of this thesis was to prove that we can optimize this approach even more, by introducing a comprehensive mobile modeling language (IFML) and a code generation tool (ITPG). This two can be used to prototype more concepts and prototype them faster, but at the same time they help us experiment easily with components and choose what best fits the need of our application structure.

Since IFML focuses more on the view part, observed from an MVC point of view, we find it very intuitive and expressive when it comes to building a mobile application structure. Moreover, its graphical approach helps us build more user centric apps, simply because to build a model in IFML, one must think and act like a user.

Another thing we found really useful was the *DataBinding* for view components, list views and detail views. The *DataBinding* specifies the data source which can be anything from an XML file to a table in a database. It also contains conditional expressions and visualization attributes which determine the specific content obtained from the data source and the content shown to the user respectively.

Although IFML is a very robust and intuitive graphical modeling language, we found several features that can be improved in order to make it friendlier and more complete when it comes to modeling mobile applications. Regardless the fact that IFML is a view oriented language (in the MVC architecture), we still think the model and the controller must be covered as well in order to have a more complete mobile modeling language. Simple mapping components that transform to SQLite databases could be the first step in covering the model part of an MVC application. On the other hand, since the controllers are usually binded with the windows (screens) in mobile apps, all action and event components in IFML should belong somewhere in these controllers. Therefore, a more precise mapping of components that contain JavaScript code could help us create a more powerful and more capable code generator.

Besides extending IFML to cover all three parts of MVC, we should consider a deeper component compatibility. In our case, we feel like the IFML components could be extended with additional attributes in order allow a seamless transformation from IFML to Titanium. In many places we found that Titanium supports different types of the same component, while that is not the case with IFML. For example, Titanium supports several types of the *Window* element: simple, tabbed, split-window and navigation. In contrary, IFML has no window element types, so it is very hard to create an effective code generator when it comes to the *Window* component. This is the main and most important UI element in any mobile app, but improvements regarding support for different types of the same component can be made in components such as the *List* or the general *View Component*.

The general *View Component* is supposed to cover all sorts of elements, from buttons to image galleries. This is not the best approach, since all of these elements have unique properties and attributes, making it hard to depict them with one single component. Therefore, the third and final major improvement of IFML would be increasing the set of components. Many common and necessary mobile UI elements like menus, drawers, spinners, as well as mobile specific gestures such as swipe or pinch, and sensors such as GPS sensor or accelerometer could not be used in IFML models. This mobile specific features are essential when modeling mobile applications.

Nevertheless, many of these features are coming with the *AutoMobile* project as we already mentioned before, so things are moving forward when it comes to model-driven mobile application development.

When it comes to the possible future work regarding the code generator we developed, there are three things that I would focus the attention on:

- ***Offer multiple outputs*** - Add other platforms for which the tool could generate code, such as Phonegap or even native iOS and Android code generation.
- ***Generate database out of forms*** - While developing the example app in this thesis, the idea of generating the database out of the forms in the app seemed more and more practical. Starting from the fact that forms are the most commonly used mechanisms for data input, it makes perfect logic to derive the actual database tables out of them.
- ***Step by step wizard*** - developing an online tool that offers the user to create the IFML model online, choose various customization options, select the output (Titanium, Phonegap, native iOS, native Android, etc.) and finally download the generated project.

9. Bibliography

Note: URLs have been last accessed on 30th of November 2014.

- [1] Fergal MacErlean. "First Neanderthal cave paintings discovered in Spain", New Scientist, 10 February 2012.
<http://www.newscientist.com/article/dn21458-first-neanderthal-cave-paintings-discovered-in-spain.html>
- [2] John Heggestuen. "One In Every 5 People In The World Own A Smartphone, One In Every 17 Own A Tablet [CHART]", Business Insider, 15 December 2013.
<http://www.businessinsider.com/smartphone-and-tablet-penetration-2013-10>
- [3] "Global mobile statistics 2014 Part A: Mobile subscribers; handset market share; mobile operators", mobiForge, 16 May 2014.
<http://mobiforge.com/research-analysis/global-mobile-statistics-2014-part-a-mobile-subscribers-handset-market-share-mobile-operators?mT#mobiletablet>
- [4] "Phone Finder results", GSMArena.com.
<http://www.gsmarena.com/results.php3?>
- [5] "Modeling language", Wikipedia.
http://en.wikipedia.org/wiki/Modeling_language
- [6] "The AutoMobile project".
<http://automobile.webratio.com/>
- [7] Brambilla, Marco, Andrea Mauri, and Eric Umuhoza. "Extending the Interaction Flow Modeling Language (IFML) for Model Driven Development of Mobile Applications Front End." In *Mobile Web Information Systems*. Springer International Publishing, 2014. 176-191.
- [8] Jim Cowart. "Pros and Cons of the Top 5 Cross-Platform Tool", Developer Economics, 12 November 2013.
<http://www.developereconomics.com/pros-cons-top-5-cross-platform-tools/>
- [9] Dmitry Chervov. "PhoneGap vs Titanium", Sphere Consulting Inc Blog, 2 June 2014.
<http://blog.sphereinc.com/2014/06/phonegap-vs-titanium/>
- [10] "Titanium Studio", Appcelerator Inc.
<http://www.appcelerator.com/titanium/titanium-studio/>
- [11] "Titanium Platform Overview", Appcelerator Docs.
http://docs.appcelerator.com/titanium/3.0/#!/guide/Titanium_Platform_Overview
- [12] Donald A. Norman. *The Design of Everyday Things: Revised and Expanded Edition*. Basic Books, 2013.
- [13] Elaine McVicar. "Designing for Mobile, Part 1: Information Architecture", UX Booth, 25 September 2012.
<http://www.uxbooth.com/articles/designing-for-mobile-part-1-information-architecture/>

- [14] Elaine McVicar. "Designing for Mobile, Part 2: Interaction Design", UX Booth, 26 March 2013.
<http://www.uxbooth.com/articles/designing-for-mobile-part-2-interaction-design/>
- [15] "App Structure", Android Developers.
<http://developer.android.com/design/patterns/app-structure.html>
- [16] Brambilla, Marco, Jordi Cabot, and Manuel Wimmer. "Model-driven software engineering in practice." In *Synthesis Lectures on Software Engineering* 1, no. 1 (2012): 1-182.
- [17] "Backbone.js".
<http://backbonejs.org/>
- [18] "Alloy Collection and Model Objects", Appcelerator Docs.
http://docs.appcelerator.com/titanium/latest/#/guide/Alloy_Collection_and_Model_Objects
- [19] "Interaction Flow Modeling Language", WebRatio.
<http://www.webratio.com/portal/content/en/ifml-standard>
- [20] "Creating Your First Titanium App", Appcelerator Docs.
http://docs.appcelerator.com/titanium/3.0/#/!/guide/Creating_Your_First_Titanium_App
- [21] "APPlause".
<http://applause.github.io/>
- [22] Franzago, Mirco, Henry Muccini, and Ivano Malavolta. "Towards a collaborative framework for the design and development of data-intensive mobile applications." In *Proceedings of the 1st International Conference on Mobile Software Engineering and Systems*, pp. 58-61. ACM, 2014.
- [23] Heitkötter, Henning, and Tim A. Majchrzak. "Cross-Platform Development of Business Apps with MD2." In *Design Science at the Intersection of Physical and Virtual Design*, pp. 405-411. Springer Berlin Heidelberg, 2013.
- [24] Heitkötter, Henning, Tim A. Majchrzak, and Herbert Kuchen. "Cross-platform model-driven development of mobile applications with md 2." In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pp. 526-533. ACM, 2013.
- [25] Cavarlé, Glenn, Alain Plantec, Vincent Ribaud and Christophe Touze. "Towards agile cross-platform application development with Smalltalk and Model Driven Engineering." In *International Workshop on Smalltalk Technologies*, Cambridge England, 2014.
- [26] Balagtas-Fernandez, Florence T., and Heinrich Hussmann. "Model-driven development of mobile applications." In *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pp. 509-512. IEEE, 2008.
- [27] Thompson, Chris, Jules White, Brian Dougherty, and Douglas C. Schmidt. "Optimizing mobile application performance with model-driven engineering." In

Software Technologies for Embedded and Ubiquitous Systems, pp. 36-46. Springer Berlin Heidelberg, 2009.

- [28] Diep, Chi-Kien, Quynh-Nhu Tran, and Minh-Triet Tran. "Online model-driven IDE to design GUIs for cross-platform mobile applications." In *Proceedings of the Fourth Symposium on Information and Communication Technology*, pp. 294-300. ACM, 2013.
- [29] Parada, Abilio G., and Lisane B. de Brisolará. "A model driven approach for Android applications development." In *Computing System Engineering (SBESC), 2012 Brazilian Symposium on*, pp. 192-197. IEEE, 2012.