The *WL++* Environment for Model-Driven Engineering of Cross-Platform Mobile Applications

by

Blerina Bazelli

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

With the proliferation of mobile devices and the adoption of mobile applications as the de-facto mediators for most daily-life activities, including communication, shopping and edutainment, the systematization of mobile software engineering has become an important research problem. Mobile-application construction must become more systematic, flexible and adaptable, and less costly with reduced time to market. Code-generation techniques based on domain-specific languages present both opportunities and challenges for the construction of such applications. In this thesis, we propose an abstract model to represent catalogue-style mobile applications and a graphical code-generation environment, namely *WL++*, for creating such mobile applications, based on specifications of the application back-end data model and its user-interaction behavior. Our framework enables the rapid development of multi-platform mobile applications, relying on state-of-the-art technologies such as Worklight and Backbone.js. More specifically, *WL++* allows developers to create diagrammatic models of the to-be-generated application's logical model and annotate them with information regarding the user interface widgets used to interact with the model elements. Then, it produces a relational back-end for storing the model data, a set of RESTful APIs for accessing and updating the back-end, and a multi-platform application that relies on the IBM Worklight framework to access the APIs and render the relevant data through the chosen widgets. We describe the *WL++* mobile-app generation framework and we illustrate its functionality with three applications.

Acknowledgements

First, I would like to thank my supervisor Dr. Eleni Stroulia for her valuable guidance throughout these two years of my research journey. With her expertise in the field of Software Engineering, assistance and patience she gave me the necessary motivation to accomplish more goals than I had initially imagined. The Master Early Achievement Prize, travel grants and the several conferences I had the chance to attend are the result of a great supervision combined with hard work.

I would also like to express my gratitude to my family and especially my fiancé Nikos. Through their love, advice, and support I learned to make my own decisions and follow my dreams.

Finally, I would like to thank IBM CAS Research Canada for funding this project.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The shift from utilitarian cell phones for communicating through phone calls and text messages to stylish devices with the capabilities of a personal computer happened in a very short time frame. The ubiquitous availability of these devices has made them the primary mode of accessing information and services on the Internet. According to Pew [32] as of May 2013, 63% of adult smartphone owners (in North America) use their phones to go online and 34% of smartphone Internet users go online mostly using their phones, and not using some other device such as a desktop or laptop computer.

Due to the broad usage of mobile devices (e.g., smartphones, tablets), providers are highly motivated to deliver mobile applications to enable users to access their information and services. The question then becomes choosing a platform on which to deliver these applications. Android and iPhone owners are equally common within the cell-owner population as a whole (Figure 1.1), although this ratio differs across various demographic groups [1]. Given this market picture, application developers are highly motivated to deliver their applications at least on both platforms. If one were to follow traditional software-engineering methodologies, mobile-application development would need to comply with the native programming languages of each target operating system: for instance, Android applications require the knowledge of the *Java* programming language whereas iPhone applications are developed in *Objective C*. Parallel development on multiple platforms implies increased effort, higher cost, and more difficult maintenance and evolution, challenges that get exacerbated by the highly-competitive mobile-app market.

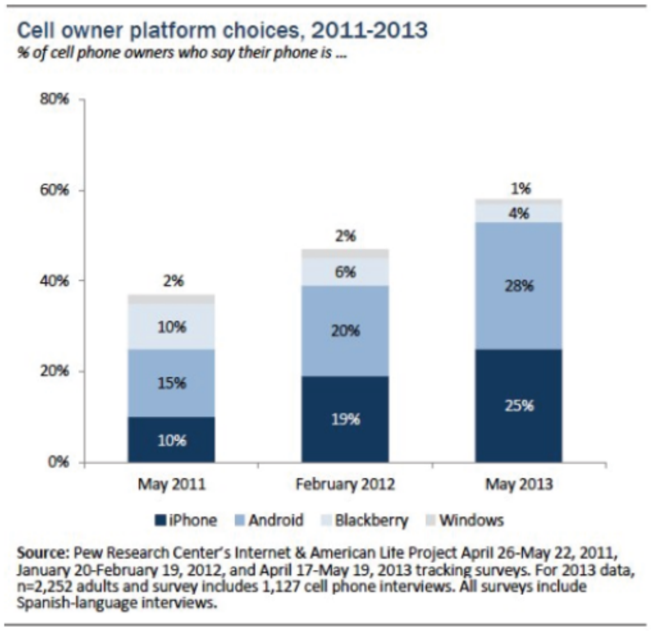


Figure 1.1: Platform Choices (2011-2013) [1]

Technologies such as Phonegap[1] and IBM Worklight[2] offer middleware services on which developers can create dynamic web-based applications that can be compiled into platform-specific code. These applications are in fact dynamic web applications that run on the device's internal browser (web view); they look and feel like native applications due to the aforementioned technologies [33] that enable the use of native User Interface (UI) widgets and access to the device's hardware components (e.g., camera, accelerometer, GPS etc.). Although native applications provide full access to the device's features and offer better user experience in terms of performance [6], cross-platform applications are a valid choice when the application user-interaction model is relatively simple, the application does not need to meet extremely fast response times, and development resources are at a premium.

1.1 Thesis Overview

The recent rise of frameworks such as Phonegap and IBM Worklight, technologies such as HTML5, CSS3 and JQuery Mobile, offer great potential in the field of

[1]http://phonegap.com
[2]http://www-03.ibm.com/software/products/en/worklight

cross-mobile application development. Also, given the number of mobile applications that already exist, one could say that they play an important role in our lives. However, developing mobile applications is a time consuming process and needs developers with expertise. By combining all the aforementioned technologies, we would be able to build a framework that could allow mobile applications developers to create cross-platform mobile applications easily and efficiently.

This work focuses on a broad family of applications namely catalogue-style applications. This category of applications exhibits to have several commonalities such as data insertion, review, deletion etc. Focusing on this class of applications enables us to develop a model of the application data and the services manipulating this data. Consequently, we assume that the data elements are (a) *structured entities* consisting of multimedia attributes, and (b) *collections* of these entities. We further postulate that the most important (and typically desired) features for these applications are (a) *creation* of new entities to be added to application repository, (b) *retrieval* of (potentially escalating) entity details given an entity identifier, (c) *faceted search* (given a combination of entity properties and desired value ranges for them return a collection that matches the input criteria), and (d) *entity-state manipulation* through RESTful APIs accessing and updating selected instances.

Roy Fielding, in his dissertation [10] explained REST (Representational State Transfer) as an architectural style that relies on a stateless and simple client-server communication. REST takes advantage of the existing HTPP protocol and client applications may send HTTP requests through this protocol to a RESTful web service in order to create, retrieve, modify, or delete data. These actions are performed through predefined methods namely GET, POST, PUT and DELETE. Moreover, a RESTful web service relies on resources which are conceptual entities represented by nouns. The aforementioned methods may change or not an entity. For instance, the GET method only retrieves entities and therefore cannot change its contacts whereas the POST, PUT, and DELETE methods are able to change existing entities permanently. Other architectural styles include DCOM (Distributed Component

Object Model)[3], CORBA (Common Object Request Broker Architecture)[4], SOAP (Simple Object Access Protocol)[5] to replace DCOM and CORBA, RPC (Remote Procedure Call)[6] etc. A RESTful service is usually preferred especially when it comes in messaging frameworks for mobile web services; a REST mobile web server (MWS) outperforms a SOAP MWS, in terms of server utilization and request waiting time [2], and in terms of efficiency and scalability [26]. Furthermore, RESTful services allow the transmission of different data formats, including JSON, which is more lightweight than XML which is typically transmitted through SOAP.

1.2 Contributions

To support the systematic development of such multi-platform applications, first we propose an abstract model describing catalogue-based applications and second, a code-generation environment integrated within the Eclipse IDE namely *WL++*. The target audience of this framework is developers. Therefore, *WL++* aims in assisting developers in generating the application's front-end user interface, its local and back-end storage and the RESTful APIs necessary for the above functionalities. Furthermore, it is also capable of generating mobile clients that can be deployed in multiple platforms based on existing back-ends. The plugin user interface is composed of an editing area and a set of predefined widgets. Through drag and drop actions of widgets into the editing area users are capable of creating links among components in order to build the proper diagrams that describe the application's architecture. Then, the applications diagrams are converted into source code, in a model-driven engineering process. Users are also able to either generate mobile applications that are deployed locally or applications accompanied by back-ends. Furtheremore, *WL++* can take as input an existing back-end service in the form of the APIs it uses to access the back-end and construct the modeling part of the application diagram in a reverse engineering process. Finally, the code-construction

[3]http://technet.microsoft.com/en-us/library/cc958799.aspx
[4]http://www.corba.org/
[5]http://www.w3.org/TR/soap/
[6]http://technet.microsoft.com/en-us/library/cc787851(v=ws.10).aspx

framework includes a general monitoring service that observes the API calls and the data exchanged between the application and the back-end server. This data is optionally stored into a database in JSON format, in order to be consumed by a downstream domain-specific analysis-and-recommendation algorithm. The monitoring component could be used to forward recommendations as push-notifications to the mobile application. Consequently, this work makes three important contributions:

First, it proposes a modeling language for specifying the data models and the user interaction of catalogue-style applications.

Second, it offers an integrated environment for enabling developers to specify and generate their applications.

Third, it evaluates this model-driven methodology with three example applications: a contact manager application to store contacts information, an application to keep track of one's daily physical activity and an application operated by nurses when visiting their patients.

1.3 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 reviews the background of this work, in terms of the technologies that have given rise to the popularity of mobile web-based applications and the related research on systematizing the development of mobile applications. In Chapter 3, we describe the architecture and the tools based on which the framework was built. We also present the features of *WL++* and how the input application model is translated into the desired source code. In Chapter 4 we describe the process of generating a contact manager application by using the *WL++* application specification plugin. The process of generating two additional mobile applications (Physitivity and Hourly Rounds) is described in detail in Section 5. Finally, we conclude with a discussion about our future plans in Chapter 6.

Chapter 2

Literature Review

In this chapter, we discuss the related research by starting with an overview of the differences between native and cross-platform mobile applications (Section 2.1). Then, we introduce and compare the existing frameworks used to deploy cross-platform applications into different platforms (Section 2.2). Next, we describe the Model-View-Controller pattern and highlight the importance of applying it when developing web applications (Section 2.3). Finally, we introduce the concept of Domain Specific Languages (DSLs) and mention several popular DSL designing tools (Section 2.4). Finally, a discussion about code generation tools (Section 2.4.3) follows along with several projects motivated by the objective to support the development of both native and cross-platform mobile applications (Section 2.5).

2.1 Native & Cross-Platform Mobile Apps

Large software providers, such as Google, Apple, and Nokia to name a few, have been competing against each other in offering Software Development Kits (SDKs) to support developers in building mobile applications on their platforms. Typically, each one of these SDKs includes a device emulator for testing purposes. An SDK assumes expertise with a single programming language and produces mobile applications that can only be deployed on the corresponding platform. For instance, to develop applications targeting the Android platform, developers should download the Android SDK and be fluent in Java. On the other hand, the development of iPhone/iPad applications requires the download of the Xcode IDE and the knowledge of the Objective-C programming language. This means that developers dedicate some cognitive effort in order to become familiar with its underlying programming model. Although, native applications are able to take full advantage of all the device's features and thus potentially offer a better user experience, they are fundamentally limited in their deployment scope and consequently in their adoption.

The operating system market share changes rapidly and therefore it is difficult, practically impossible, to predict which operating system(s) will become more prevalent even in the near future [17]. It should also be noted that the operating systems' market share may change per region making the decision of what operating systems to support even harder. With that being said, in the case where an application aims to be used by a large population, ideally it should be developed to run on at least the top three or four dominant operating systems.

However, developing an application for multiple operating systems takes a significant amount of time as the application must be developed and tested multiple times. It also requires either developers having multiple skills (*i.e.,* able to program in different programming languages) or several teams of developers working on each application separately. To make matters worse, multiple applications are more difficult to be maintained. Finally, when there is a need for an update, changes should be implemented and send to all the operating systems separately whereas in the web-based applications case changes are implemented much faster and updates are sent to all the devices at the same time regardless their operating systems.

It is only relatively recently that cross-platform development tools, such as Phonegap[1], became able to access the hardware of mobile devices, such as accelerometer, camera and GPS. This ability has been instrumental in the adoption of the cross-platform mobile-application development practice. At the same time the overall functionality and capabilities of these tools have been continuously improving. One dimension of improvement arises from the fact that the underlying technologies leveraged by these tools, (*i.e.,* HTML5, CSS3, and JavaScript) are continuously improving. In addition, most cross-platform tools, including Phonegap, can be extended with special-purpose plug-ins, written in the native programming

[1]http://phonegap.com/

language of each target platform, to leverage each platform's specific features. The Phonegap community has been very active in this regard and a wide variety of plug-ins have already been implemented. As an example, there is a plug-in for Android devices which produces native notifications; the generation of these notifications and their context are controlled by Phonegap in JavaScript. Finally, substantial progress has been made in developing user-interaction extensions that imitate the look-and-feel of purely native applications, (*i.e.,* JQuery Mobile[2], Sencha Touch[3], and app-UI[4]) and techniques to improve the application responsiveness (*i.e.,* with image preloading, and hardware acceleration).

2.2 Mobile Application Construction IDEs

In Table 2.1 we present a list of cross-platform mobile application tools, with Phonegap, Rhomobile and Appcelerator being the most popular among them [19]. While Phonegap and Rhomobile follow a similar philosophy, Appcelerator differs significantly in terms of the way it accesses native features through a JavaScript API and the type of the generated application (web-based vs. native). IBM Worklight is a full mobile application platform built on top of Phonegap. The advantages of Worklight over Phonegap include the application server, a mobile-browser simulator and a set of adapters used to achieve the communication of an application with a back-end. The commonality among the first three listed frameworks is the fact that the final application is a web-based application although they offer the ability to generate hybrid and native applications by using programming languages specific to each platform (e.g., Phonegap applications can take advantage of native features through plugin developed in the targeted platform language). On the other hand, as mentioned before, Appcelerator Titanium, MoSync, and ifactr (based on Xamarin), produce as output a native application. In fact, although Appceletator supports four platforms to date, the way it has been developed makes the support of more platforms harder.

[2]http://jquerymobile.com/
[3]http://www.sencha.com/products/touch
[4]http://triceam.github.io/app-UI/

Framework	Mobile OS Support	Language	IDE	Open Source Licence	Generated Application
Phonegap	iPhone, Android, Blackberry, WebOS, Windows Mobile, Symbian, Bada	HTML, CSS, Javascript	Depending on the IDE supported by the OS	Apache Public, Licence v2	Web-based
Rhomobile	iPhone, Android, Windows Mobile	HTML, CSS, Javascript, Ruby	RhoStudio	MIT	Web-based
IBM Worklight	iPhone, Android, Windows Mobile, Blackberry	HTML, CSS, Javascript	Eclipse	No	Web-based
Appcelerator Titanium	iPhone, Android, Windows Mobile, Blackberry	Javascript	Titanium Studio	Apache Public, Licence v2	Native
MoSync	iPhone, Android, Windows Mobile, Blackberry, Symbian	C/C++	Eclipse	GPL v2	Native
ifactr	iPhone, Android, Windows Mobile, Blackberry, Symbian	C#	Visual Studio	No	Native

Table 2.1: Comparison of popular cross-platform mobile frameworks based on the OS they support, developing language, IDE, Licence and the type of the output application

Cross-platform applications require less effort and fewer resources than native applications for their development and maintenance; consequently they tend to be less expensive than their native counterparts. Furthermore, technologies such as HTML, CSS and Javascript are widely used and therefore it is much easier to find skilled developers who are familiar with them that it is to find developers with expertise in each native technology stack. These are the main reasons why multi-platform application development is beginning to emerge as the more popular mobile-application development paradigm.

2.3 The Model-View-Controller Pattern

Without using the right tools and applying the proper architectural patterns usually web applications become difficult to debug and maintain as the code grows. This usually leads to less efficient writing and repetitive code. The Model-View-Controller (MVC) concept was introduced in 1988 as an extremely useful software engineering pattern used to build user interfaces in Smalltalk-80 [21].

According to the MVC architectural pattern, any application (desktop, mobile and/or web-based) can be structured into three main components (Model(s), View(s), Controller(s)). The application's **models** represent the application's data. **Views** are responsible for the data to be displayed. A view may be composed of UI components such as buttons, input forms, date fields etc. and allows users to interact with the application. Moreover, multiple views can be developed as nested sub-views within a single view. This is considered as a good practice as it allows smaller views to be reused within the same or across applications. Finally, **controllers** handle the user input and update the proper models when necessary.

Although the concept remains the same, there is a variance of implementations of the MVC pattern. In the Active Model MVC pattern, when the model is changed, it notifies the proper views whereas in the Passive Model MVC pattern, the model does not take any action and the Controller is responsible to notify the views when a model changes. In the first case, we have the Observer Synchronization according to which multiple views share a single data model. The View acts as an Observer to the Model and therefore, when the Model is updated, the View is notified regarding the change and then requests the new data [13]. In the second case, the controller needs to have a reference of the views to be updated (Flow Synchronization) [12].

When the MVC pattern is not applied in the development stage of an application the aforementioned objects tend to be tightly coupled. There are significant benefits regarding the MVC pattern. First, it decouples views/models by allowing the controller taking care of events that may indicate data changes and thus, notify the proper views to change as well by displaying the modified data. With that being said, MVC also allows the development of multiple user interfaces without the need of modifying the business logic. The code is easily debuggable and maintainable as although user interfaces may change over time the main functionality of the application usually remains the same. Furthermore, by applying the MVC pattern, a project could be developed by a team of developers where small groups could work on the User Interface, Models and Business logic (Controller). This technique allows teams to develop a piece of software without the need to know all the requirements from the initial phase of the project. For instance, the business logic can be developed even if the user interface requirements are not fully delivered.

2.3.1 Model-View-* Frameworks

As mentioned before, there is a wide variety of patterns such as MVC, (Model View View Model) MVVP, (Model View Presenter (MVP). Generally, the Model and View components remain the same across all these patterns. The last component however, may be a Controller (e.g., MVC), a Presenter (e,g., MVP) or anything that works best for the developer. In this case we have the Model View * (MV*) pattern. JavaScript frameworks do not usually follow the MVC pattern in its strict definition and therefore they belong to the category of MV* frameworks.

There are several such MV* client side JavaScript frameworks/libraries (e.g., Angular.js, Backbone.js, knockout.js, Ember.js) developed for single page web applications. Angular.js is also referred as MVW where W stands for "whatever works for you" [24]. However, depending on the application to be developed some frameworks may be more suitable than others. Generally, before deciding on what framework one should use the following factors may be considered: 1) the amount of AJAX calls (communication) with a back-end, 2) the structure's similarity with the back-end (a RESTful service composed of classes that may represent the models on the client side), 3) event-binding (views are notified on model change events and automatically adapt), 4) dependence on other libraries, 5) documentation and strong community that could support developers, 6) evolution and improvement over time (e.g., bug fixes, new features). McKeachie [23] compared the most popular JavaScript MV* frameworks in terms of maturity, community, size, dependency etc. The maturity of a framework is usually determined by factors such as the number of users, the documentation (how often it is updated), how stable is the API etc. Taking into account these factors, Backbone.js seems to be the most mature framework followed by Angular JS, Knockout and Ember.js. From the community perspective McKeachie considered as a good factor the GitHub watchers indicator. Based on that, the most established frameworks are Backbone.js and Angular JS. Another factor that may play a role when deciding on which framework fits one's needs is its size (since these frameworks may reside on the client side, the loading time is important as a framework with few lines of code is loaded much faster). Some frameworks have strong dependencies on other libraries whereas others not (while allowing the user to import the libraries of their own preferences) and therefore are more lightweight. In Figure 2.1, Backbone.js, Spine, Knockout and CanJS are only a few kilobytes and that is the reason why they are usually referred as libraries and not frameworks. On the other hand, Ember, Batman and Angular.js are considered more as frameworks rather than libraries.

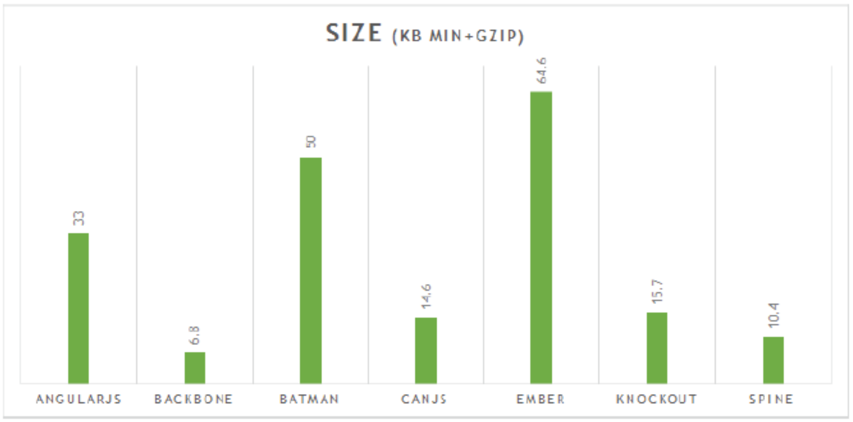


Figure 2.1: Comparison of MV* frameworks/libraries based on their size in kB [23]

2.4 Model-Based Code-Generation Environments

In this section we briefly explain what a Domain Specific Language [37] is and what are its benefits over a General Purpose Language. Then, we mention few Model-To-Code code generation environments such as XPand, Acceleo and Xtext and explain how the models defined by developers are translated into source code.

2.4.1 Domain Specific Languages

A Domain Specific Language (DSL) is a language focused on a specific domain space [37]. Therefore, it is only worth using it when solving a problem that belongs in the domain that the DSL is designed for [11]. Domain specific languages have become very popular. The CSS markup language, HTML, SQL and MATLAB are few examples of the plethora of DSLs that exist and are broadly adopted and used nowadays. In order to design a DSL, it is important first to analyze in detail the application domain by extracting all the features (common and less common (application specific)). Defining a new DSL has advantages and drawbacks; first, although DSLs enhance quality, productivity, reliability, maintainability, portability and reusability of the code, there is a cost regarding the designing and the implementation as well as testing the DSL in multiple case studies.

2.4.2 DSL Designing Tools

Tools that support the development of DSLs can come in a form of either textual syntactic editors or graphical editors. Some textual syntactic editors include Xtext[5], EMFText[6] and MPS[7]. These tools allow developers to define the text syntax for their own DSLs from an ECore[8] metamodel and offer several features such as syntax coloring, value validation and quick fixes proposals. Apparently, these editors have not been developed as standalone but are usually tailored as external plugins to existing IDEs such as Eclipse (Xtext, EMFText) and IntelliJIDEA[9] (EPS).

Besides the textual editors through which users can describe an application model via a DSL, there are many Graphical Editors (e.g., GMF, MetaEdit+, AToM, Spray) that support the description of application models through a graphical representation of the specific DSL's features. There have been efforts to compare popular graphical DSL editors; however, the results vary as each graphical editor exhibits to have advantages/disadvantages over other competitors. Beatens et.al. [3] compared three DSL graphical editors (GMF, MetaEdit+ and AToM) in terms of their architecture and development process. Both MetaEdit+ and AToM are standalone applications whereas GMF is an Eclipse plugin. Also, AToM and GMF are free and can be adopted mostly by researchers whereas MetaEdit+ is offered through a license and therefore is more suitable for industrial purposes. For our framework we chose to use GMF as it is an open source tool and can be used within the Eclipse environment (which we were already familiar with). Other studies [38] revealed that Spray would be a great solution for research purposes because it is open source and has a strong community that supports and keeps developing it. Furthermore,

[5]http://www.eclipse.org/Xtext/
[6]http://www.emftext.org/index.php/EMFText
[7]http://www.jetbrains.com/mps/
[8]http://www.eclipse.org/modeling/emf/?project=emf
[9]http://www.jetbrains.com/idea/

Spray is also built on top of EMF just like the GMF tool. On the other hand, MagicDraw may be a good solution for industrial environments. It is offered through a license, is well-documented and has an easy-to-use user interface.

Finally, tools for developing programming languages and Domain Specific Languages include Xtext[10] and tools to generate code based on the DSL include Xtend[11] and Acceleo[12]). The aforementioned frameworks/toolkits use an editor that requires the user to insert several commands based on the DSL specified in order to generate the desired output. The translation of these commands to actual code (output) is done through tools such as Xtend and Acceleo. Xtend also offers a simplistic way to write Java code which later is transformed into actual Java code. However, both Xtend and Acceleo are able to perform model to text transformations. More specifically, they rely on templates which a developer created and code is injected in these templates according to the application models. They may get as input an Ecore Metamodel and generate any type of files (.java, .txt, .html etc.) according to the templates.

2.4.3 Code Generators

Model Driven Engineering techniques aim to generate code from a high level model which is described in a DSL. Code Generation Environments serve as tools that take as input the high level model (either in a textual or diagrammatic form) and generate the desired code. However, the type of the output (generated code) can only be determined by the developer and therefore, code generation tools work based on templates. For instance, we suppose that we have developed a DSL for an online shopping website. To represent a product, we use two attributes: name and price. We define such a model using the following syntax:

```
Product[
  name:String,
  price:Double
]
```

[10]http://www.eclipse.org/Xtext/
[11]http://www.eclipse.org/xtend/
[12]http://www.eclipse.org/acceleo/

Then, the developers decide that the above code must be transformed into a Java class. Therefore, through specific rules, the components of the DSL must be associated with the desired components of the to-be-generated Java class through a template which would be represented by the code snippet below:

```
class <<this.name>>{
 <<FOREACH attribute AS attr ITERATOR it>>
   private <<attr.type>> <<attr.name>>;
 <<ENDFOREACH>>
}
```

The output then would be a Java class according to the template along with the information retrieved from the instance model defined in the DSL:

```
class Product {
 String name;
 Double price;
}
```

In order to perform model to code transformations, one would need a code transformation language tool such as XPand[13], Xtend[14] or Acceleo[15]. XPand and Acceleo are offered through the Eclipse Model To Text Project (M2T). All those tools are able to parse the model and based on specific templates to give the desired output.

2.5 Mobile Application Development Tools

There are several mobile app development tools where each one has different goals to accomplish. Cabana for example is a multi-platform mobile development system developed to be used by students in introductory computer-science courses [9]. Students are able to create cross-platform mobile applications by modeling them as a set of nodes, connected to each other. The nodes represent screens, buttons and custom code modules responsible for the applications' logic, which must be implemented in JavaScript. Although, Cabana meets its educational/pedagogical

[13] http://www.eclipse.org/modeling/m2t/?project=xpand
[14] http://www.eclipse.org/xtend/
[15] http://www.eclipse.org/acceleo/

objectives, its features do not provide sufficient support for the development of real applications. First, its user-interaction model in terms of a screen-navigation graph is limited and makes the development of complex entities difficult. More importantly, however, it does not support the generation of back-ends, which is an essential part for the majority of mobile applications.

The work closest to ours is MD^2, a model-driven approach for cross-platform application development [16]. MD^2 defines a domain-specific language in terms of which the to-be-generated application is represented as a set of nested models. The language was developed in Xtext and therefore, the tool used to generate the code is Xtend. Then, source code is generated for Android and iOS platforms through the Android and iOS engine respectively. Furthermore, a back-end (Java Service) is generated for the applications with capabilities of simple CRUD (Create Retrieve Update Delete) operations. This gives the ability of communication between the application and the local/remote back-end. The MD^2 framework is clearly motivated by similar objectives as our work. However, it adopts a different code-construction methodology that involves two different compilers for the two target platforms. This decision implies that MD^2 covers fewer platforms than our framework, requires more complex application modeling, and can potentially result in better performing applications with more varied and versatile look-and-feel.

On the other hand, our *WL++* framework focuses on a particular class of applications, with fairly simple interaction constraints which can be met by adopting a common middleware layer that can be compiled into more platforms. The end users of our tool are developers; we therefore took into consideration the fact that the generated code must be understandable and maintainable over time. To that end, *WL++* adopts Backbone.js[16], a JavaScript framework that enables the clean separation (and smooth integration) of user-interface from (with) business-logic concerns. Also, the generated source code includes comments to assist developers in making further changes. We believe that the capabilities of cross-platform frameworks have been expanded and therefore applications generated by these tools are becoming more competitive to native ones. Besides, the code base that is shared

[16]http://backbonejs.org/

among different platforms (Android, iOS, Windows) makes them popular by targeting larger groups of users. Finally, through *WL++*, developers can also generate mobile clients based on existing back-ends, a functionality that is missing from MD^2.

Ribeiro et al. [29] developed a DSL for Mobile Applications, namely XIS-Mobile. Following the MDD approach, they extended the XIS UML profile project through which developers can model web applications [8]. According to the XMI Mobile approach, a native application can be generated by defining four major views: Entities View (that represent the entities of the application), UseCases View (responsible for the operations that could be performed on the data), Architectural View (allows the connection of the application with external entities/services) and the User-Interface View (where navigations and user interface components are being defined). Through the XIS-Mobile language, simple native applications targeting the Android and the Windows Phone platforms are generated. In their work, the evaluated their language by developing a To-Do List application. However, this work is still in its very initial steps.

Botturi et al. [4] tried to solve the problem of interoperability of mobile application by extracted the common features of several mobile application platforms (Android, Windows Phone, iOS). Therefore, they attempted to design a single version of an application that could eventually be translated into the a platform specific application (e.g., Android) by using Model-Driven engineering techniques. So first, developers need to specify the structural, behavioural and navigational aspects of the to be generated application. Then, transformation rules map each platform independent component to a platform specific one. Therefore, the generated application is also a native application and several transformation rules for each platform should be created.

Currently, there is a plethora of mobile applications available in the market. The available SDKs allow the community to develop applications at a rapid pace. Gasimov et al. [14] examined the existing mobile applications and categorized them into four major categories based on their purpose: (a) transaction, (b) content dissemination, (c) social networking, and (d) personal productivity and leisure based. Moreover, they mentioned the main challenges regarding the development of mobile applications (power consumption, cross-platform compatibility and software quality) and proposed the idea of standardization of mobile hardware/software protocols [18]. Our *WL++* framework is generally targeted towards personal productivity and leisure applications and has been inspired by two different mobile applications that our group had already developed, before embarking in the effort to systematize mobile-application development with a framework. More specifically, we have already implemented two multi-platform self-management mobile applications for e-health, namely *EASI* and *Physitivity* [36], and it is on the basis of this experience that we are developing *WL++*.

Chapter 3

The *WL++* Application Construction Framework

In this chapter we discuss the software architecture of *WL++* and how this architecture emerges from the three major components it integrates. The first component is IBM Worklight, a comprehensive cross-platform mobile-application development and deployment environment (Section 3.1). The second component is Backbone, a framework that defines the organization of the produced mobile applications, in a manner that should facilitate the process of extending and maintaining them in the future (Section 3.2). Finally, the third component is our own *WL++* application modeling plugin, that relies on the Graphical Modeling Framework (GMF) for designing and constructing mobile applications. In Sections 3.3 and 3.4 we explain the metamodel that led to the construction of the *WL++* plugin and the clusters of components it includes. Next, we present the code generation engine of *WL++* along with the templates and external libraries used to generate cross-platform applications with supporting back-ends (Section 3.6). Section 3.7 and Section 3.8 describe the back-end generated and the code structure of the mobile application respectively. Finally, Section 3.9 lists the steps to be followed in order to generate a mobile application from an existing back-end through a reverse engineering process.

3.1 Cross-Platform Applications with IBM Worklight

IBM Worklight is a tool designed to support multi-platform application development. To that end, it provides an intuitive structure for cross-platform applications. We chose the Worklight IDE as it is a full mobile application platform leveraging a powerful Javascript library (Phonegap) through which it accesses native features of the mobile devices. Phonegap has been broadly adopted by cross-platform mobile application developers as it supports a wide variety of platforms. The IBM Worklight framework is composed of several components:

- **Worklight Studio**

  It allows the development of native, hybrid and standard web mobile applications. The Worklight framework is built on top of Phonegap, a Javascript library also known as Cordova. Therefore, it supports all the platforms supported by Phonegap (e.g., Android, iOS, Windows). Phonegap is a well documented library that allows the access to devices' APIs through native code and there are already plenty of open source mobile applications developed based on that. Worklight Studio also offers a "What You See Is What You Get" (WYSIWYG) editor, which allows developers to preview the HTML pages before they are complied in a real browser.

- **Worklight Server**

  It allows the communication between the mobile application and the remote back-end by using adapters. These adapters belong to the server-side of the application and serve the purpose of sending/retrieving data from back-ends from/to the client application. Adapters are composed of several procedures (functions) being invoked by the client application when a communication with a back-end is required. Then, the back-end returns the data which is received within the adapter and finally transmitted in the client application (Figure 3.1). Worklight gives the option of choosing among four different adapters: (a) HTTP adapter, (b) SQL adapter, (c) Cast Iron adapter and (d)

JMS adapter. The HTTP adapter is used in cases where RESTful or SOAP services need to be invoked whereas the SQL adapter is used when there is not a middleware between the application and the external database. Therefore, it is developed to execute SQL queries in order to perform operations such as insert, update, delete etc. Finally, the *Cast* Iron can be used to retrieve data from enterprise data sources and the JMS adapter allows the communication with a JMS-enabled messaging provider.

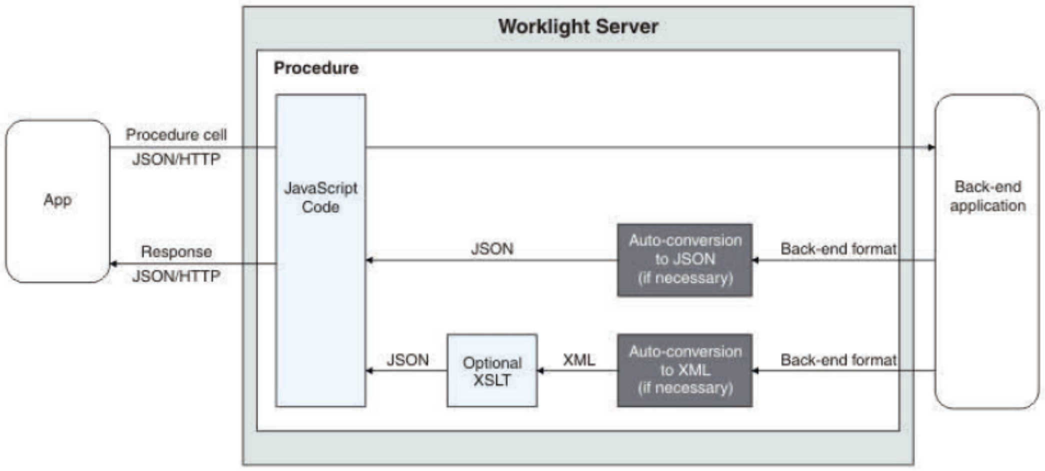


Figure 3.1: The role of adapters in client-server communication [15]

- **Worklight Device Runtime Components**

  Although Phonegap offers the ability to access native device features across multiple platforms through a single code base (HTML5, CSS3, JavaScript), IBM Worklight goes a step further by providing significant advantages over using only the Phonegap library to build cross-platform mobile application. More specifically, Worklight integrates security features by having an on-device encryption and offline authentication. Also, the run time skinning feature allows developers to adapt the application in run time accordingly depending on the mobile device it runs on (in terms of operating system, screen resolution etc.).

- **Worklight Console**

  Through the Console, developers can view their deployed applications along with their environments. By adding a new environment developers are able to view the application through the browser on a web simulator associated with the added environment (Android, iOS etc.) (Figure 3.2). Each simulator has different properties such as device's dimensions and therefore developers can use these properties to fully test their application on different devices. Furthermore, developers can also test the Cordova API (accelerometer, camera, geolocation, network etc.). For instance, to test the accelerometer in case of an accelerometer implementation developers can set the proper values through the console and then check the simulator's response. Finally, the console allows the addition of several simulators at the same time side by side which is useful for comparison purposes.

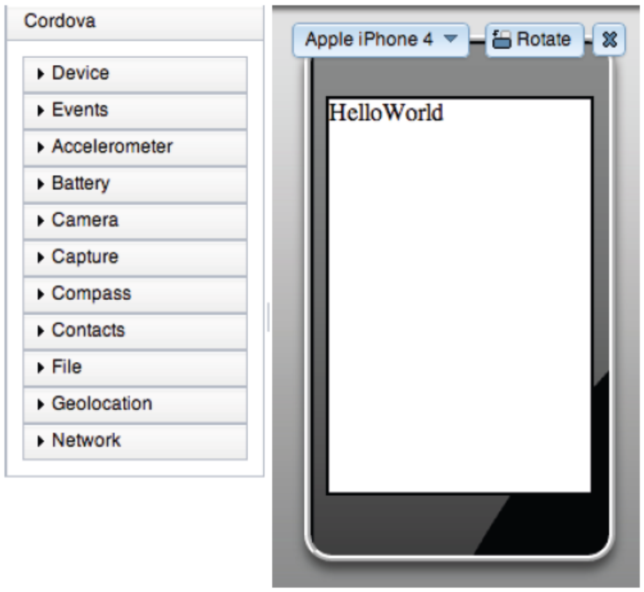


Figure 3.2: The Worklight Mobile Browser Simulator

- **Application Center**

  It provides an enterprise application store for sharing applications across an organization for use in that organization.

Resources common to all platforms exist in a common directory. Platform-specific optimizations and skinning can be done on a per-platform basis, through environment-specific directories. The contents of the environment-specific directories are integrated with those of the general-purpose directories in order to compile platform-specific applications. At compilation time, platform-specific CSS and Javascript assets are concatenated with the corresponding platform-independent ones, and platform-specific HTML files and images replace similarly named platform-independent ones. This model is dictated by the interpretation rules of the corresponding languages. In CSS and Javascript, subsequent definitions replace existing ones; therefore concatenation ensures that platform-independent specifications will be applied as long as no platform-specific ones have not been defined. This is not the case with HTML, and this is why asset replacement is required. This latter behavior, however, implies that HTML templates need to be defined at both levels in our case will be merged, not replaced. Only, templates with the same name that are redefined will be replaced.

3.2 Modular User Interfaces with Backbone

Backbone is one among a new breed of JavaScript tools conceived to superimpose a conceptual structure, akin to the model-view-controller pattern, to web applications through *models*, *collections*, and *views*. With this structure, it is possible to construct a set of reusable components that can be utilized by multiple applications. The *model* construct supports key-value bindings and custom events. Essentially following the inversion-of-control pattern, when the state, i.e., value, of the model changes, interested views, i.e., views associated with the model key, get notified. Closely related is the *collection* construct: collections group models and provide a API for model manipulation. Additional functionality can be added to both models and collections. Backbone also provides the capability to map models and collections to a RESTful interface, thus supporting the archiving of the model state in back-end repositories. In addition, in *WL++* we have added functionality to map models directly to storage locally on a mobile device.

The Backbone-relational library supports the creation of multiple models and collections, with explicitly defined relations to one another, that may be reused across all *WL++* applications. Each model also contains domain-specific attributes and functionalities, which are easily modified while maintaining the same structure of relations. In this manner, we are able to effectively design and develop reusable logical models for the application family, rather than reusable individual data structures, which is what the Backbone models effectively are.

At the presentation layer Backbone provides the *view* construct. Typically, a view is associated with a model (or a collection) and an element on the HTML page, and it is responsible for rendering the contents of the (collection of) model(s) (Figure 3.3). Views also declare events and the logic to handle those events. Each view is typically associated with an HTML template, and may contain sub-views which independently handle their own content. In our architecture, nested views are created for the existing models to be displayed, and typically for the members of a collection.

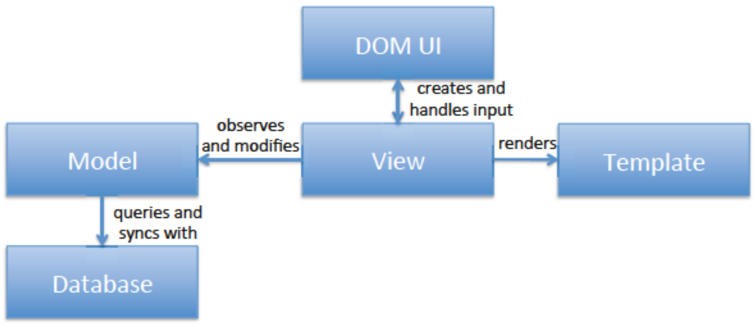


Figure 3.3: The interaction among Backbone components [31]

The *router* handles the URL changes through hash tags. In order to navigate from one screen to another, the old view must be destroyed (it is a good practise to destroy old views in order to avoid memory leaks) and the new view must be initialized. When an event occurs, such as a button click for example, then the hash tag may change. Every hash tag is mapped to a function within the router which initializes the new view with the proper model or collection to provide the data needed from the template to be rendered (Figure 3.4). Backbone depends on the underscore.js library that serves as a templating engine used to generate the HTML to be injected into the DOM. However, Backbone allows the use of other templating libraries as well such as handlebars.js.

```
app.Router = Backbone.Router.extend: {
    routes: {
        url_1 : function_1,
        url_2 : function_2,
        ...
        url_n : function_n
      },

    function_1 : function{
        var view = new View({
          model : model
          collection: collection
        });
      }
    ...
  }
```

Figure 3.4: The Backbone Router

3.2.1 Backbone Templating Engine

Typically, in each Backbone view a single (or multiple) template(s) are compiled using the underscore's library command *_template('template to be compiled')*. Every template is composed of the text that is displayed without modifications, essentially the generic and reusable text, and multiple variables (each one specific to each particular application). When a template is compiled, these variables take the value passed to the template as a parameter. By using the *underscore.js* library developers can declare variables within their templates by using the delimeters of their choice ("%" in our case). Whenever the compiler identifies a variable, it replaces it with the proper value that usually is extracted from the models and/or collections of models. All the templates are defined as multiple scripts of type "template" within usually a single HTML file and are rendered whenever the template belongs to the view to-be-rendered. Because the views differ across applications, these scripts are generated through the application specification plugin based on the views, models and their attributes.

In Figure 3.5 an example of such a template is presented. As we can see there are two variables declared: "data.name" and "data.age" (the "data" variable is a JSON object) (Figure 3.33a). Also the script includes HTML tags that will be parsed and interpreted by a web browser. The template is rendered through the

```
<script type="template" id="tmpl">
<div>
  <h2>Name: %data.name%</h2>
  <h2>Age : %data.age%</h2>
</div>
</script>
```

(a) Template to be compiled

```
var template = _.template($("#tmpl").html());
$el.append(template(data));
```

(b) Compiling and adding the template into the DOM while passing the data object as a parameter

```
<div>
  <h2>Name: Tom </h2>
  <h2>Age : 23 </h2>
</div>
```

(c) The HTML added into the DOM

```
   Name:  Tom
   Age  :  23
```

(d) The text to be displayed

Figure 3.5: An example of an underscore template

underscore function *_.template()* and then the “data” object is passed as a parameter (Figure 3.33b). After the proper data is injected into the variables, the output can be appended into the DOM (Figure 3.5d).

3.3 The *WL++* Metamodel

In this section, we first describe the logical model of catalogue-style applications and how this model corresponds to specific Backbone components. Then, we define the metamodel necessary to build our application specification plugin.

3.3.1 The Logical Model of *WL++* Applications

The logical model underlying the design of *WL++* application framework is conceived to support many of the features one would expect from a catalogue-style application. Users can record entries of different types, which can later be modified or deleted. These entries are persistent on the device, and can be archived at the server side through a RESTful interface for remote storage [7]. Past entries can be displayed as a list where each list item allows the editing of the entry. Moreover, the list is filterable based on properties of the data model. This architecture also allows many points of customization, such as, for example, to the attributes of the models and the templates of the views. Limited coupling between the components of the system allow for views to be rearranged and customized as necessary.

The development of families of applications through the adoption and refinement of architecture frameworks in specific contexts is an established software-reuse practice. In principle, the objective in designing and developing application frameworks is (a) to implement general behaviors that many application instances share in reusable components, and (b) to design a process for systematically and consistently specializing and integrating these components in the context of specific applications. The Model-Driven Engineering (MDE) methodology advocates the definition of abstract models, capturing the behaviors of a family of related applications, and their systematic transformation into executable and deployable assets [20, 30]. Following this paradigm, the process of generating a software system is faster and less error-prone as the code generated has been previously tested. Furthermore, as models represent abstractions of parts of the systems, it is much easier to be understood by users/developers. According to Weigert and Weil [40] the application of model-driven engineering methods has led to multiple benefits such as productivity and quality improvement. By concealing low-level code details through the use of models results into more stable and reusable models throughout the system. In addition, inspections on the generated code have become sparser due to the less error-prone auto-generated code.

In Figure 3.6 we can see the conceptual model of the *WL++* application family, composed of the major Backbone components (models, views, and navigations). The core of the application model is the Application itself, which consists of models, views and adapters. Each model may have multiple representations such as text, location, date, image etc. On the other hand, a view may be either a dashboard, a form view, a list view or a login view. These views are associated with models

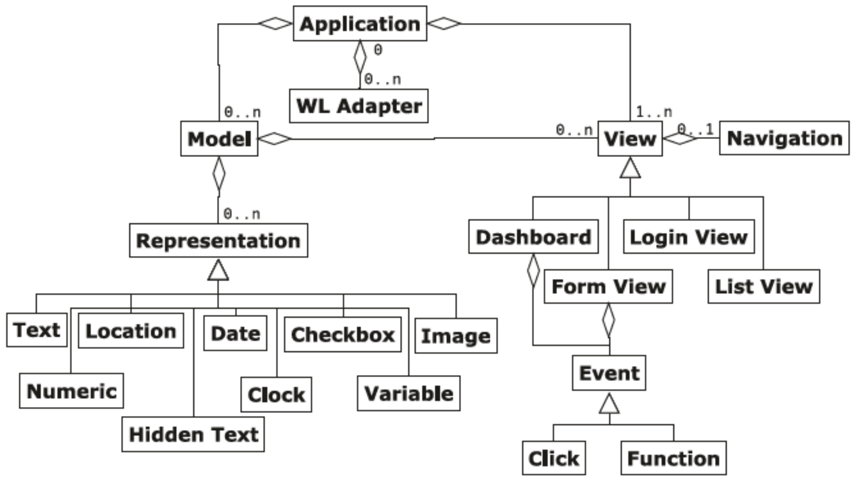


Figure 3.6: Conceptual Model of catalogue-based applications

which represent the data to be rendered and displayed to the end-users. Furthermore, since there is no controller in Backbone, views are also responsible for event handling. For this purpose, the "click" event component is available through which developers can add "click" events and specify the actions to be performed when such an event is triggered. Also, a "function" component represents a customized function that can be implemented manually by the developer later on.

3.3.2 The Metamodel Definition

Based on the above conceptual model, we constructed a metamodel using the Emfatic language[1]. Emfatic is a language that has been developed to represent in a textual form Ecore models by using a special syntax. After we defined all the components of the application's model, we generated the Ecore Metamodel by using the EuGENia[2] tool. The Ecore metamodel diagram is depicted in Figure 3.7. Through the Emfatic language, we can create the basic objects to be displayed as widgets in the application specification plugin's user interface and also allow the permitted connections among these objects. Currently, there is a basic set of widgets that are essential in the development of a complete mobile application. These widgets are separated in six clusters as shown in Figure 3.7: (a) Objects, (b) Connections, (c) UI Components-Variables (d) Views (e) Events and (f) Back End. The *Objects* cluster

[1] http://www.eclipse.org/epsilon/doc/articles/emfatic/
[2] http://www.eclipse.org/epsilon/doc/eugenia/

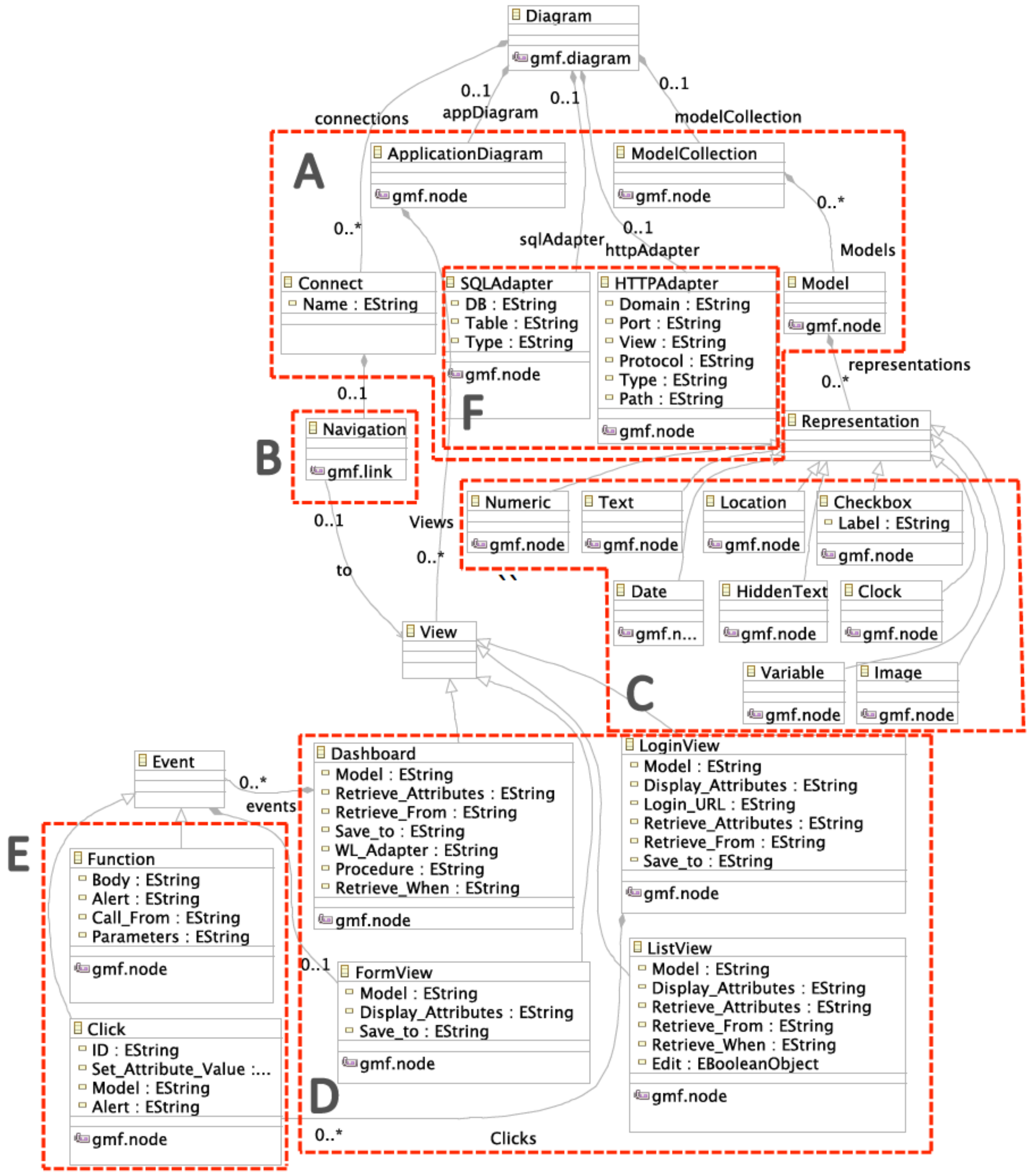


Figure 3.7: The ECore Metamodel Representation

includes high level components such as the *ApplicationDiagram* for the views and the *ModelCollection* for the models. The *Connections* cluster includes the navigation links to connect the views by representing this way the screen navigations. The *UI Components-Variables* cluster has all the widgets related to the user interface that allow users to insert data with different types (e.g., Numeric, Date, Text, Hidden Text etc.). Different types of views are available within the *Views* cluster. Views represent forms, lists, dashboards etc. The *Events* cluster's widgets are added within the Views and represent events or functions to be executed when an event is triggered. Finally, from the *Back-End* cluster users may select the type of Worklight adapters that suits their needs in order to generate a supporting back-end to the application.

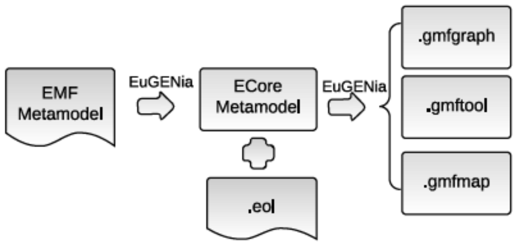


Figure 3.8: A graphical editor is generated from a Metamodel by using the EuGENia tool

Finally, we use the EuGENia tool to generate the GMF editor based on the metamodel specified. In fact, there are four plugins generated (Figure 3.8):
(a) Diagram editor plugin: The application specification plugin consisting of the editor and the palette. The GMF framework provides only a group of widgets namely Objects. Nevertheless, we are able to customize the way the widgets appear in the palette by using the Epsilon Object Language. Basically, we created a file with a *.eol* extension that specifies the different clusters of widgets. This file must be placed along with the Ecore metamodel file before the plugin is generated.
(b) Edit plugin: It includes all the providers necessary for the model elements to be displayed on the user interface. It also has a label property for all the models which is displayed along with the model when dropped on the editing area.
(c) Editor plugin: The editor plugin is a generated example editor to create and configure instances of a model.

(d) Test plugin: It provides templates and methods to parse the resource set created from a metamodel instance.

EuGENia is a tool that facilitates the generation of the Ecore metamodel and the above plugins. However, this process can also be done manually through the GMF Dashboard (Figure 3.9) by first selecting the Ecore metamodel and then generating the components as follows: *.genmodel* ⇒ *.gmfgraph* ⇒ *.gmftool* ⇒ *.gmfmap*. The generated diagram project is an Eclipse plugin project that can be exported and installed as such.

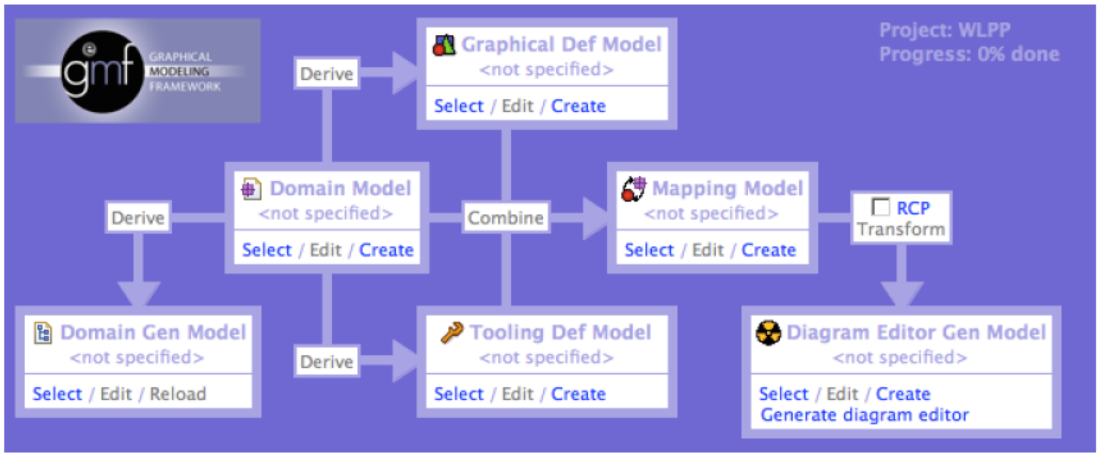


Figure 3.9: GMF Dashboard Overview [27]

3.3.2.1 Constraints

During the construction phase of the metamodel, the Emfatic language provides constraint-related commands. Defining constraints is important as it assists users while creating their own diagrams (components may be added/connected only to specific ones etc.). For this purpose, we were able to define several constraints that could lead to valid diagrams and make the building of diagrams an easier process by reminding developers the actions that are allowed and the ones that are not.

First, *Views* can only be added within the application's diagram objects and also a *Model* within the Model Collection object. By following this rule, views and models are separated and therefore a model can be re-used across multiple views.

Also, the *Navigation* Connection is able to only connect Views. The relation among views and models is indicated through the views' properties (where models can be declared). Therefore, in order to be consistent, there is only allowed to add a Navigation between two views, which represents the navigation from a screen to another.

A *Worklight Adapter* must be placed outside of the Application Diagram and the Model Collection as conceptually it does not belong to any of these objects. *UI and Variables* components can be only added within models. The metamodel specifies a list of such components that can be included in a model component. In the case that we desire to change the order they appear on the Form View, we could modify the aforementioned list by reordering the widgets.

Finally, *Events* components can be only attached to Views. Since the views represent the user interface through which users are able to interact with the application, events such as the click event and custom functions (that can perform calculations and/or modify models' attribute values) can be attached in a list form to the proper views.

3.4 The Application Specification Plugin

We conceptualized an easy-to-use graphical user interface with a drag-and-drop interaction. The application specification plugin is composed of three parts: (a) the editing area, (b) the property console and (c) the palette (Figure 3.10). The editing area allows the creation of application diagrams through the available widgets contained in the palette. As mentioned before, the predefined widgets are separated in six groups (Objects, Connections, UI Components-Variables, Views, Events, Back-End) so that developers can easily find and select the widget they need. Widgets can be dragged and dropped in the editing area. However, due to the restrictions specified while generating the metamodel, some widgets must be placed within other components. Each one of the components may have multiple properties which are displayed in the property console after a widget is placed in the editing area and is selected. Below we explain the components contained in each cluster along with their properties.

➢ **Objects**

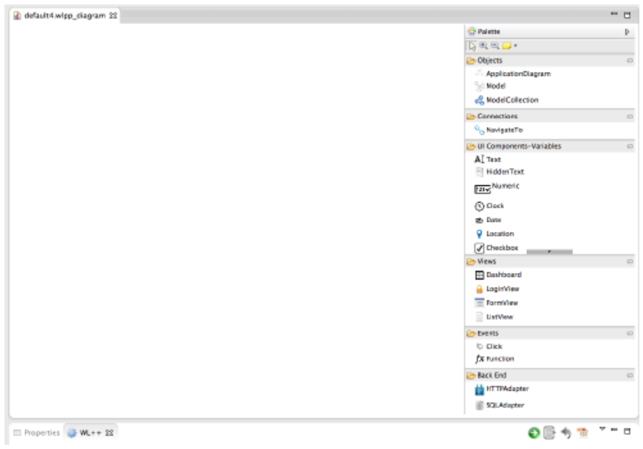


Figure 3.10: The *WL++* Application Specification Plugin

The **Application Diagram** is the main component that consists of the views of the to-be-generated application. Therefore, there is a restriction rule which permits only View components to be added in the application diagram. The Application Diagram has one property namely "name" that indicates the application's title and is displayed on top of the first screen of the application.

Each **Model** corresponds to a Backbone model component. A model has a "name" property that is essential when it comes to declaring the models to the proper Views. Usually an application consists of several models. All these models are declared within the **Model Collection** object.

➢ **Connections**

The connections among views are declared by "Navigation" arrows. These connections can be added only between views and essentially represent the Backbone's routing system. A **Navigation** type of connection has three properties: *name, from* and *to*, where *from* and *to* are the connected views.

➢ **Events**

Every application offers some events to allow end-users to interact with it and also event handling methods to process the input information. We have defined the **Click** event as a widget that may be added into the Views and it can either perform actions within an existing button (e.g., Save button included by default in a Form View) or new buttons that are created when events are added into the views. The properties of a **Click** event are: *Name*: the label of the button, *Model*: the model associated with the event (e.g., a model's attribute may change while this event takes place), *ID*: the button's ID (if there is no such button ID, then it is created automatically), *Alert*: the alert property is useful and can be used in case where we have a special message to display when a button is clicked (for instance, a message informing the user that the entry has been saved successfully).

Most of the times, applications have functions that may compute and return a value based on user's input. Therefore, in such cases, we need a **Function** widget representing a custom function. In our case, the**Function** widget has five properties: *Name*: it indicates the function's name, *Model*: the models associated with it (this essentially means that attributes of this model can change values), *Set Attribute Value*: the command to be executed while the attribute of the model is specified to change, *Alert* which is similar to the *Alert* attribute of the **Click** event above.

➢ **UI Components-Variables**

The **Text** widget can be added as an attribute to a model. It has a *label* property, which is defined by the user during the creation of the UML diagram. This widget appears as a *text field* where the mobile-application user may enter their textual input. The **Hidden Text** widget is displayed as an input box on the mobile application user interface; however, the typed symbols are hidden and appear as stars. This widget may be useful in cases of user registration/login forms. Similarly to the *Text* and *Hidden Text* widgets, the **Numeric** widget is also displayed as a label along with a *numeric input field*, accepting numerical values as input. This type of widget would be appropriate in cases of telephone numbers, product prices, age etc.

The **Clock** widget represents a date-time picker in the application user interface, initialized with the device's time. The **Date** widget results in a date-time picker in the application user interface, implemented using the Mobiscroll JavaScript library. The **Location** widget can be used to indicate that an attribute is a location represented by a pair of latitude-longitude coordinates. The widget accesses the device's GPS through the HTML5 Geolocation API and stores these values in the local mobile application's database. A **Checkbox** is represented by a *label* along with a *checkbox*. The **Image** widget appears as an *image box* along with a button to upload images by selecting pictures from the mobile device's picture library. It could be easily expanded into a widget accessing the device's camera in order to capture picture/video instances. Sometimes, there is a need for an internal variable that does not have a visual representation. Such kind of variables are declared by using the **Variable** widget.

➢ **Views**

A typical mobile application has multiple views and usually a Dashboard that assists the navigation among the rest of the views. Based on the requirements and the experience we gained from two applications belonging to the field of healthcare developed from scratch we concluded that four types of View objects are sufficient to build a fully functional catalogue-based application. The available views are listed and explained analytically below.

As mentioned before, Dashboards include other Views and therefore they provide a way to navigate from the main screen to Forms and Lists. By default the **Dashboard** is composed of several icons/buttons that represent the Views. Icons specific to each view are provided with the framework; however, developers are able to modify them by replacing the existing icons with customized ones. The Dashboard widget has few properties: *Model*: the model associated with this view, *WL Adapter*: the adapter in case where an interaction with a back-end should take place, *Procedure*: the adapter's procedure to be called, *Retrieve From*: the REST URL, *Retrieve When*: a condition that serves as a filtering function when data is retrieved, *Retrieve Attributes*: the attributes to be retrieved, *Save To*: the collection where the data must be stored (locally).

The **Login View** serves the purpose of user verification when there is a back-end involved and users must log in in order to have access to the data or enter new data which needs to be stored in an external database. Figure 3.11 shows a Login view composed of a **Text** widget for the username input and a **Hidden Text** widget for the password.

The **Form View** is responsible for displaying the model attributes representations to the end-users. Through this view users are able to enter their entries and save the data locally or remotely. A Form View has the following properties: *Model*: the model associated with the View, *Display Attributes*: the corresponding model

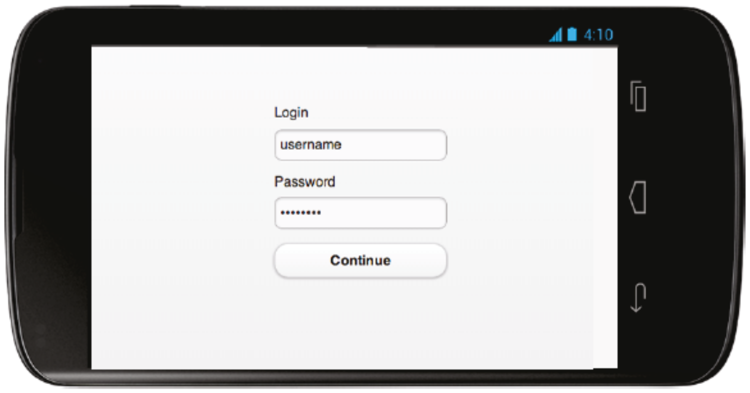


Figure 3.11: Login View

may have multiple attributes; however, we may need to display only a subset. This is possible through the *Display Attributes* property where a developer may declare the attributes to be displayed in the Form View. The **List View** is an optional widget, which serves as a view where all the existing entries are displayed. By clicking a list item, it gives the users the ability to delete an entry or edit it and save it again. The List View properties include: *Model*, *Retrieve From*: The REST URL or the collection to retrieve the entries to be displayed in the list, *Retrieve When*: a condition, *Retrieve Attributes*: attributes to be retrieved from an external or local source, *Display Attributes*: the list of attributes to be displayed, *Edit*: a boolean property indicated whether a list is editable or not.

➢ **Worklight Adapters**

The IBM Worklight Framework offers several ways to achieve an interaction with different kind of back-ends. We have decided to support two of them (HTTP, SQL) as they are the most commonly used. Through the **HTTP Adapter**, mobile applications can interact with a back-end via a RESTful web service that is also generated by the *WL++* application specification plugin. The **SQL Adapter** widget is used when a direct connection between the client application and an existing database is required. In this case, there is a local database and therefore the adapter is responsible for the execution of the SQL queries.

3.5 The Process of Code Generation

A developer may develop an application either from scratch or based on an existing back-end. The former requires the analysis of the application to be generated into models and views whereas the latter implies that the models along with their attributes are implicit in the API. Therefore, there are two ways of generating code: (a) from scratch, and (b) based on an existing RESTful back-ends or Schemas. Although both cases require different steps, the process is similar from the point where the application's models are defined. Below we briefly explain the process of generating code in both cases by referring to the *WL++* components. A more detailed explanation is given through examples in Sections 3.9 and 4.

3.5.1 Defining the Application's Data

In the case where an application is developed from scratch, there should be at least a *Model Collection* and an *Application Diagram* component populated with models and views respectively. In order to define the models, the developer has to create a new instance of the metamodel by populating a *Model Collection* widget with the necessary models. The model definition depends on the developer's perception of the application. This means that a developer may specify different models from another developer with both set of models to generate similar applications in terms of functionality. Next, *UI-component* widgets are added within each model to represent the model's attributes. Since each representation is associated with a specific type, developers do not need to define the attributes' types. Essentially, the models and their attributes represent the application's data as we mentioned before in the Model-View-Controller pattern.

So far the *Model Collection component* is created from scratch. However, in the case where there is already an existing design that makes explicit the models of the application, then the *Model Collection* could be generated based on that design. *WL++* supports two different code generation components: (a) based on RESTful back-ends that return JSON responses when invoked, and (b) XSD Schemas. In the first case, the developer gives as input an URI that is automatically invoked by

WL++. Alternatively, developers can also give as input a JSON response. Then, a JSON parser parses the response in order to display the model's attributes extracted to the developer. This process is done when clicking the "Extract Attributes from JSON" button (Figure 3.12). At this point developers can add, delete, or modify the attributes extracted and then generate a representation of the models (XMI file) that is compatible with the *WL++* code generation engine. This step is done by clicking the "Generate XMI" button (Figure 3.12). Based on the XMI representation, a new diagram that depicts the *Model Collection* components with the models and attributes is generated.

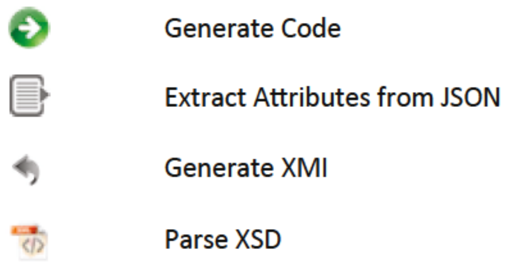


Figure 3.12: The set of buttons of the *WL++* plugin

3.5.2 Defining the Views and Navigations

When the models are defined, the next step is to create the user interface in terms of views and the router in terms of navigations among the views. Each view may represent a dashboard, form or list of items and therefore it is associated with specific models. Although the construction of models is not necessary to be done before the construction of views, it is more convenient as views must be associated with the proper models. When the views are added in the application diagram, the developer specifies the routing among them by adding navigations from one view to another. After the completion of the diagram, the "Generate Code" button is clicked to generate the application (and the back-end) (Figure 3.12).

3.6 Code Generation Engine

Along with the generation of the .diagram plugin, there are generated several Java interfaces and implementation classes representing the components of the Ecore metamodel. For instance, a node is represented as a Java class while the elements that this node may contain as a list of attributes (along with *getters* and *setters* methods).

In Figure 3.13 we can see how the Java implementation of the components defined in the Ecore metamodel is related to the Backbone components. First, the *Model* class has a "name" attribute which corresponds to the Backbone model's name, a list of representations which are the Backbone model's attributes. Each attribute has a default value and is declared within the *store* object in order to be stored in the device's database. Therefore, the attributes "name" and "list of attributes" are information to be injected in the template specified by the "template" attribute. (Figure 3.13a, 3.13b).

A *View* entity is associated with a specific (collection of) model(s) and is responsible for the HTML-based templates which this View expects to be appended in the DOM element (Figure 3.13c). The output View file follows the Backbone view's structure and it includes the templates' initialization and the handling of the events that may occur while the end-user interacts with the application. For example, an event may be a click of a button or a change of an attribute value; if such events occur then the view handles them through proper functions. The "render" function renders the template by adding specific data extracted from the associated with the view (collection of) model(s) (Figure 3.13d).

Finally, the *Navigation* entity includes a list of routes and the corresponding functions. The routes are extracted from the connections among the different views (stored as a <*from, to*> combination) while the functions are responsible for initializing the new view by passing the proper (collection of) model(s) (Figure 3.13e). This information is declared within the property console. The *JSRouter* also contains a "changePage" function that takes as parameter the page to be rendered. This function first destroys the old view(s) and then it replaces them with the new one.

```
Model {
  name,
  <list of attributes>,
}
```

(a) Model

```
Models.name=Backbone.RelationalModel.extend({
  relations[],
  store:{<list of attributes>},
  defaults{<list of attributes>}
});
```

(b) JSModel

```
View {
  name,
  model(s),
  event(s),
  <list of templates>
}
```

(c) View

```
Views.name=Backbone.View.extend({
  initialize:function(){...},
  events:{...},
  render:function(){...}
});
```

(d) JSView

```
Navigation  {
  <list of routes>
  <list of routing functions>
}
```

(e) Navigation

```
Router.Routing=Backbone.Router.extend({
  initialize:function(){...},
  routes: {
      <list of routes>:<list of routing functions>
  },
  changePage:function(page){...}
});
```

(f) JSRouter

Figure 3.13: The Java classes (3.13a, 3.13c, 3.13e) and the corresponding JavaScript Backbone entities (3.13b, 3.13d, 3.13f)

The above implementations of the Java classes, represent only the skeleton of the classes while they miss significant functionality. Therefore, additional methods were added in order to be able to add the proper functionality and generate the desired output. These methods are mainly responsible for the calculation of new values and template rendering (by passing data to the proper templates).

An instance of the Ecore metamodel is created when a new *WL++* project is initialized through the Eclipse IDE. Then, two files are generated: (a) a *.diagram* file representing the diagram editor (editing area and palette) and (b) a *.wlpp* file that represents the information of the diagram in a textual format and is updated when the developer hits the *Save* button. Basically, GMF produces a representation set of the components depicted in the diagram in a separate file that includes all the information regarding the models, attributes, their representations, connections and labels. Figure 3.14 shows an example of a *.wlpp* file that is produced from a diagram containing a *Form View* widget within an *Application Diagram* widget.

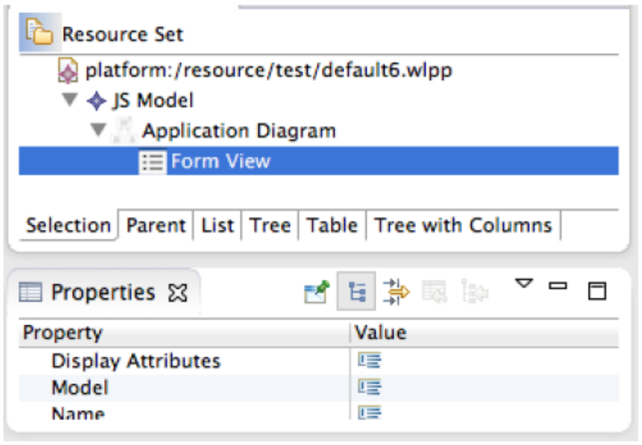


Figure 3.14: The resource set displayed in a tree view representing an example graphical diagram composed of a *Form View* view

The plugin generated by EuGENia provides us with classes that are able to parse the application's diagram representation file. Therefore, while traversing the resource set, for each component (e.g., model, view, navigation, adapter) an instance of the corresponding to that component Java class is initialized. In fact, components are associated with the proper predefined templates. As we can see in Figure 3.15, the resource set (XMI representation) of the application diagram is the input that is parsed by the *WL++* code generation engine in order to identify the modes, views and properties and initialize the proper Java classes (we will get back to this point in the future). Since there are templates defined within the Java implementation of the resources, the proper data is passed into these templates that are then rendered by producing the desired output (mobile application files, back-end related files).

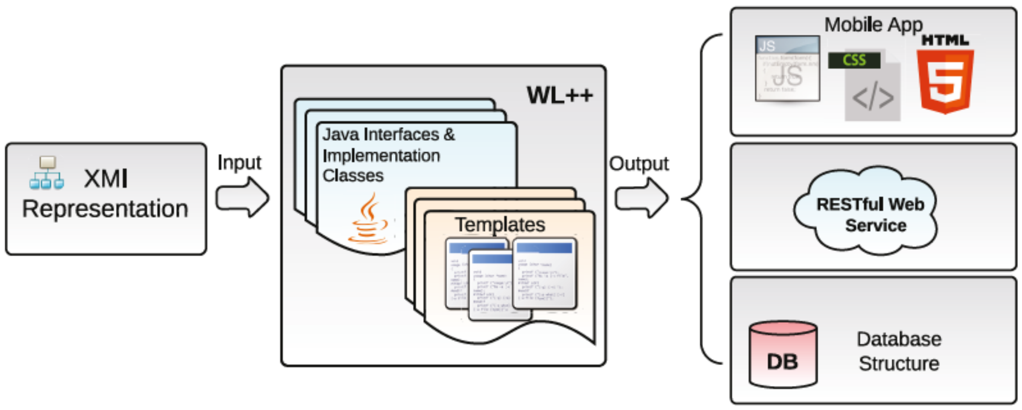


Figure 3.15: Model to Code Transformation with *WL++*

3.6.1 The StringTemplate Templating Engine

The *WL++* plugin integrates the *StringTemplate*[3] Java Templating Engine based on which the output files are generated. *StringTemplate* is a lightweight and versatile library as it allows the use of self-defined delimiters while defining the variables. These templates are rendered by the Java classes generated based on our Ecore metamodel and populated by several methods where additional functionality and calculations have been manually added. Generally, these templates include the structure of the files to be generated and multiple parameters that represent customized functions to be injected based on the diagram built. In fact, by using the *StringTemplate* engine, it is possible to inject data within the templates by either defining a "get" attribute that return the value of a specific attribute or a "get" method returning a calculated value or a list of values within the Java class. Then the attribute's (method's) name is declared as an attribute within the template and it gets the results returned by the Java class when the template is rendered. For example, a form view may have several events (click, blur, change etc.) These events

[3]http://www.stringtemplate.org/

```
events: {
    obj.events; separator=",\n"
  }
```

(a) The events section of the Form View String Template

```
public Vector<String> getEvents() {

      ...

  return events;
}
```

(b) Compiling and adding the template into the DOM while passing the data object as a parameter

```
events: {
    "click #Save-btn"        :   "save",
    "change [id^='Name']"    :   "attributeFromElementId",
    "blur [id^='Name']"      :   "attributeFromElementId",
    "change [id^='Age']"     :   "numberFromElementId",
    "blur [id^='Age']"       :   "numberFromElementId"
  }
```

(c) Compiling and adding the template into the DOM while passing the data object as a parameter

Figure 3.16: An example of a template being rendered by the StringTemplate engine

according to the Backbone documentation must be defined within the view they belong to. Therefore, in the template we define the skeleton of the events section that follows the Backbone structure (Figure 3.16). Typically, for each event there is a variable which is mapped to the "get" method defined in the Java class. When the template is rendered then the "getEvents" method is executed. The return statement returns the events to the template which then replaces the attribute "events" with the list of events separated by a new line (Figure 3.16c) as indicated by the *separator*=",\n" command.

WL++ includes twelve main templates associated with each one of the following components:

- *Views:* Dashboard, Login View, Form View, List View

  The views correspond to the pages that a user may navigate to. These views are constructed based on the predefined templates and the application diagram. A template consists of the skeleton of a particular view which then is populated with user interface components represented by the widgets specified in the application specification diagram.

- *Core:* Client Model, UI Representation

  A model is created based on the model's attributes added during the construction process of the application's diagram. Developers are responsible for defining their own models and specifying their attributes. As these attributes correspond to UI representations, we have defined a file that includes all the UI representations of each attribute. For instance, a text widget is represented by a text input form while a checkbox widget by a label and a checkbox component.

- *Router:* Applications' Router

  The router plays the role of the controller; it associates URLs to views in order to allow end-users to navigate from one screen to another. This template specifies the router's syntax according to Backbone where the routes are added based on the navigation arrows among views created through the *WL++* graphical editor.

- *WL Adapter:* adapter-impl, adapter-xml These files are constructed by following the skeleton of a typical Worklight adapter. Besides the default procedures performing CRUD operations, specific parameters are replaced with the proper data retrieved from the application's diagram. This data consists of the REST service's URI, port, parameters passed to the RESTful service etc. Alternatively, in the case of the SQL adapter the username, password, database name and port must be declared.

- *REST:* Server Model, REST, Database

  The Server Model reflects the client model (Backbone model) and is implemented in Java. Therefore, a template representing a Java class is filled with attributes retrieved from the application's model. Since each attribute is associated with a Java type (e.g., text widget corresponds to a String attribute, a GPS location to a set of two double attributes etc.) we define these attributes and generate *getters* and *setters* methods for each one of them. The Database.Java file focuses on the communication of the service with the external Database and therefore, there reside the CRUD operations implemented. Finally, the REST URI paths are defined in the REST.Java file.

There are also defined few sub-templates related to specific modules such as the local-storage module. These sub-templates are injected by the code-construction engine into the main templates when needed. For instance, the local-storage template is used for data that needs to be stored into a web-sql database on the device.

3.6.2 External Libraries

Besides the Phonegap library which is built within Worklight, the main JavaScript library used is Backbone.js. However, we have used few extra external libraries that are explained in detail below.

- *MobiScroll*

  MobiScroll[4] is a lightweight library (approximately 20kB) that provides developers with native looking user interface widgets regarding date and time data. End-users can select a date by scrolling or increasing/decreasing date values through plus/minus buttons. It also includes a theming feature through which a developer can specify whether a datetime picker widget should look like a native Android component or an iOS one for example. This can be easily defined through the parameter "theme" as *"theme: android"* or *"theme: ios"* respectively. Although MobiScroll started as a free datetime picker widget, now it has expanded its functionality and offers many more widgets such as Calendar, Range, Color, Rating, Temperature etc. However, users need to purchase the new version of this library.

- *FastClick* FastClick[5] is an open source project aiming to eliminate the delay of the event-to-be-fired after a user has touched the device's screen. This

[4] http://mobiscroll.com/
[5] https://github.com/ftlabs/fastclick

delay is relatively small (approximately 300ms) and it happens due to the fact that the device's browser waits to detect if there will be a second touch (in case of where a user may intend to perform a double tap). However, when developing cross-platform applications, the responsiveness is important and therefore, such libraries are extremely helpful in making web-based apps behave as native ones.

- *JQuery Mobile*

  There is a wide variety of libraries/frameworks (such as Sencha Touch, JQuery Mobile, PhoneJS[6], Kendo UI[7]) that are able to handle events, allow navigation among different pages, make requests to external web services and databases through AJAX call etc. Those frameworks are designed to create responsive web-based applications by leveraging HTML5 and CSS3 features. PhoneJS and Kendo UI are built based on JQuery whereas Sencha Touch has a different philosophy and fundamental differences from the one of JQuery [28]. Although both frameworks at this point are mature with a rich documentation including many examples and demos, Sencha Touch appears to be a more proper solution for Graphics-Driven applications whereas JQuery is mainly used for Form-Driven applications. Moreover, JQuery Mobile syntax is much easier and closer to the HTML syntax without being complicated by JavaScript code as it happens in the case of Sencha Touch. Although Sencha Touch is developed especially for mobile apps and seems to be faster compared to JQuery Mobile, JQuery Mobile covers a large area of browsers and mobile devices [28]. Previous experience with JQuery Mobile and the nature of application we intend to develop played an important role in its adoption and integration in the *WL++* framework.

- *Backbone WebSQL*

[6]http://propertycross.com/phonejs/
[7]http://www.telerik.com/kendo-ui

The Backbone WebSQL[8] library provides an implementation of the *Backbone.sync* option in order to sync data from/to the application and the WebSQL. By using the Web SQL Database API developers can create databases that can be queried using various SQL statements.

3.7 The generated Back-End

Along with the mobile application that is deployed on a mobile phone and runs locally, the *WL++* plugin generates a RESTful API through which the mobile application can send/receive data to/from external databases. This communication is established via a set of built-in adapters within the Worklight framework (Section 3.1). More specifically, through the *HTTP* adapter a mobile application is able to connect with a RESTful web service. The REST APIs supported by our plugin implement the CRUD and filtering operations relying on the attributes of the application models. Filtering is often useful when we have to deal with long lists of entries. For example, a shopping catalog that contains a large number of different products would not be possible to be reviewed in order to find a specific product quickly.

In Figure 3.17 we can see the output layers of the *WL++* application specification plugin. The generation of the RESTful web service depends on the presence of an HTTP adapter. When such an adapter is detected by the code generation engine, then a RESTful web service is generated. The web service's files are compatible with a web service implemented using the Jersey[9] framework.

To that end, we developed a generic adapter through the Worklight interface which is customized based on the mobile application to be generated. The HTTP adapter is deployed by the Worklight server (See top layer of Figure 3.17). The communication between the application and the back-end would be also possible by using AJAX requests. Although, the Worklight adapters also use AJAX, they offer advantages such as security and transparency of the retrieved data among others.

A common characteristic of the latest mobile applications is that of "push no-

[8]https://github.com/MarrLiss/backbone-websql
[9]https://jersey.Java.net/

Figure 3.17: The different layers of the auto-generated components

tifications". As users interact with the application by storing/receiving data, the application often responds through notifications, typically containing information extracted based on the user's interaction history or that of his/her peers. For this purpose, *WL++* includes a monitoring service which lives on the server side and keeps track of all the API calls that have been invoked. It also records the data exchanged between the mobile application and the back-end server in a JSON format. Therefore, this data can be further analyzed and used in order to make recommendations by sending notifications from the server to the mobile device.

3.8 Application Code Structure

The output of the *WL++* code generation engine is a "gen" folder, that includes the application files and the corresponding back-end files, assuming that the application design indicates the need for a back-end. The structure follows a typical Worklight project (Figure 3.18). This practice allows developers to quickly set up a Worklight project and transfer the files from the generated folder to the corresponding

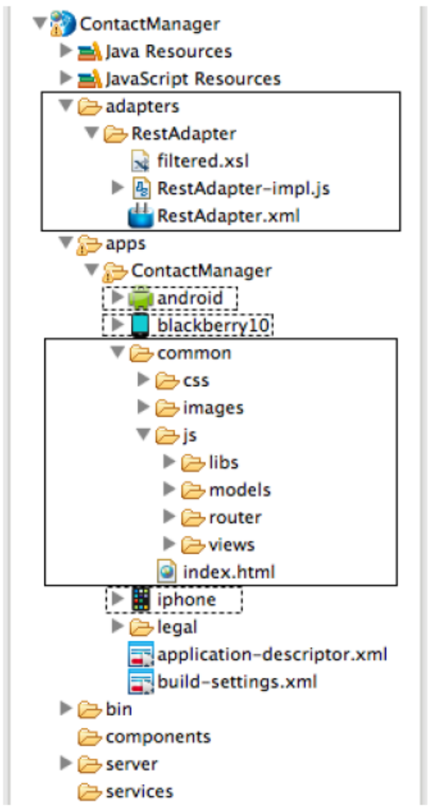


Figure 3.18: The IBM Worklight Project Structure. Three environments have been added (Android, Blackberry 10 and iPhone)

Worklight project, to be deployed.

A Worklight application consists of three main file categories representing (a) references required for the application development and deployment, (b) the application and its adapters, and (c) server-customization components. When a new project is created, then a typical "Hello World" application is automatically generated by the framework. This project is composed of several JavaScript, HTML, and CSS files under the "common" folder.

Therefore, the *WL++*- produced files are all stored under the "common" folder. Basically, this folder contains all the files shared among the targeted environments (Figure 3.18). A user may add new environments such as Android or iOS when needed in order to be able to extend or override the common files so the application may look/behave differently while deployed in different platforms. Furthermore, a Worklight project has files responsible for the framework's initialization (e.g., the *initOptions.js* file ensures that the Worklight JavaScript framework is initialized). These files, are generated automatically and imported into the HTML files by *WL++*, as it happens while creating a new Worklight project.

While Backbone does not require any special structure regarding the files, the experience we gained from the parallel developing of prototype applications led us to the decision to categorize the generated files into four separate folders: (1) *libs*, (2) *models*, (3) *views* and (4) *router* (Figure 3.18). This practice assists the process of navigation and modification of the proper files when needed.

The *libs* folder contains all the essential libraries such as Backbone related libraries, JQuery and JQueryMobile as well as other libraries related to special widgets (e.g., Mobiscroll.js and underscore.js). The *models* folder typically contains the JavaScript files that describe all the models of the application. When there is a collection of models, it is also placed in the same file as the corresponding model from which it is composed of. The *views* folder is responsible for the user-interface rendering. Usually, a view is associated with a (collection of) model(s) and renders the relevant information into specific template components. Finally, the *router* folder contains the URL paths and the functions for creating the proper views according to the URLs. When users navigate from one view to another then the router creates the proper views and renders them according to the template associated with each view (and/or sub-view).

A Worklight project may have one or multiple adapters in order to communicate with existing back-ends and all of them reside in the *adapters* folder. Apparently, an application that does not require any connection with a back-end, does not need any adapters to be implemented. The need for a back-end is indicated by the developer when a *Worklight Adapter* widget is added in the application diagram, created using the *WL++* plugin. In this case, two more folders are generated: (a) the *adapters* folder and (b) the *restAPI* folder. For each adapter, two files are generated namely, *adapter.xml* and *adapter-impl.js*. This structure is specified by Worklight; more specifically, when a new adapter is added, a new folder is created under the general *adapters* folder. The *adapter.xml* file has important information related to the back-end, such as the URL and the port number of the web service, and also, the names of the procedures (functions) which are implemented in the *adapter-*

impl.js file. Essentially, the *adapter.xml* file plays the role of the controller which invokes the proper procedures an event is triggered on the application side. Therefore, Worklight decides (a) the organization of the front-end files (HTML, CSS and JavaScript) and (b) the high-level structure of the REST APIs that will link to remote databases through a RESTful API, whereas Backbone decides in some more detail the structure of the files.

Besides the adapter files, *WL++* generates the *restAPI* folder which has multiple Java files in order to be copied in a Web Service project. The code is responsible for the communication of the application (through the adapter) with a remote database through a set of RESTful APIs. The APIs correspond to basic CRUD and filtering operations (search by specific attributes). More specifically, the first time the mobile application is launched, a remote database is created with two tables, (a) for data storage and (b) for API monitoring purposes. When new data is entered, then the corresponding procedure is invoked in order to send a POST request to the server with the data entered; therefore, the Web Service stores this information in the database while the monitoring service keeps track of the API calls and the transmitted data.

3.9 Reverse Engineering with *WL++*

So far we have described the generation of a mobile application with a supported back-end from a modeling diagram assuming that there does not exist any back-end and therefore it must be generated by the *WL++* plugin. However, there are existing back-ends that need mobile clients in order to access the data through pre-defined APIs. The data models are implicit in the API and therefore, we developed a process for semi-automatically extracting data models from these APIs instead of defining them from scratch as explained before. Below we explain the process of constructing models and generate cross-platform mobile clients through RESTful back-ends that support JSON responses (Section 3.9.1) and through XML Schema Definitions (XSDs) (Section 3.9.2).

3.9.1 Constructing Models from JSON Responses of RESTful Web Services

The application specification plugin is able to take as input the existing back-end in terms of the APIs it uses and construct the main models based on the retrieved response after an API call has been invoked.

The reverse engineering component requires as input either RESTful URLs specified by the developer (including instance values for the query parameters) or the JSON response returned after the URL invocation. For example, we could give as input the following URL that retrieves the "Soul Surfer" movie details (e.g., "release date", "rating", "actors") from the Rotten Tomatoes Movie Database[10]. The URI requires an API key which means that the user must obtain one by registering first, the query (in this case the title of the movie for which the details are to be found) and the total number of pages to be returned.

"http://api.rottentomatoes.com/api/public/v1.0/movies.json?**apikey**=apikey&**q**=Soul %20Surfer&**page_limit**=1"

The first step is to analyze the URL by using the Steiner's URL analysis method and extract the query parameters [34]. According to the Steiner's REST URL analysis tool, an URL is composed of the base, path, method name (e.g., GET, POST) and the parameters. In our case, the URL is composed of three parameters: *apikey*, *q* and *page_limit*. These parameters along with their values are stored in order to be displayed to the developer later on along with the parameters extracted from the JSON response. Therefore, *WL++* consumes the above URL and sends an HTTP request. The JSON response (Figure 3.19) is parsed while the parameters and their instance values are stored and displayed in the *WL++* console (along with the attributes extracted from the URL itself) (see Figure 3.20). The *WL++* console represents a table with four columns. The column *Attributes* includes the parameters extracted from the URL and the JSON response. The column "Type" represents the type of each attribute (e.g., Text, Numeric, Date) whereas the column "Instance

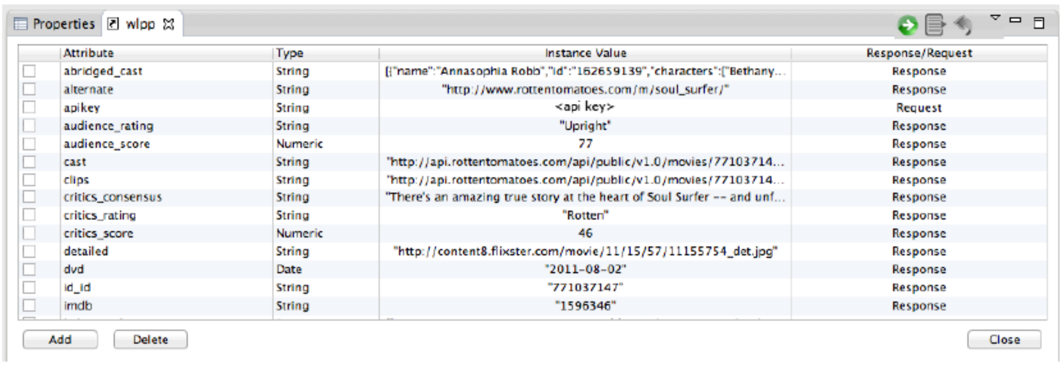
[10] http://www.rottentomatoes.com/

value" is the value of the parameter for which we need to determine the type of the parameter. Finally, the column "Response/Request" indicates the source of each attribute (Response if the attribute was extracted from the JSON response and Request if the attribute was extracted from the URL).

```
{
    "total": 2,
    "movies": [{
        "id": ''771037147",
        "title": "Soul Surfer",
        "year": 2011,
        "mpaa_rating": "PG",
        "runtime": 106,
        "critics_consensus": "",
        "release_dates": {
            "theater": "2011-04-08",
            "dvd": "2011-08-02"
        },
        "ratings": {
            "critics_rating": "Rotten",
            "critics_score": 46,
            "audience_rating": "Upright",
            "audience_score": 76
        }
```

Figure 3.19: A subset of the API invocation response

The "Type" column is automatically populated with the type of each attribute. The JSON response has no information regarding the type of attributes. Therefore, we attempted to determine their types from their instance values. For example, by checking if a String can be converted into an Integer or a Date type, we can annotate this attribute as of type Numeric or Date respectively (Algorithm 1). Generally, if an instance value is type of Integer, Double or Float then the attribute is annotated as Numeric. For the Date type, we check if an instance value has the same format as several predefined Date formats (Figure 3.21). As we can see in Figure 3.19, although the attribute "dvd" has a value of "2011-08-02" which is return by the service as a String, it is automatically annotated with the type Date in the *WL++* console (Figure 3.20) as it matches the date format defined in line 7 of



	Attribute	Type	Instance Value	Response/Request
☐	abridged_cast	String	[{"name":"Annasophia Robb","id":"162659139","characters":["Bethany...	Response
☐	alternate	String	"http://www.rottentomatoes.com/m/soul_surfer/"	Response
☐	apikey	String	<api key>	Request
☐	audience_rating	String	"Upright"	Response
☐	audience_score	Numeric	77	Response
☐	cast	String	"http://api.rottentomatoes.com/api/public/v1.0/movies/77103714...	Response
☐	clips	String	"http://api.rottentomatoes.com/api/public/v1.0/movies/77103714...	Response
☐	critics_consensus	String	"There's an amazing true story at the heart of Soul Surfer -- and unf...	Response
☐	critics_rating	String	"Rotten"	Response
☐	critics_score	Numeric	46	Response
☐	detailed	String	"http://content8.flixster.com/movie/11/15/57/11155754_det.jpg"	Response
☐	dvd	Date	"2011-08-02"	Response
☐	id_id	String	"771037147"	Response
☐	imdb	String	"1596346"	Response

Figure 3.20: The *WL++* console populated with attributes retrieved from an API invocation

Figure 3.21.

Algorithm 1 Determining an attribute's type based on its instance value

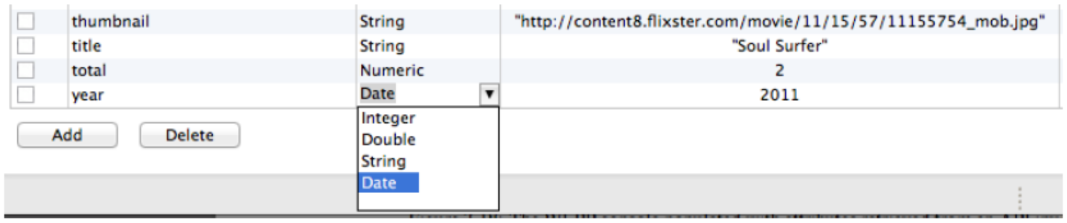
```
1: procedure GETATTRIBUTETYPEFROMINSTANCEVALUE(attribute)
2:     if (attribute.getInstanceValue() is (Integer or Double or Float)) then
3:         return Numeric;
4:     if (attribute.getInstanceValue() is Boolean) then
5:         return Checkbox;
6:     if (attribute.getInstanceValue() is Date) then
7:         return Timestamp;
8:     if (attribute.getInstanceValue() is String) then
9:         return Text;
```

However, there are cases where the type of an attribute is not determined correctly. For example, the attribute "year" has an instance value of "2011". This value is annotated as Numeric (as it can successfully be converted into a number and it does not match any of the date formats) although it represents a Date type of attribute. For these cases, the plugin allows developers to modify the type of the attributes by simply clicking on the type cell next to the specific attribute and selecting the proper type from a drop-down list of types (Figure 3.22). Furthermore, through the **Add** and **Delete** buttons (at the bottom of the *WL++* console), developers can add new attributes, and delete existing ones.

When the process of defining the attributes is completed, by clicking the "Generate XMI" button, an XMI representation of the model composed of the attributes shown in the console is generated (Figure 3.23). The XMI representation follows

```
"M/dd/yyyy"
"dd.M.yyyy"
"M/dd/yyyy hh:mm:ss a"
"dd.M.yyyy hh:mm:ss a"
"dd.MMM.yyyy"
"dd-MMM-yyyy"
"yyyy-mm-dd"
"yyyy-MMM-dd"
```

Figure 3.21: The date formats defined to determine if an instance value is type of Date



☐	thumbnail	String	"http://content8.flixster.com/movie/11/15/57/11155754_mob.jpg"
☐	title	String	"Soul Surfer"
☐	total	Numeric	2
☐	year	Date	2011

Figure 3.22: Manually defining the type of an attribute

the structure of the XMI generated by the GMF framework. Essentially, this is the *.wlpp* file that was generated during the forward code generation process (Section 3.6). Therefore, we used a template to populate with models, their attributes and types as required by the GMF framework in order to initialize an instance modeling diagram and then to generate the application files using the *WL++* application specification plugin.

Consequently, the next step is to convert the XMI representation of the model(s) into a graphical representation where developers can add views and navigations, and associate the views with the proper models. The model along with the attributes that were generated automatically from the invocation of the API is depicted in Figure 3.24. Because each attribute was annotated with a type by *WL++*, the model appears to have been created by selecting the proper widgets from the palette (the types of the attributes correspond to the types of the widgets defined within the palette). For example, for the URL parameter "title" and the response parameter "synopsis" the *Text* widget was used. The "runtime" attribute represents the length of the movie in minutes and therefore its type was identified as a *Numeric* widget. Both the "critics_score" and the "audience_score" are parameters that represent the

```xml
<?xml version="1.0" encoding="UTF-8"?><WLPP:JSModel
   xmi:version="2.0" xmlns:xmi="http://www.omg.org/
   XMI" xmlns:xsi="http://www.w3.org/2001/
   XMLSchema-instance" xmlns:WLPP="http://
   ualberta.ca">
<modelCollection>
  <Models Name="movies.json">
    <representations xsi:type="WLPP:Text" Name="
       title"/>
    <representations xsi:type="WLPP:Numeric" Name="
       runtime"/>
    <representations xsi:type="WLPP:Date" Name="
       theater"/>
    <representations xsi:type="WLPP:Numeric" Name="
       critics_score"/>
    <representations xsi:type="WLPP:Numeric" Name="
       audience_score"/>
    <representations xsi:type="WLPP:Text" Name="
       synopsis"/>
  </Models>
</modelCollection>
</WLPP:JSModel>
```

Figure 3.23: The XMI generated based on the JSON response

score of the movie given in a number that ranges from 0 to 100. They both are specified as *Numeric* widgets whereas the “theatre” attribute is depicted as a *Date* widget since it represents the date when the movie was released.

Finally, when new views are added to the application diagram, a mobile application can be generated by following the same process as in the forward code generation (Section 3.5.1). To illustrate this process, the developer may add three views: a *Form View* and a *List View* to display the movie-related data, and a *Dashboard* for navigation purposes as shown in Figure 3.25. Both the *Form View* and the *List View* are associated with the ‘Movie” model through the views’ properties as shown in Figure 3.26. The code generated results in the MovieDatabase application, two screenshots of which are depicted in Figure 3.27.

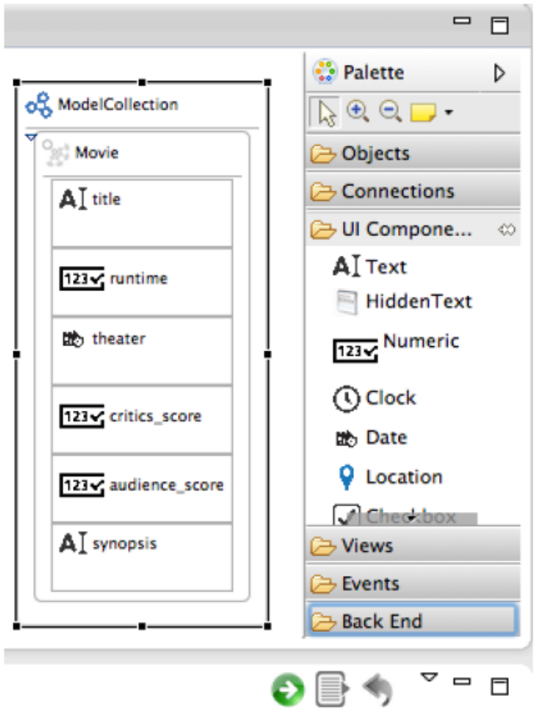


Figure 3.24: The model generated from the API invocation

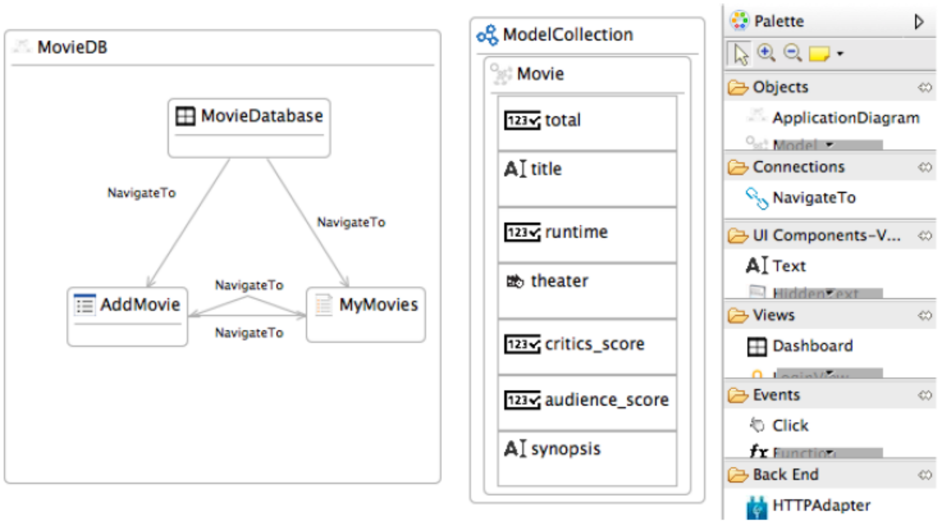
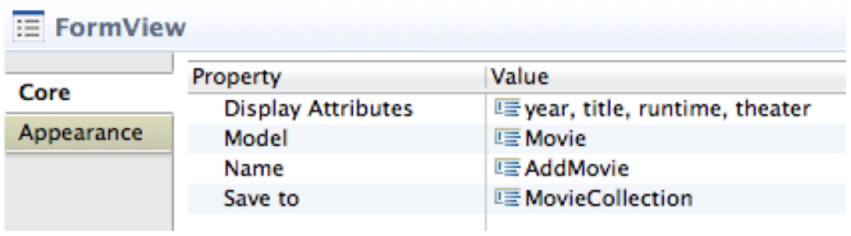


Figure 3.25: The application diagram created for the movie database example

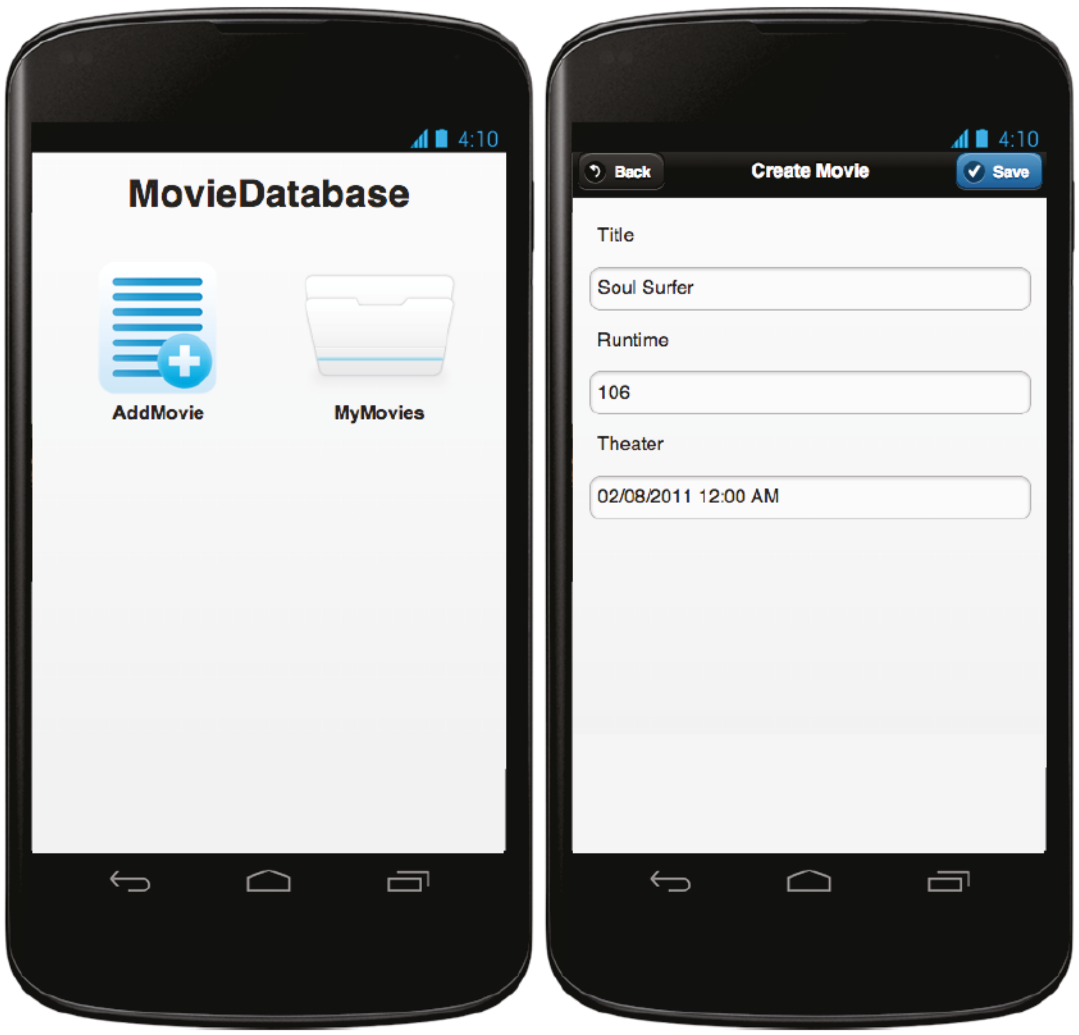

FormView

Core

Appearance

Property	Value
Display Attributes	year, title, runtime, theater
Model	Movie
Name	AddMovie
Save to	MovieCollection

Figure 3.26: The properties section of the Form View



Title

Soul Surfer

Runtime

106

Theater

02/08/2011 12:00 AM

(a) Dashboard View (b) Form View

Figure 3.27: Screenshots of the application generated based on the model extracted from the API invocation

3.9.2 Constructing Models from XSD Schemas

Another way of constructing models by using *WL++* is by giving as input an XML schema Definition (XSD). XSD schemas describe the structure of XML documents. *WL++* has a built-in XSD Parsing component which parses the given file in order to construct the proper models. Below we can see an example of an XSD schema (Figure 3.28).

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/
   XMLSchema">
<xs:element name="note">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="to" type="xs:string"/>
      <xs:element name="from" type="xs:string"/>
      <xs:element name="heading" type="xs:string"/>
      <xs:element name="body" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
</xs:schema>
```

Figure 3.28: An example of an XML schema. (Source: http://www.w3schools.com/schema/)

XSDs are composed of complex and primitive types of attributes. Unlike JSON, an XSD schema explicitly provides type information for each element attribute and therefore the type-inference step above is not necessary. For example, in Figure 3.28, the element *note* is a complex type composed of several elements: *to, from, heading* and *body* and all these elements are type of String as indicated by the "type" attribute of the element. The steps to be followed to generate a mobile application from the XSD schema of Figure 3.28 are listed below:

Step 1.

First, the tool parses the XSD schema in order to extract all the elements along with other information such as their "name" and "type" attributes. The parsing is done by using the Xerces2 Java Parser[11] library which is also able to validate XSDs.

[11] http://xerces.apache.org/xerces2-j/

Typically, each complex type that includes primitive types can be represented as a model. Therefore, after the parsing process is completed, the complex elements extracted from the XML schema can be converted into models. More specifically, a complex element represents a model and its primitive elements the model's attributes. Since the XSD schema has also the types of the attributes we do not need instance values as in the case of the JSON response to determine each attribute's type. In our example, *WL++* detects one model namely *note* including four String attributes (*to, from, heading* and *body*). The set of the detected models is then converted into an XMI representation (Figure 3.29) using our existing template (that was used previously during the models generation from JSON responses).

```
<?xml version="1.0" encoding="UTF-8"?>
<WLPP:JSModel xmi:version="2.0" xmlns:xmi="http://
   www.omg.org/XMI" xmlns:xsi="http://www.w3.org
   /2001/XMLSchema-instance" xmlns:WLPP="http://
   ualberta.ca"><modelCollection>
<Models Name="Note">
    <representations xsi:type="WLPP:Tex" Name="to"/>
    <representations xsi:type="WLPP:Text" Name="from
       "/>
    <representations xsi:type="WLPP:Text" Name="
       heading"/>
    <representations xsi:type="WLPP:Text" Name="body
       "/>
  </Models>
</modelCollection>
</WLPP:JSModel>
```

Figure 3.29: The XMI generated based on the XML schema in Figure 3.28

As long as the XMI representation of the models is generated, a *WL++* application diagram can be also generated. However, this diagram does not include views and navigations which have to be added manually by the developer. The models could be also modified by adding, editing, or deleting attributes. In this example we noticed that the labels of the attributes *to* and *from* are reserved words in SQL. Because SQL queries are executed to create the local database and to store/retrieve data, the developer must avoid using reserved words that may cause potential

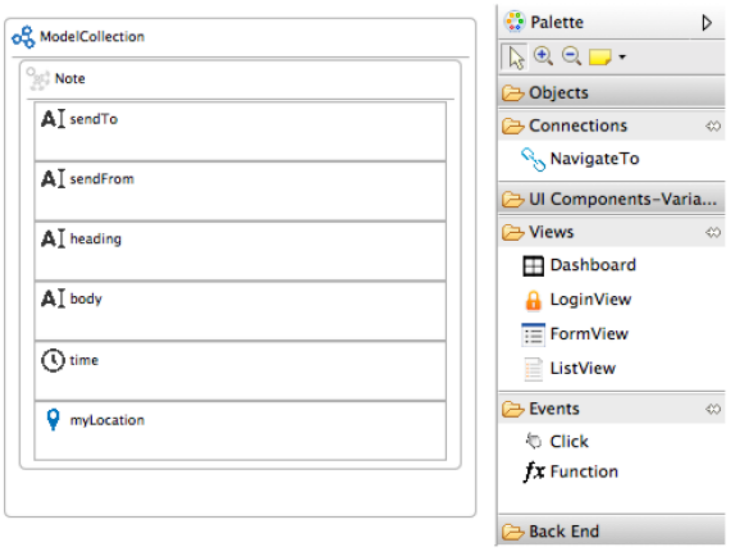


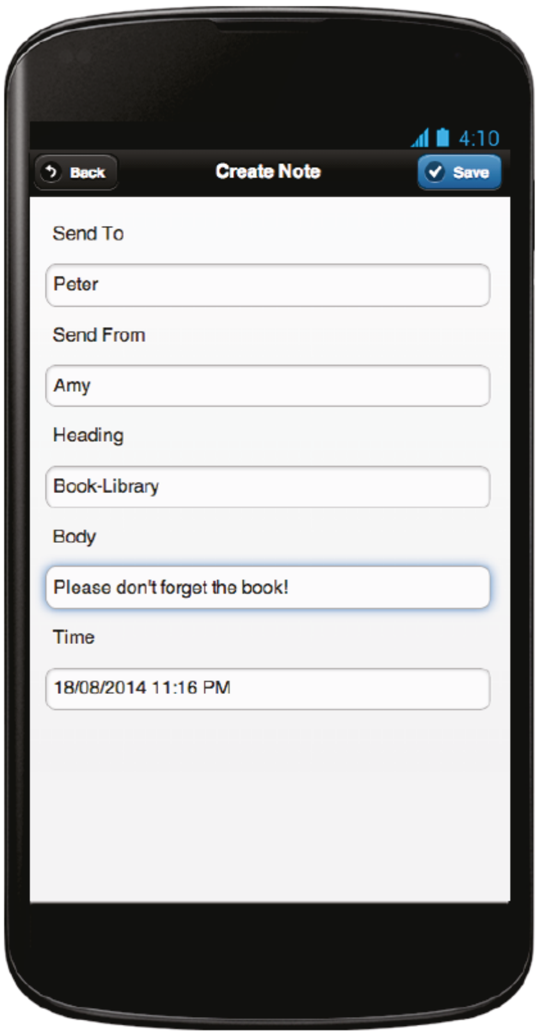
Figure 3.30: The model collection created from the XML schema. Two more attributes (time, myLocation) were manually added

conflicts. Therefore they should either modify these labels before generating and launching the application or change the model attributes's values after the application is generated. Consequently, we suppose that the developer decides to modify the *to* label to *sendTo* and the *from* label to *sendFrom* as the former solution is easier and more efficient. The developer also adds two more attributes to the model namely *time* which is a *Clock* widget and *myLocation* of type *Location* (Figure 3.30). The attribute *time* will be initialized with the devices local time and represent the date/time creation of the note whereas the *myLocation* attribute will indicate the user's location when the note is saved.

Step 2.

After the addition of the views and adapter(s) (if necessary) then an application (and back-end) are generated by clicking the "Generate Code" button. For this example, the developer has added only three views, a *Dashboard*, a *Form View* and a *List View* to display the stored notes. In Figure 3.31 the *Form View* of the application is depicted along with the *List View* were existing notes are displayed. As we can see in Figure 3.32 the location of the user is stored as a set of latitude and longitude values.

Step 3.



Send To

Peter

Send From

Amy

Heading

Book-Library

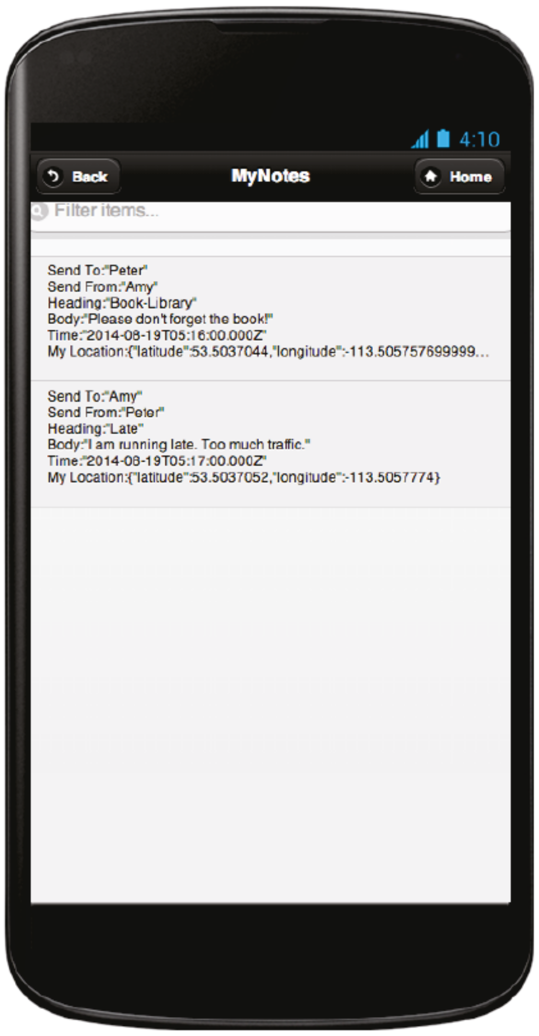
Body

Please don't forget the book!

Time

18/08/2014 11:16 PM

(a) Form View



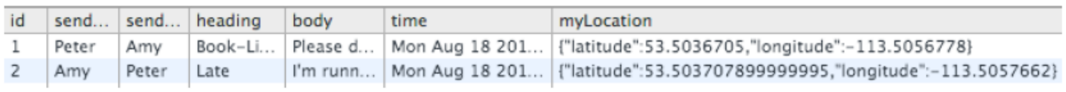
Send To:"Peter"
Send From:"Amy"
Heading:"Book-Library"
Body:"Please don't forget the book!"
Time:"2014-08-19T05:16:00.000Z"
My Location:{"latitude":53.5037044,"longitude":-113.50575769999...

Send To:"Amy"
Send From:"Peter"
Heading:"Late"
Body:"I am running late. Too much traffic."
Time:"2014-08-19T05:17:00.000Z"
My Location:{"latitude":53.5037052,"longitude":-113.5057774}

(b) List View

Figure 3.31: Screenshots of the NoteApp application generated based on the model extracted from the XSD schema

As the set of views has been constructed, the last step is to define the navigations among them by adding connection links that are translated into the application's router. Finally, if we suppose that there is a back-end service consuming XML data instead of JSON, the *WL++* plugin can produce XML instances of the data entered based on the model created following the XSD structure. According to our example, the XML produced by the addition of two data models is depicted in Figure 3.33.



id	send...	send...	heading	body	time	myLocation
1	Peter	Amy	Book-Li...	Please d...	Mon Aug 18 201...	{"latitude":53.5036705,"longitude":-113.5056778}
2	Amy	Peter	Late	I'm runn...	Mon Aug 18 201...	{"latitude":53.503707899999995,"longitude":-113.5057662}

Figure 3.32: The local database where the information of the NoteApp application is stored.

```
<note>
  <sendTo>Peter</sendTo>
  <sendFrom>Amy</sendFrom>
  <heading>Book-Library</heading>
  <body>Please don't forget the book!</body>
  <time>"2014-08-19T05::16:00.000Z"</time>
  <myLocation>{"latitude":53.527018399999996,"
     longitude":-113.52650299999999}</myLocation>
</note>
```

```
<note>
  <sendTo>Amy</sendTo>
  <sendFrom>Peter</sendFrom>
  <heading>Late</heading>
  <body>I'm running late. Too much traffic.</body>
  <time>"2014-08-19T05:17:00.000Z"</time>
  <myLocation>{"latitude":53.527009199999995,"longitude
     ":-113.52650639999999}</myLocation>
</note>
```

Figure 3.33: XML data produced after the creation of two sample notes

Chapter 4

Contact Manager: An Illustrative Example

In this chapter, we describe the process of generating a Contact Manager application by using the *WL++* application specification plugin. The application is composed of two parts (a) a client application that can be installed on devices supporting a wide variety of platforms and (b) a RESTful back-end service that allows the access of an external database for the purpose of storing and retrieving contacts. More specifically, the Contact Manager aims in adding contacts (along with a set of attributes such as name, phone number, e-mail etc.) by storing them locally on the mobile device and externally on a remote database. Also, these contacts should be displayed in a list where users would be able to edit and/or delete them. Finally, there is a filtering operation regarding contact's attributes; for instance, "search by name", "search by e-mail" or "search by custom query".

This type of application has been already built by developers multiple times as it is considered an application that includes many features and illustrates successfully the use of new frameworks/technologies. We found plenty of implementations in public code repositories and personal blogs [5, 22, 25]. Below we explain in detail the steps that need to be followed in order to generate a Contact Manager application using the *WL++* application specification plugin:

Step 1. Through the Eclipse IDE we first create a General Project (Figure 4.1).

Step 2. Then, we right click on the created project and select *New WLPP Diagram* which creates an instance of a *WL++* modeling diagram (Figure 4.2). We name this

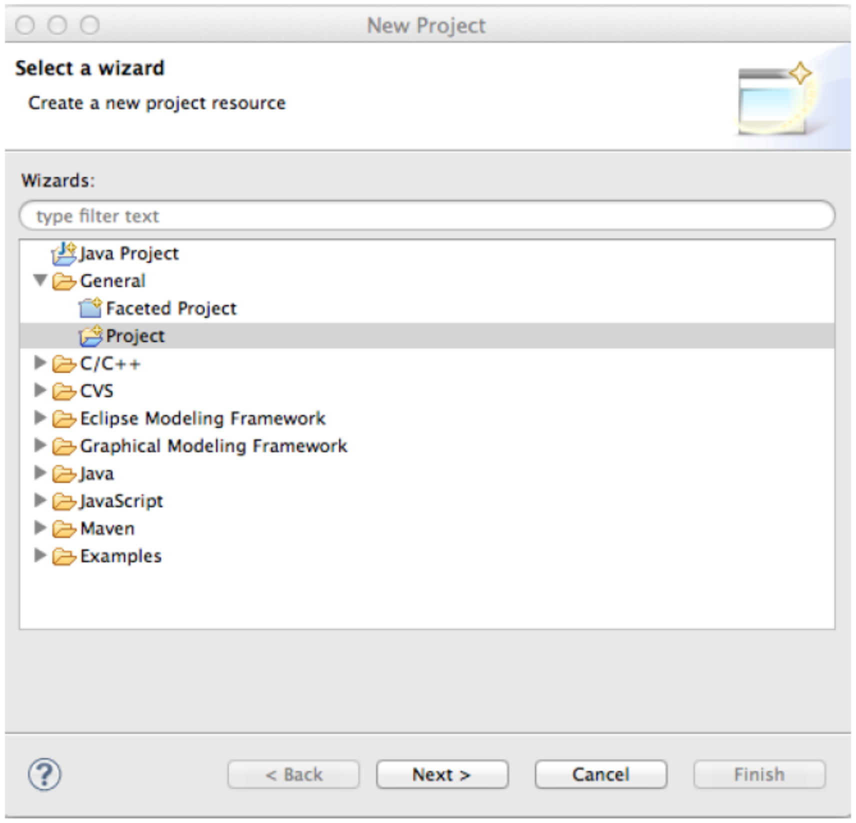


Figure 4.1: Creating a New General Project

metamodel instance "cm". A new set of files is created with extensions *.wlpp* and *.wlpp_diagram* (Figure 4.3). The former includes information regarding the domain model while the latter contains graphical related information. Both these files are tightly associated as when the *.wlpp_diagram* file is modified and the "Save" button is clicked, then the *.wlpp* file is automatically updated with the new information. Similarly, when the the *.wlpp* file is modified, its graphical representation is also updated. After a new project is created, the *.wlpp_diagram* file is selected by default and therefore we are able to see the tools that are available for creating a diagram in the palette.

Step 3. The diagram is composed of two major components: (a) the application diagram which contains the views that will be added later on, and (b) the application model collection for the models to be defined. Therefore, we first drag the *Application Diagram* widget from the palette (from the Objects cluster) and drop it in the editing area. We name it "Contact Manager" as this indicates the application's

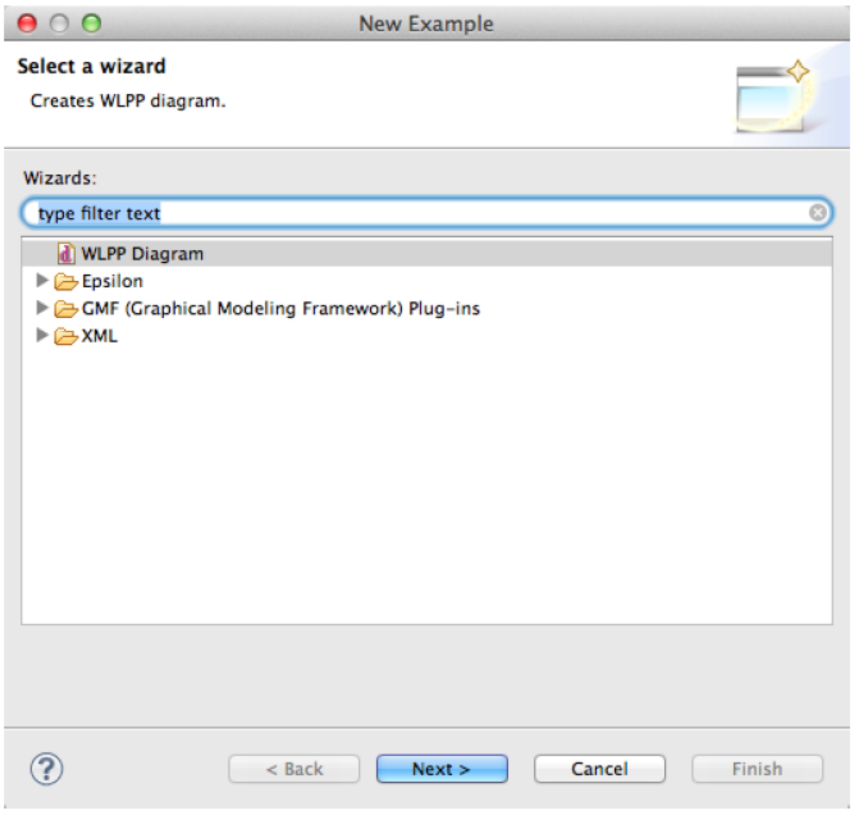


Figure 4.2: Creating a New Metamodel Instance

title. Then, we drag and drop a *Model Collection* widget next to our *Application Diagram* (Figure 4.4).

Step 4. At this point the developer has to decide the models that the application is composed of. This process depends on the developer and their perception of the application. In the case of the Contact Manager, we assume that there is a model representing a Contact. Therefore, we select the *Model* widget from the palette, drag it to the editing area and label it "Contact".

Next, we add a set of attributes to the Contact model. There is no limit on the number of attributes to be added and therefore we can add all the attributes we consider important for a typical Contact Manager application. For this particular example, we were influenced by a similar application, which was developed using the Phonegap framework [5]. Therefore, we included some of the properties belonging to the Employee Directory application (i.e., *Name, Last Name, Phone Number, E-mail and Picture*). These properties are declared within our model by

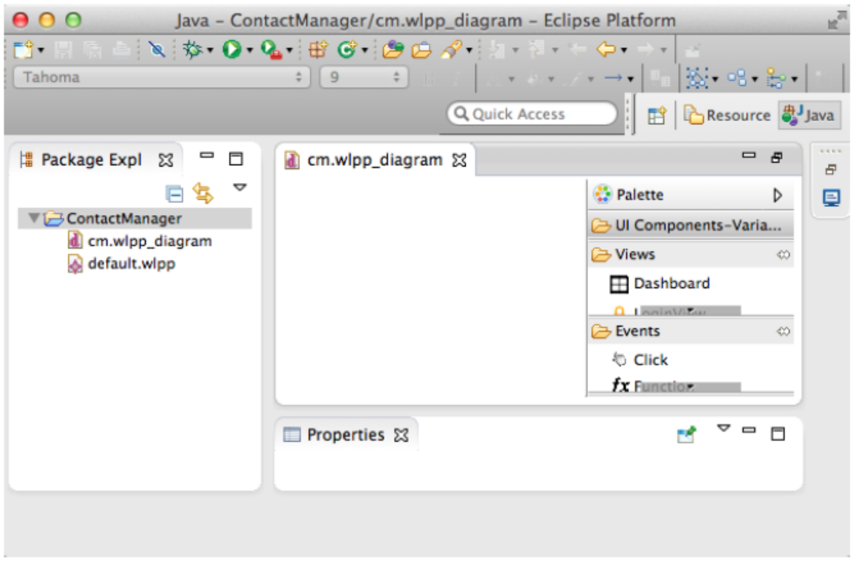


Figure 4.3: Application Specification Plugin User Interface

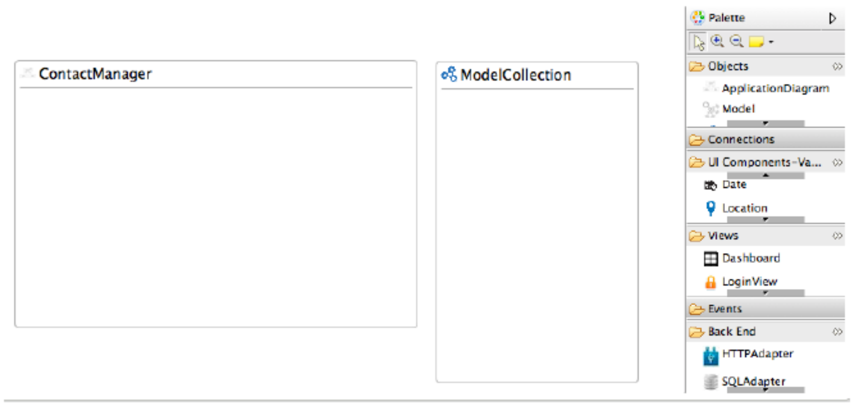


Figure 4.4: The application diagram after the addition of the *Application Diagram* and the *Model Collection* widgets

selecting the proper widget types. Therefore, for the properties *Name, Last Name* and *E-mail* we selected the **Text** widget as it represents a text field on the application's graphical user interface whereas for the *Phone Number* attribute we selected the **Numeric** widget. The latter appears as a number field on the user interface (thus, it only accepts numerical values). We decided to have a *Birthday* property so we added a **Date** widget as well. Finally, for the *Picture* attribute we dragged and dropped the **Image** widget to our model. This widget is composed of an image box and a button to select images from the mobile device's image library. The order in which the aforementioned attributes were added to our model is the same as the one they appear on the form view to be displayed on the mobile device. In the case where we would like to change the properties' order (for example, to have the property *E-mail* appear before the property *Phone Number*), we could simply modify the order of the properties' list on the application diagram and re-generate the application). Alternatively, we could modify the generated HTML file (Figure 4.5).

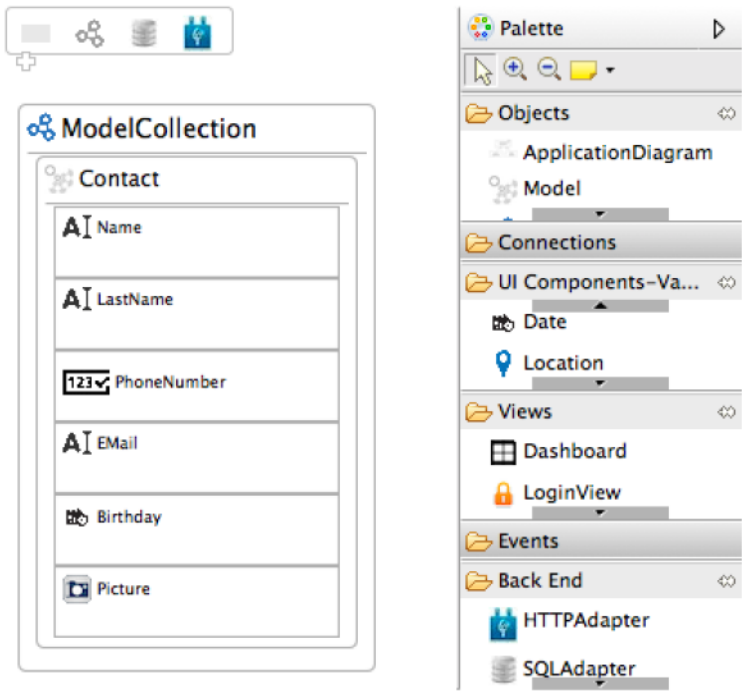


Figure 4.5: The model collection including a model namely "Contact" along with several representation

Step 5. Next, we add the proper views to the *Application Diagram*. First we drag and drop a **Dashboard** view widget which contains two more views and serves the purpose of navigation through the different views. The "AddContact" view is a *Form View* and must be connected by a *Navigation* type of connection to the dashboard in case where we would like to navigate to this view via the dashboard. If we decide our contacts to be displayed in a list we should also add a *List View* and connect it to our dashboard (Figure 4.6). So far, we already have the application diagram that results to a cross-platform mobile application to be deployed locally on the mobile device storing our contacts' information on a Web SQL database (device browser's storage) [39].

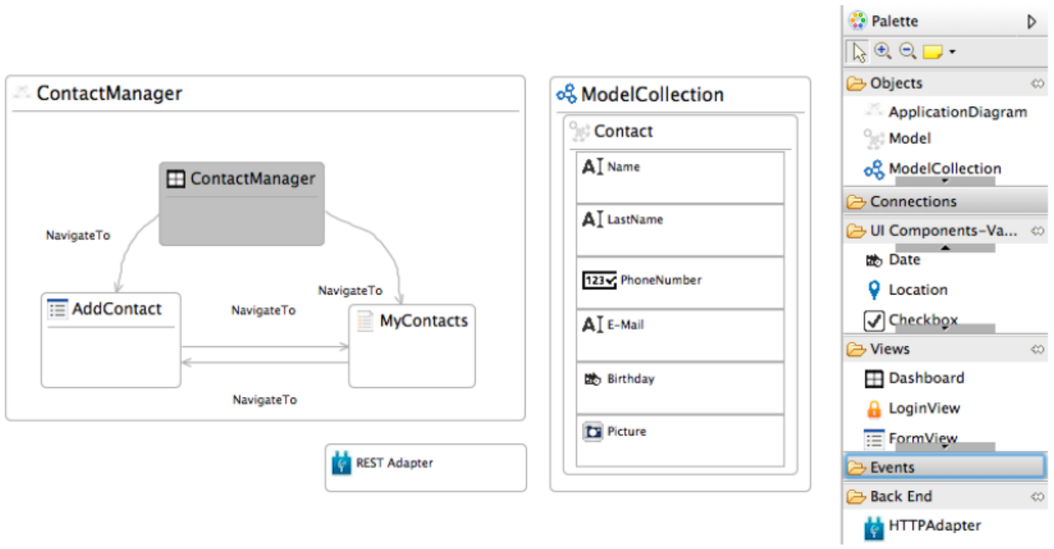


Figure 4.6: The Contact Manager Application Diagram

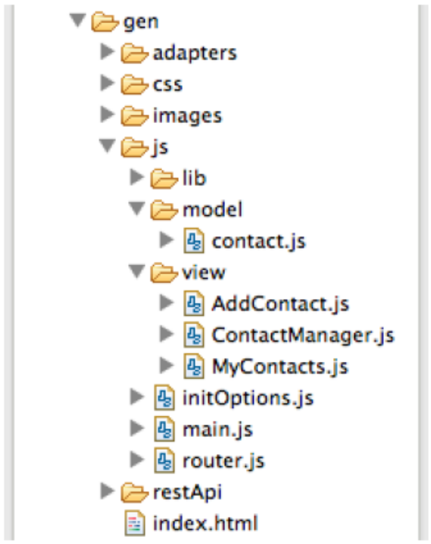
Step 6. We continue by adding a *Worklight Adapter* (Figure 4.6) to generate a back-end service for our application by dragging the *HTTP Adapter* widget to the editing area (an adapter cannot be added in the *Application Diagram* or the *Model Collection*). Along with the adapter, a RESTful-service that not only supports CRUD and filtering operations but also creates the database structure is generated. The database structure is defined by our model's attributes. For example, in our case the database's table would be composed of seven columns namely *id*, *Name*, *LastName*, *PhoneNumber*, *EMail*, *Birthday* and *Picture* (for the picture's path).

Step 7. Finally, in order to generate the output, we click the "Generate Code" button which is located at the bottom of the palette. When the code generation process is completed, a folder containing the application's files namely "gen" is generated. Figure 4.7 depicts the structure of files of the output folder. The "adapters" folder is generated only when there is a Worklight adapter on the application's diagram. The "js" folder has the JavaScript code (libraries, models, views, router) of the mobile application. Finally, the "restApi" folder contains the files related to the RESTful web service. In Figure 4.8, we can see a subset of the REST APIs generated. The paths are constructed based on the adapter's name, the application's name and the model's name which the API is referred to. All the contacts stored in the database can be also accessed through a browser by entering the following URL (if using a server that runs locally on port 8080):

"http://localhost:8080/REST/ContactManager/Contacts"

The above URL triggeres the "getContacts" method which when executed returns all the contacts stored in the database in a JSON format. Through the following URLs we can search contacts by attributes such as (a) by id, (b) by name or (c) by a customized query such as "id >1 and name='Ray'":

(a) "http://localhost:8080/REST/ContactManager/Contacts/search/id",
(b) "http://localhost:8080/REST/ContactManager/Contacts/search/name",
(c) "http://localhost:8080/REST/ContactManager/Contacts/search/q"



```
▼ gen
    ▶ adapters
    ▶ css
    ▶ images
    ▼ js
        ▶ lib
        ▼ model
            ▶ contact.js
        ▼ view
            ▶ AddContact.js
            ▶ ContactManager.js
            ▶ MyContacts.js
        ▶ initOptions.js
        ▶ main.js
        ▶ router.js
    ▶ restApi
      index.html
```

Figure 4.7: The file structure of the generated folder

In order to deploy the generated application, the output files that correspond to the application need to be copied to a Worklight project and the ones related to the RESTful web service to a web service project. The IBM Worklight framework allows developers to add different environments and therefore support multiple mobile platforms. By doing this, it is easy to export installation files for each platform and use them to install the generated application on real mobile devices. More specifically, the effort of a developer that needs to deploy the generated application in different platforms has to do with the generation and extraction of the installation files specific to each platform. On the other hand, the generated RESTful web service must be uploaded on an online server so that the application is able to access the external database through the APIs that were defined. Each API call is monitored by the monitoring service that keeps a record of it on a table of the external database. The specific API and the data sent through the service are stored in a JSON format.

Figure 4.10 depicts the storyboard of the Contact Manager application (four views and the available navigations from one view to another). For instance, users can navigate from the "Dashboard" to the "Add Contact" or "My Contacts" view and from the "Add Contact" to the "My Contacts" view. The navigation is possible through the event handling that exists within the views (e.g., when the *Save* button is clicked on the "Add Contact" view, then the application navigates to the list view that contains the contacts). From the list view, end-users can either navigate back to the dashboard or edit a specific entry ("Edit Contact"). Finally, Figure 4.9 shows three different screenshots of the generated application (Figure 4.9a, 4.9b, 4.9c) along with two screenshots of the Employee Directory that was developed from scratch by Christophe Coenraets (Figure 4.9e, 4.9e). The first two ((Figure 4.9a, 4.9b) represent the data entry screen (Form View) which is conceptually related to the Employee Directory view in Figure 4.9d whereas the Figures 4.9c and 4.9e show the entries of the Contact Manager and Employee Directory respectively in a list view.

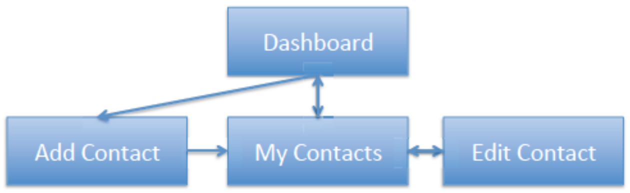
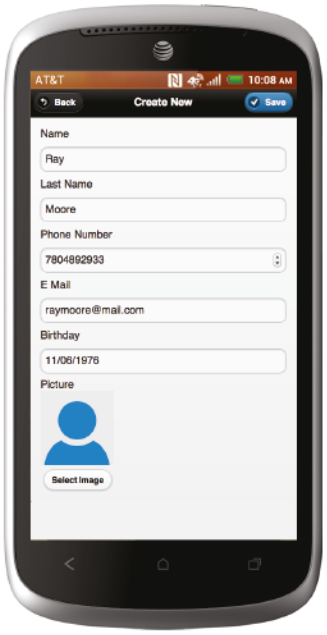


Figure 4.10: Storyboard of the Contact Manager Application

```
@Path("/Contacts")
public class Rest {

  private DatabaseCRUD DB;

  public Rest() throws ClassNotFoundException, SQLException{
    DB = new DatabaseCRUD();
  }

  @POST
  @Path("/createDB")
  @Produces(MediaType.TEXT_PLAIN)
  public String createDB() throws ClassNotFoundException, SQLException{
    DB.createDB();
    return "Database Created";
  }

  @GET
  @Produces(MediaType.JSON)
  public String getContacts() throws ClassNotFoundException, SQLException{
    return DB.getContacts();
  }

  @GET
  @Path("search/{id}")
  @Produces(MediaType.JSON)
  public String getContactById(@PathParam("id") String id) throws SQLException, ClassNotFoundException{
    return DB.getContactById(id);
  }

  @GET
  @Path("search/{name}")
  @Produces(MediaType.JSON)
  public String getContactByName(@PathParam(''name'') String name) throws SQLException, ClassNotFoundException{
    return DB.getContactByName(Name);
  }

  @GET
  @Path("search/q")
  @Produces(MediaType.JSON)
  public String getContactByQuery(@Context UriInfo uriInfo) throws SQLException, ClassNotFoundException{
    String query = uriInfo.getRequestUri().getQuery();
    return DB.getContactByQuery(query);
  }
}
```

Figure 4.8: A subset of the REST APIs generated by *WL++* code generation engine



AT&T 10:08 AM

Back Create New Save

Name

Ray

Last Name

Moore

Phone Number

7804892933

E Mail

raymoore@mail.com

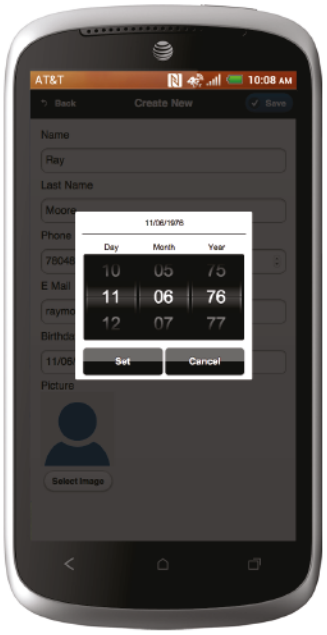
Birthday

11/06/1976

Picture

Select Image

(a)

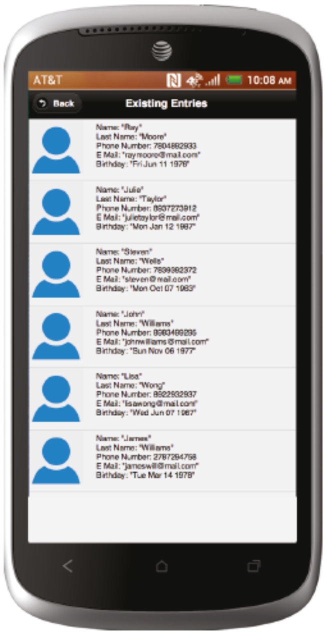


11/06/1976

Day	Month	Year
10	05	75
11	06	76
12	07	77

Set Cancel

(b)



AT&T 10:08 AM

Back Existing Entries

Name: "Ray"
Last Name: "Moore"
Phone Number: 7804892933
E Mail: "raymoore@mail.com"
Birthday: "Fri Jun 11 1976"

Name: "Julie"
Last Name: "Taylor"
Phone Number: 8937273912
E Mail: "julietaylor@mail.com"
Birthday: "Mon Jan 12 1987"

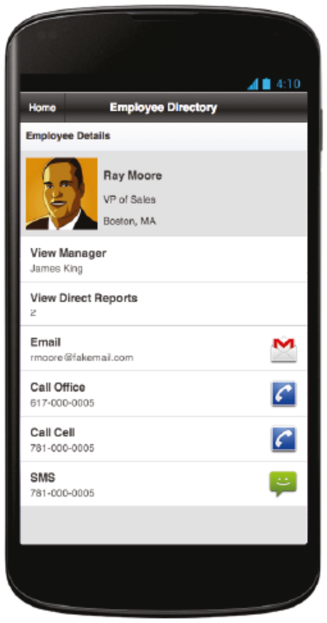
Name: "Steven"
Last Name: "Wells"
Phone Number: 7839392372
E Mail: "steven@mail.com"
Birthday: "Mon Oct 07 1963"

Name: "John"
Last Name: "Williams"
Phone Number: 8983489295
E Mail: "johnwilliams@mail.com"
Birthday: "Sun Nov 06 1977"

Name: "Lisa"
Last Name: "Wong"
Phone Number: 8922932937
E Mail: "lisawong@mail.com"
Birthday: "Wed Jun 07 1967"

Name: "James"
Last Name: "Williams"
Phone Number: 2787294758
E Mail: "jameswill@mail.com"
Birthday: "Tue Mar 14 1978"

(c)



4:10

Home Employee Directory

Employee Details

Ray Moore
VP of Sales
Boston, MA

View Manager
James King

View Direct Reports
2

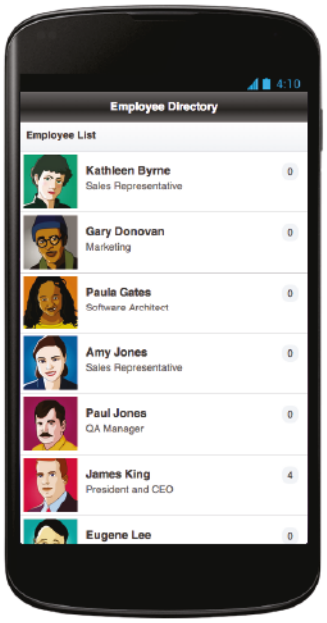
Email
rmoore@fakemail.com

Call Office
617-000-0005

Call Cell
781-000-0005

SMS
781-000-0005

(d)



4:10

Employee Directory

Employee List

Kathleen Byrne Sales Representative	0
Gary Donovan Marketing	0
Paula Gates Software Architect	0
Amy Jones Sales Representative	0
Paul Jones QA Manager	0
James King President and CEO	4
Eugene Lee	0

(e)

Figure 4.9: The Contact Manager app generated with *WL++* (a-c) and the Employee Directory app [5] (d-e)

4.1 Contact Manager vs. Employee Directory

In this Section we compare the Contact Manager application generated by *WL++* with the Employee Directory in terms of functionality and lines of code (LOC) for different file categories (Table 4.2). The latter is a Phonegap application developed manually by Christophe Coenraets [5].

4.1.1 Functionality Comparison

Both applications have similarities in terms of functionality and architecture as they both use the Backbone.js library. In the case of the Contact Manager, the main model is a single Contact whereas in the case of the Employee Directory the model is an Employee. Both applications can display an instance of the model and also all the existing models included in the model collection (Figure 4.9a-4.9d, 4.9c-4.9e). There is also a "Home" button through which users can navigate to the Home page and a "Search" input filed to search for contacts/employees based on specific keywords. However, in the case of the Contact Manager, the Contacts are added one by one by the users using an input form. On the other hand, the Employee Directory assumes that there is a database that includes all the employees and therefore it uses several SQL queries to populate the list of employees. Table 4.1 presentes the similarities and differences of both apps in terms of functionality.

Functionality	**Employee Directory**	**Contact Manager**
Insert new entries	Form	Database
View a single entry	✓	✓
Browse through a list of entries	✓	✓
Search entries based on keywords	✓	✓
Navigate to home screen	✓	✓

Table 4.1: Comparison of the Employee Directory app and the Contact Manager app in terms of functionality

4.1.2 LOC Comparison

In total the Employee Directory is composed of 1344 LOC whereas the Contact Manager application has 753 LOC (excluding the LOC belonging to the back-end as the Employee application does not have a back-end implementation). The SQL queries regarding the population of the list of employees consist of 116 LOC. However, this functionality can be mapped to the form view of the Contact Manager application that is used as an input form to add new contacts. Below follows a description of the differences between the two application in terms of functionality, file categories, and lines of code:

- *HTML:* This category includes the LOC of all the HTML files. Both applications are single page applications and therefore, there is one HTML file which contains the most essential templates/library imports that are always present in every application.

	HTML	Models	Views	Other JavaScript	Templates	CSS	WL Adapters	REST service	Total (excluding back-end)
Employee Directory	26	31	202	469	167	449	0	0	1344
Contact Manager	92	55	196	117	21	272	158	445	753

Table 4.2: Comparison of the Employee Directory App and Contact Manager App based on the Lines of Code (LOC)

- *Models:* The Employee Directory application architecture is very similar to the Contact Manager as they both leverage the Backbone.js library to separate the user interface from the application logic. Therefore, we compared the LOC of the model defined on the Employee Directory (model: Employee) over the LOC defined on the Contact Manager (model: Contact).

- *Views:* In the case of the Employee Directory app, there is a single file representing the views of the application which consists of 202 LOC. On the other hand, Contact Manager has three separate files representing each view. *WL++* generates a different file for each view in order to facilitate the maintenance of the application in the later stages of development. The LOC in the case of the Contact Manager application are in total 196.

- *Other JavaScript:* There are additional JavaScript files in both applications. The Employee Directory handles the communication with the device's database by doing a manual construction of SQL queries in order to fetch/store data in the database. However, for this purpose, *WL++* uses the

Backbone-WebSQL[1] library (as mentioned in Section 3.6.2, which is responsible for the retrieval/storage of the data. Furthermore, in this category, we have included the file including the code that instantiates all the models, collections, views and the local database.

- *Templates:* Although there are multiple templates defined, we counted the LOC of the templates that were used for this application. More specifically, we calculated the total LOC of the templates by subtracting the LOC of the initial HTML file (before the code generation) and the HTML file (after the code generation). By doing this, we know how many LOC have been added in the initial HTML file and therefore, how many LOC of the templates have been used for this specific application. In our case there were 21 LOC added. On the other hand the Employee Directory has five templates (167 LOC in total), which are used when users navigate from one screen to another.

- *CSS:* We defined our CSS file as a static file, which is the same for every application to be generated. This file is copied in the CSS folder of the generated application the same way the libraries are copied to the lib folder. This way, more CSS files can be added and switched according to the developer's preferences. In our case, the CSS file has 272 LOC compared to the Employee Directory that has 449 LOC.

- *WL adapters & Restful Web Service:* Although, the Employee Directory does not have a back-end, we have included the *WL++*-adapters and the RESTful service back-end in the chart in order to calculate the total LOC generated by *WL++*. Since the back-end is composed of the adapters and the web-service, in total there are 603 LOC.

[1]https://github.com/MarrLiss/backbone-websql

Chapter 5

Evaluating the *WL++* Framework

In the previous chapter we have already described the steps followed to develop the Contact Manager application by using *WL++* as proof of concept. In this chapter, as a means of evaluating our *WL++* framework, we describe two real-world applications we have developed with *WL++*: (a) a physical activity tracking application to increase one's daily physical activity (Section 5.1) and (b) a patients' visiting application to support nurses going through their rounds (Section 5.2). For each of these applications, we discuss the motivation behind their development and the development process using *WL++*. We also report some descriptive statistics on the application size and complexity reflecting the degree to which *WL++* contributed to the overall application development.

5.1 Physitivity: A Physical Activity Tracking App

Physitivity is a cross platform mobile application that aims to encourage users to increase their daily physical activity. When installed, it is aware of a few typical activities (walk, run, bike or climbing stairs) but, more interestingly, it enables users to define new types of activities based on their own everyday life, such as doing laundry or shoveling snow for example. The application is also aware of the user's calendar and location. While active, the application locates the user's location every 15 minutes, suggests nearby activities based on the user's calendar availability and displays an alert. When users engages in an activity, they can record the activity's duration and intensity, in units relative to the activity. For example, when recording stair climbing, the number of steps is recorded and the qualitative tempo of the activity, i.e., light, moderate, and vigorous. Additionally, users may provide meaningful tags to each activity. At any time, users may see their history, based on the activity types and chosen tags, or a graphical overview. When the physical activity is outdoors, the app uses the smart-phone GPS to simplify (and improve the quality of) data recording. If users plan to perform an activity such as walk, run or bike from their current location to a destination location, they can simply indicate the two locations on a map and start. Until users indicate that an activity has stopped, the device GPS will periodically record the user's current location and thus record the activity trace and time. Of course, users can always add new past entries and edit or delete existing ones.

Physitivity has been developed before *WL++* as part of the "Smart-Condo" course based on requirements developed with Occupational Therapists and Physical-Activity experts during this course. Simultaneously, EASI, an application for people with diabetes was also developed by a different team within the context of the same course at during the same timeframe. With EASI users were able to record their blood glucose, food, physical activity and insulin intake [35]. Based on the experience gained from both these applications in terms of architecture, functionality and user interface components, *WL++* was created. For evaluation purposes, we attempted to re-develop Physitivity by using the *WL++* framework. The application diagram of Physitivity is shown in Figure 5.1. Below are listed the main steps to be followed in order to create the application diagram and generate Physitivity:

Step 1.

First we add the mandatory components (*Application Diagram* for the views and *Model Collection* for the models). Next, we continue by defining the models along with their attributes within the *Model Collection* section. In this case, the *Model Collection* is composed of five models namely: "Activity", "Walk", "Run", "Personal Activity" and "Stairs". The "Walk" and "Run" activities are similar conceptually and therefore share similar attributes. More specifically, several attributes were added for these activities namely: fromLocation, toLocation, fromTime, toTime, duration, light, moderate, vigorous.

For the **fromLocation** and **toLocation** attributes we drag and drop the *Text* widget from the palette. As a result, both these attributes are displayed as text input fields where end-users are able to type their location by giving meaningful tags (e.g., "home", "office", "university" etc.). For the **fromTime** and **toTime** attributes the *Clock* widget is selected as it needs to be initialized with the device's local time. Since we have the time an activity starts and ends, let's supposed that the application should also display the duration of an activity to the users. For this purpose, we also add a *Numeric* widget namely **duration**. Later on we will explain where the calculation takes place and how it is indicated by the developer though the application diagram. Finally, three *Checkbox* widgets are added representing the activity's intensity (light, moderate and vigorous). Ideally, a Radio Button widget would be more suitable in this case since only one option has to be selected. However, as currently the plugin does not support a radio button widget, we chose the checkbox one assuming that end-users will only check one of the three available options.

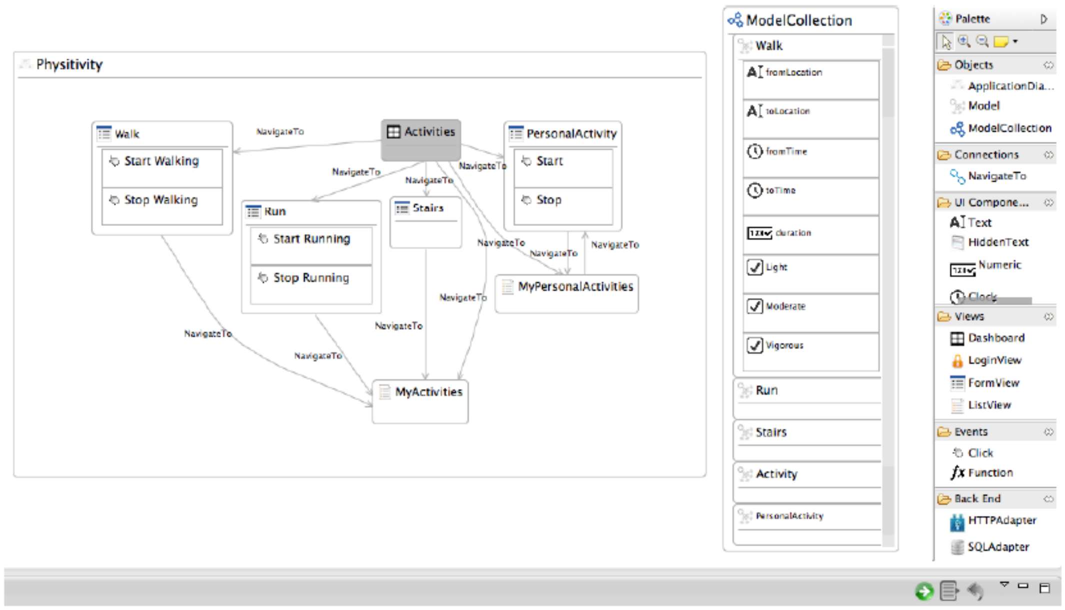


Figure 5.1: Physitivity Application Diagram

Next, we add a "Stairs" model which is composed of three attributes: (a) description, (b) location and (c) stairs. The **description** is a *Text* widget and represents the location of the stairs taken. Users can specify the location by using one or multiple tags. The **location** attribute is a *Location* widget that gets the current location of the user using the mobile device's GPS. The *Location* widget does not have a user interface component but is only represented by a set of latitude-longitude values that are stored on the device's local storage. Finally, the **stairs** attribute is a *Numeric* widget representing the total number of the stairs taken.

Step 2.

After the definition of the models, we populate the *Application Diagram* component with the essential views. The main screen of the view to be displayed when the application is launched is a *Dashboard View* (namely **Activities**) which leads to the rest of the views (the ones that are connected to the Dashboard). For each Activity within the *Model Collection*, we add a *Form View* and thus, there are four form views in total: Walk, Run, Stairs, Personal Activity. Finally, we continue by adding two separated *List Views*: (a) My Activities which includes all the activities except from Personal Activities and (b) My Personal Activities used to store only personal activities. The form/list views are associated with a particular (collection of) model(s) through their properties (specified in the *WL++* property console).

Step 3.

So far, only the models and the views have been specified. However, we need to add the navigations from one view to another. The navigation is indicated by adding navigation links among the views to be connected. As we can see in Figure 5.1, the Activities Dashboard screen leads to all the Form views (Walk, Run, Stairs, Personal Activity) and also to the List views (MyActivities, My Personal Activities). Furthermore, when a Run or Stairs activity is added, end-users navigate to the My Activities list whereas in the case of a Personal Activity in the My Personal Activities list.

To trigger events, the *Click* widget from the Events cluster is selected. By adding this widget to the views, there is a click event that can be triggered under specific conditions specified through the *Click* widget properties. First, we specify an ID for the event and as explained in section 3.4 in the case where this ID does not exist, a new event is created. For instance, the Walk view has two such events: (a) start and (b) end. The property “Set Attribute Value” of the Click widget has been set as : “fromTime=new Date()”. Thus, when the “Start Walking” button is clicked, the **fromTime** attribute of the Walk model is initialized with the device’s local time. Similarly, when the “Stop Walking” button is clicked, the **toTime** attribute is initialized with the device’s time. By having the starting/ending time of the activity, we can simply calculate and set the value of the duration attribute when the “Save” button is clicked.

When the Application Diagram and the Model Collection are populated, then the “Generate Code” button must be clicked to generate the Physitivity application.

5.1.1 Physitivity vs *WL++* Physitivity

In this section we compare the two applications: (a) the Physitivity developed as a course project and (b) the Physitivity generated using our *WL++* framework in terms of functionality along with some code statistics.

5.1.1.1 Functionality Comparison

As described before, Physitivity exhibits some features that are not generated through *WL++*. First, it uses the mobile-phone’s GPS to locate the user and it integrates Google maps. Therefore, it is able to display one’s walking/running path on a Google map by locating the user every minute. It is also integrated with the Google calendar service (as long as the user logs in with a valid Google account) which it uses to send notifications regarding activities to be performed while the users are not busy (according to their calendar’s free time slots). Despite of the user’s availability, the recommendations are made based on the user’s current location and history of activities. More specifically, when a user is located within a range of 500 meters of the place where an activity has already taken place, the user is notified to re-perform this particular activity. Furtherfore, it uses the device’s accelerometer to count the number of stairs when a stairs activity starts. Finally, all the activities are presented to the users through a graphical representation for an easier overview (per month, week, day) that shows the overall progress.

By using our *WL++* framework we were able to generate a simpler version of this application. In Table 5.1 we compare the two applications in terms of functionality. The cells indicated with ***x*** represent the functionality that currently cannot be generated through the *WL++* framework. However, the implementation of the accelerometer, graphical representations of data as well as the integration of google services is a matter of extensions to the current plugin. Similarly to the Image widget that is able to take an image using the device's camera or choosing an existing image, other components that access the device's features through the Cordova APIs can be added by extending the *WL++* application specification plugin.

Functionality	**Physitivity**	***WL++* - Physitivity**
Dashboard	✓	✓
View per Activity	✓	✓
Add/Edit/Delete Activity	✓	✓
Tagging Activities	✓	✓
Color Themed Activities	✓	x
Date/Time Picker	✓	✓
GPS Location	✓	✓
Google Maps	✓	x
Activity Intensity	✓	✓
History of Activities	✓	✓
Back-Button Navigation	✓	✓
Home Navigation	✓	✓
Accelerometer	✓	x
Google Calendar	✓	x
Notifications	✓	x
Filtering	✓	✓
Compute Duration/Activity	✓	✓
Graphical Representation/Chart	✓	x

Table 5.1: Comparison of the Physitivity app with the *WL++* Physitivity app in terms of functionality

5.1.1.2 LOC Comparison

In order to give an overview of the amount of code generated automatically through the framework, we categorized the files into specific categories similarly as we did with the Contact Manager application. Table 5.2 depicts the lines of code of the Physitivity compared to the lines of code of the application generated with *WL++*. First, we observe that for each file category the prototype application prevails regarding the lines of code. Both applications have the same models except from the Biking model (included in the prototype app). Because the functionality is similar to the Walk/Run activities, the Biking activity is not included in the auto-generated application. As we can see in Table 5.2, the number of LOC regarding the *Models* is very close for the two applications (266 and 213 for the Physitivity and *WL++* Physitivity respectively). The *HTML* file category as for Physitivity has 618 LOC compared to 135 for the *WL++* Physitivity. The former includes more templates and has slightly more complicated user interface. Moreover, for each activity users can either select to add a new activity that will be performed at the same time or a past activity. For all these cases, there are different templates although a single template could be used and changed on the fly when needed. This is a bad practice which leads to a high number of LOC and makes the maintenance difficult. The same practice is observed with the *Views* and the *Templates.* The *CSS* related files contain themes and more specifically a special JQuery theme that was needed in order to be able to mark each activity category with a different color. Finally, the *Other Javascript* category includes files such as the Backbone router and the main javascript file responsible for the launch of the application.

	HTML	**Models**	**Views**	**Other JavaScript**	**Templates**	**CSS**	**Total**	**Total (Excluding LOC not used)**
Physitivity	618	226	1741	430	534	1478	5067	2473
WL++- ++ Physitivity	135	213	771	164	89	272	1644	1644

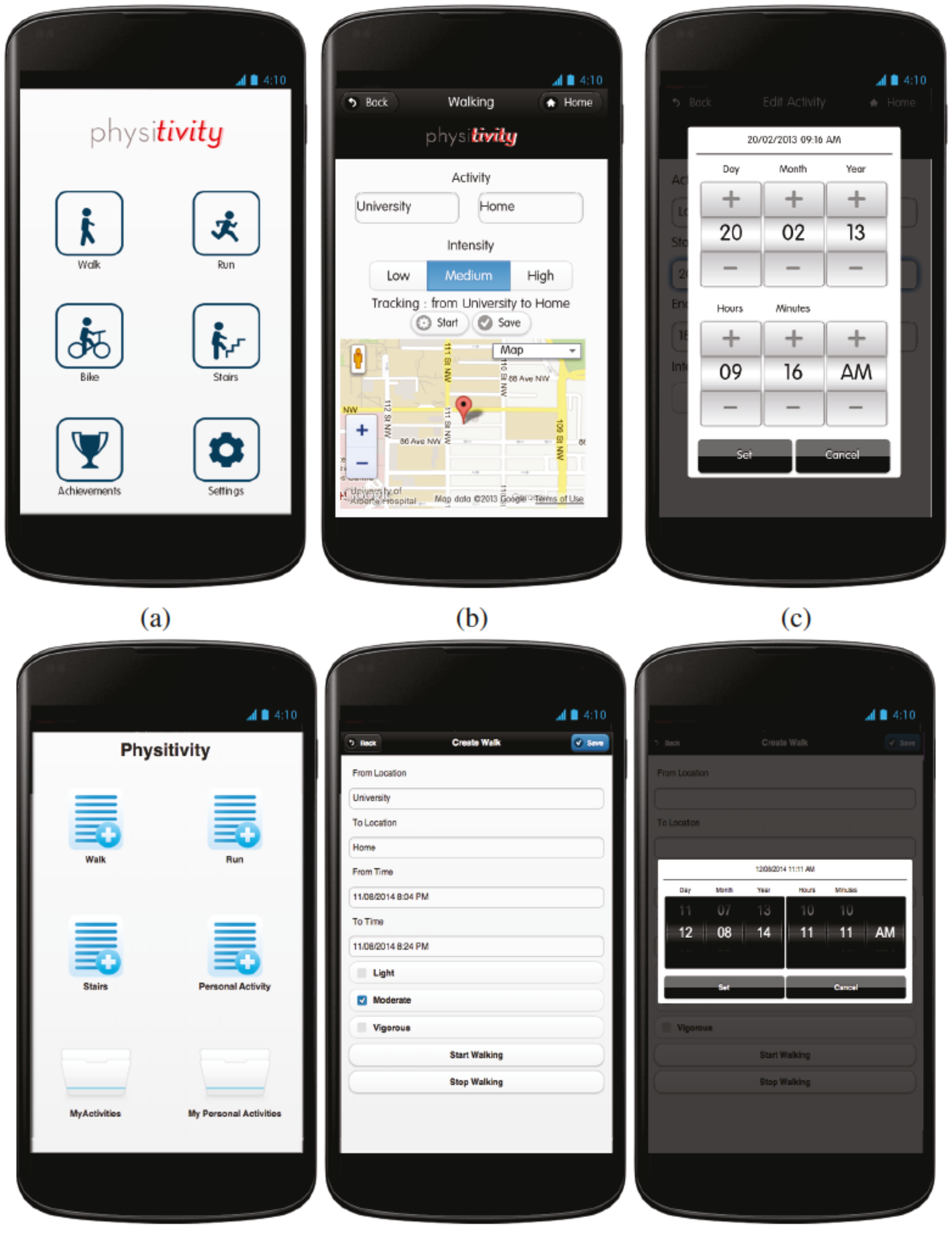
Table 5.2: Comparison of the Physitivity app and *WL++* Physitivity app in terms of Lines of Code (LOC)

As mentioned above, the prototype application includes pieces of functionality that were not generated by using the current version of *WL++*. For each one of the requirements (listed in Table 5.2 and marked with **x**), we counted the LOC referring to each one of them. In other words, the LOC for each requirement indicate the manual effort needed to implement a specific functionality. Therefore, in Table 5.3 we can see the total LOC for each functionality that is included in the prototype Physitivity app but not generated in the Physitivity version developed using the *WL++* framework. In total there are 829 LOC that need to be manually added in the generated version of Physitivity (1644 LOC). This essentially means that developers using the *WL++* plugin would generate 66,48% of the Physitivity code. Furthermore, since Physitivity has already gone through maintenance (few requirements changed, and Backbone.js was integrated in a later phase of the development), there are in total 2594 LOC that are not used from the application (although these parts of code are spread throughout the app, a big part of it resides in the CSS file).

Functionality	**HTML**	**CSS**	**JavaScript**	**Total LOC**
Color Themed Activities	1	0	20	21
Google Maps	7	0	238	245
Accelerometer	0	0	79	79
Google Calendar	0	0	98	98
Notifications	9	9	95	95
Graphical Representation/Chart	55	14	222	291
Total	63	14	752	829

Table 5.3: LOC of the functionality that has been implemented in Physitivity but not generated in *WL++*-Physitivity

The top layer of Figure 5.2 shows three screenshots of the Physitivity application (Figure 5.2a, 5.2b, 5.2c) whereas the bottom layer the corresponding screenshots of the same application generated using the *WL++* framework (Figure 5.2d, 5.2e, 5.2f). As we can see, the dashboards look similar in terms of the layout used (Figure 5.2a, 5.2d). The Physitivity application has two additional functionalities (achievements and settings) that are not included in the *WL++* Physitivity. The second set of screenshots ((Figure 5.2b, 5.2e) shows an instance of the addition of a new Walking activity. As mentioned before, Physitivity has been integrated with Google Maps and requires a valid Internet connection to be able to keep track of the user's location. Besides that, the two applications have similar headers (both include a "Back" button which keeps track of the navigation history and it is able to take the user to the previous accessed screen). The "Save" button in Physitivity is located in the main area whereas in the *WL++* Physitivity in the header. Essentially, in the first case users need to hit the "Save" button to stop the activity and in the second the "Stop Walking" button. However, by hitting "Save" in both applications, the activity details are stored on the device. Furthermore, both applications store a starting and ending time of the activity. In the first case, this is done in the background, since the time is not shown on the user interface. More specifically, when the “Start” and “Stop” buttons are clicked, a timestamp attribute is attached to the corresponding activity model. In our case, users can see the time on their screen and are able to modify it if this is a past activity. Furthermore, for each activity there are three options of activity intensity (Low, Medium, High for Physitivity and Light, Moderate, Vigorous in the case of *WL++* Physitivity). Finally, the last set of screenshots (Figure 5.2c, 5.2f) represents an instance of the Date/Time picker widget. Users are able to add past entries or edit their existing entries by modifying the starting/ending time of their activities.



(b) Activity: University – Home. Intensity: [✓ Medium]. Tracking : from University to Home

(c) 20/02/2013 09:16 AM — Day 20, Month 02, Year 13; Hours 09, Minutes 16, AM

(e) From Location: University. To Location: Home. From Time: 11/08/2014 8:04 PM. To Time: 11/08/2014 8:24 PM. Light [✓ Moderate] Vigorous

(f) 12/08/2014 11:11 AM — Day 12, Month 08, Year 14, Hours 11, Minutes 11, AM

Figure 5.2: Physitivity (a-c) and *WL++* Physitivity (d-f)

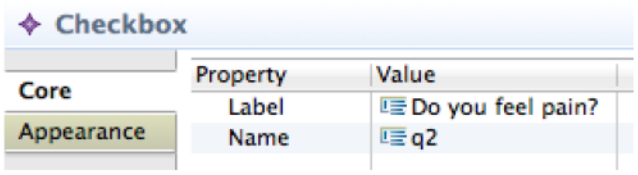
5.2 Hourly Rounds: A Patient's Visiting App Operated By Nurses

The Hourly Rounds is a mobile application intended to be operated by nurses. Nurses working in the Glenrose Rehabilitation Hospital came in contact with the Computing Science Department asking for a mobile application that could assist them during the patients' visiting process. More specifically, the nurse has to check on a patient every hour. If the patient is awake they ask three questions: (a) "Do you need to go to the toilet?" (b) "Do you have pain?" and (c) "Do you need anything?". They can also make a comment of anything interesting about the patient (e.g., in case where the patient is sleeping or has a request). This process is called Hourly Rounding. So far, nurses have been printing out forms with the above questions that they fill out every time they visit a patient. Although filling out the forms does not take a significant amount of time, it has been noticed that several times more than one nurses may visit the same patient simultaneously. Moreover, the collected data is not quite useful since it cannot be processed or analyzed in any way.

The requirements were developed by the nurses requesting the Hourly Rounds application. First, they requested an application with a back-end including nurses and patient hospitalized along with few information such as their room numbers. Essentially, on every shift, the list of the patients' rooms to be visited must be displayed on the nurse's mobile device. As the nurse goes around, she should be able to see the form with the questions to be answered, comment and save the entry. When the entry is successfully saved, she must be able to see the next patient to be visited. On top of that, after an hour of the last visit of a patient, this patient should re-appear on the list. Finally, all the information entered by the nurses must be stored permanently on a single database. Below are listed the steps that describe the development of the Hourly Rounds application by using the *WL++* application specification plugin:

Step 1.

Based on the requirements we got from the nurses, we used the *WL++* framework to generate a prototype of the Hourly Rounds application. The modeling diagram developed for this application is depicted In Figure 5.5. As we can see we have declared two models, (a) Visit and (b) Entry. The **Visit** model represents the patient to be visited and has two attributes: (a) **bed** for the bed number which is a *Numeric* widget and (b) a *Variable* widget namely **checked** which represents the status of the visit; if its value is set to *true* it means that a nurse has already checked on the patient, otherwise the values is set to *false*. The **Entry** model is related to the log of each patient. According to the requirements we have three questions and a commenting section. Therefore, we declared the questions as *Checkbox* widgets since the answer has only two states (yes/no) whereas for the commenting section we used the *Text* widget. Although the checkboxes are named as “q1” (first question), “q2” (second question), “q3” (third question), this property functions as the checkbox’s ID. On the mobile device we have the real questions displayed (the text that has been set through the checkboxes’ label property (Figure 5.3)). Finally, the last attribute of the **Entry** model is a *Clock* widget that timestamps the entry while it is saved. By adding this widget, nurses do not need to manually set the time every time they visit a patient.



Property	Value
Label	Do you feel pain?
Name	q2

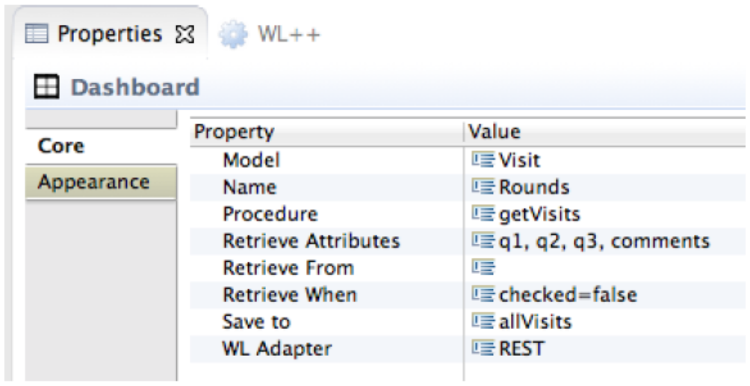
Figure 5.3: Properties of the checkbox widget

Step 2.

Next, we construct the application diagram as follows: first, we add a *Dashboard* view (namely Rounds). Through the Dashboard, nurses can navigate to two available list views (a) Visits and (b) Entries. The *Visits* list is populated with the patients’ to be visited. This list is initialized when the application is connected to the back-end requesting the visits through a RESTful API. As nurses browse through the patients’ list which only includes the number of the patients’ beds, they can click on a certain bed and view the form to be filled. When they hit the “Save” button, they navigate back to the patients’ list as indicated through the “Navigation” link from the *Entry* view to the *Visits* view. The *Entries* view contains all the forms that have been already saved. The only path to view the list of the existing entries is through the Dashboard. Therefore, by navigating back to the Dashboard, the *Entries* view can be easily accessed. Each entry can be edited and re-saved (which results to an update of the information locally and externally).

Step 3.

We also defined an adapter in order to generate a RESTful web service for our application. The need for a back-end is twofold: (a) to be able to retrieve the patients to be visited and display this information to the nurses through their mobile devices and (b) to store the forms that have been filled out on an external database. The name of the adapter and the procedure responsible for the request that needs to be sent and retrieve the visits is indicated on proper views; in our case, this information is included within the *Dashboard* view (Figure 5.4). This means that, when the application is launched, the information retrieved from the back-end is stored on the *Visit Collection* and displayed on the *Visits* list view.



Property	Value
Model	Visit
Name	Rounds
Procedure	getVisits
Retrieve Attributes	q1, q2, q3, comments
Retrieve From	
Retrieve When	checked=false
Save to	allVisits
WL Adapter	REST

Figure 5.4: Properties of the Dashboard view

In Table 5.4, we have listed the requirements and marked the ones that were successfully accomplished by generating the Hourly Rounds application. The mobile application and the back-end were generated by *WL++* . We first manually entered sample data in the database and then run the application in order to test it. As we can see, the visits were displayed in a list view and when clicked they led to a form with three questions and a commenting section (Figure 5.6c). When a form was filled out and saved, we were able to see this entry locally through the *Entries* list

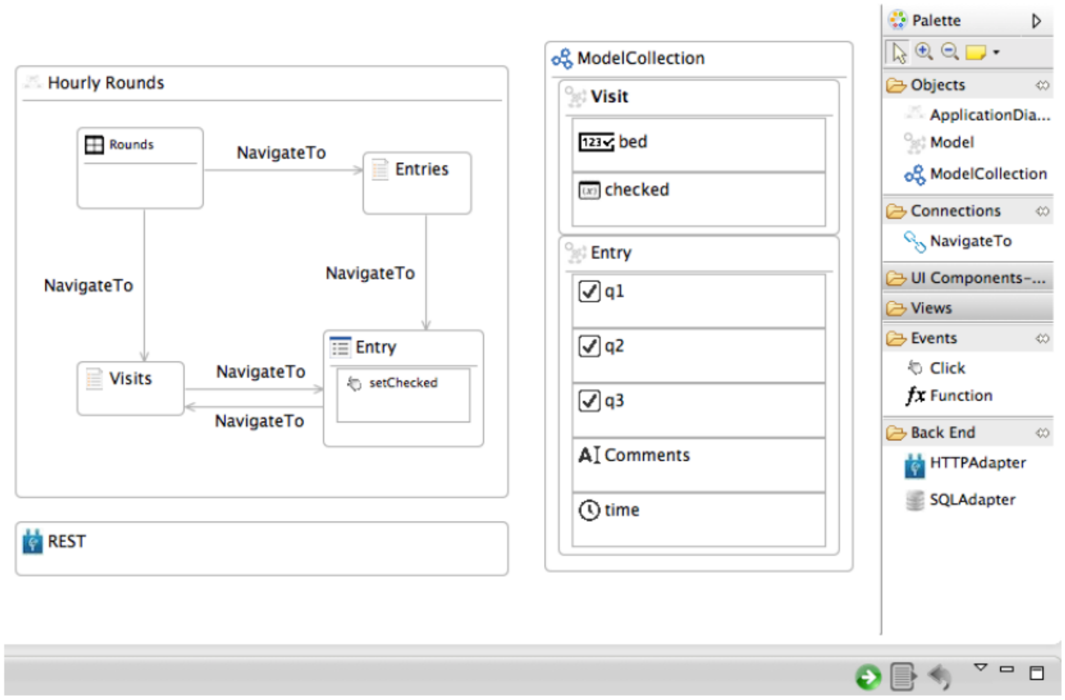


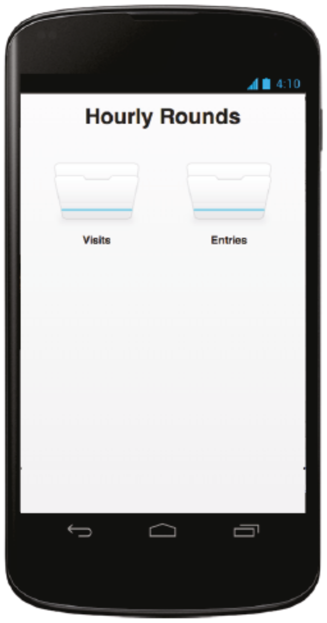
Figure 5.5: Hourly Rounds - Application Digram

view and externally on the Entries table of the database. The *checked* variable were "false" by default but converted to "true" when an entry was saved. Through the list properties, we specified that the entries that should be displayed are the ones for which the *checked* variable is "false". Therefore, the patients that had been visited by a nurse were no longer displayed on the list. The functionality that we had to implement manually was to display on the list the patients that had not been visited for an hour. Since we had the timestamp of the last visit of the patient, it was easy to add an hour and compare it with the device's time. If this difference was equal or greater than an hour, then the *checked* variable's value was becoming "false" again and the list was re-rendered. Nonetheless, this functionality could be also implemented on the back-end side by adding a trigger (special rule) on the database. More specifically, when an entry is saved, then through the trigger we instantiate a new variable with the time that the patient must be visited next and modify the "checked" variable to false. Regarding the user interface, we slightly modified the CSS file in order to change the colours and style of the title and the information displayed on the list. We also replaced the default images with customized ones

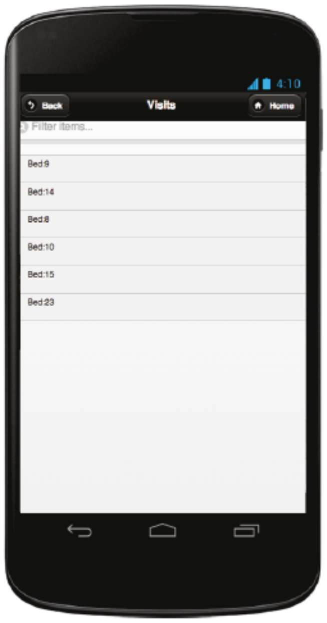
(Figure 5.6e). Finally, during the demo of the application to nurses, they noticed that the answers to the three questions were recorded as “true” and “false” (Figure 5.6d). Therefore, we modified these values to “Yes” and “No” instead (Figure 5.6f). The Hourly Rounds application will be trialed soon by few nurses at the Glenrose Rehabilitation Hospital.

Requirement	**Manual Effort Needed**
Get Visits from a back-end	No
Display visits to the nurse	No
Associate a visit with a patient	No
Display a form for each patient	No
Save the form locally	No
Save the form externally	No
Display a list of all the entries	No
Remove patients from the list when a nurse visits them	No
Add patients that have not been visited for an hour	Yes

Table 5.4: The manual effort needed to accomplish a requirement

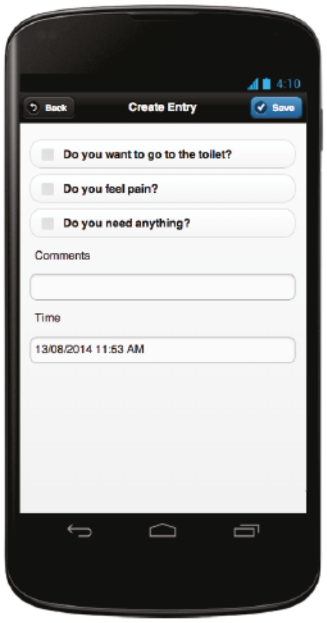


(a)



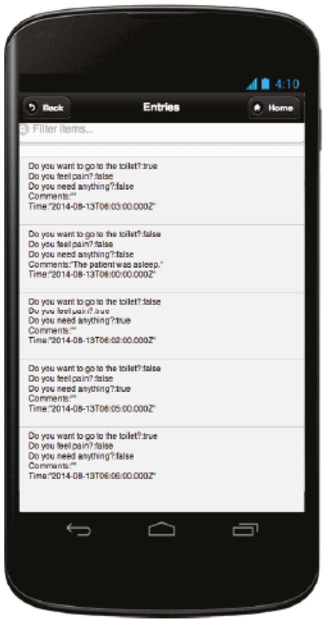
Bed:9
Bed:14
Bed:8
Bed:10
Bed:15
Bed:23

(b)



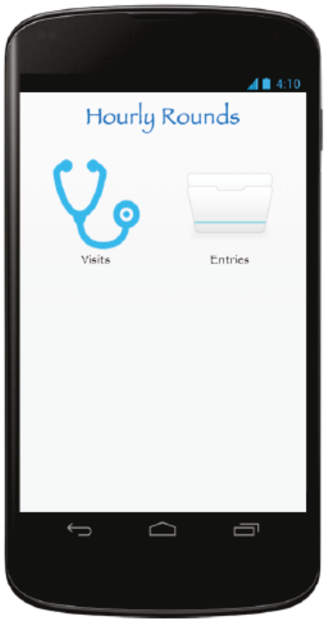
Do you want to go to the toilet?
Do you feel pain?
Do you need anything?
Comments
Time
13/08/2014 11:53 AM

(c)

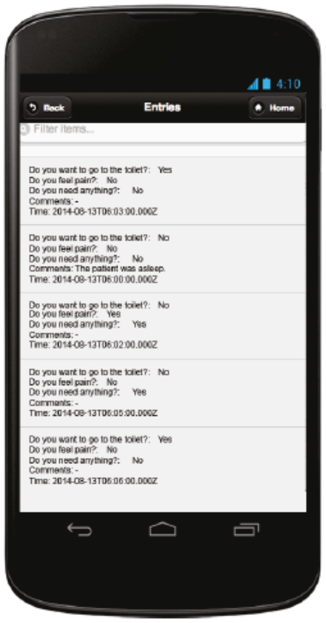


```
Do you want to go to the toilet?:true
Do you feel pain?:false
Do you need anything?:false
Comments:""
Time:"2014-08-13T06:03:00.000Z"

Do you want to go to the toilet?:false
Do you feel pain?:false
Do you need anything?:false
Comments:"The patient was asleep."
Time:"2014-08-13T06:00:00.000Z"

Do you want to go to the toilet?:false
Do you feel pain?:true
Do you need anything?:true
Comments:""
Time:"2014-08-13T06:02:00.000Z"

Do you want to go to the toilet?:false
Do you feel pain?:false
Do you need anything?:true
Comments:""
Time:"2014-08-13T06:05:00.000Z"

Do you want to go to the toilet?:true
Do you feel pain?:false
Do you need anything?:false
Comments:""
Time:"2014-08-13T06:06:00.000Z"
```

(d)



(e)



```
Do you want to go to the toilet?:   Yes
Do you feel pain?:   No
Do you need anything?:   No
Comments: -
Time: 2014-08-13T06:03:00.000Z

Do you want to go to the toilet?:   No
Do you feel pain?:   No
Do you need anything?:   No
Comments: The patient was asleep.
Time: 2014-08-13T06:00:00.000Z

Do you want to go to the toilet?:   No
Do you feel pain?:   Yes
Do you need anything?:   Yes
Comments: -
Time: 2014-08-13T06:02:00.000Z

Do you want to go to the toilet?:   No
Do you feel pain?:   No
Do you need anything?:   Yes
Comments: -
Time: 2014-08-13T06:05:00.000Z

Do you want to go to the toilet?:   Yes
Do you feel pain?:   No
Do you need anything?:   No
Comments: -
Time: 2014-08-13T06:06:00.000Z
```

(f)

Figure 5.6: Hourly Rounds before the manual modification (a-d) and after (e-f)

5.3 Comparison of Applications in Terms of Complexity

By integrating the Backbone library in our *WL++* framework, we achieved to generate similar applications in terms of architecture. Therefore, Backbone offers developers the ability to quickly learn the structure of the applications and therefore results in a more efficient maintenance process. However, as we observed by all the generated applications (Contact Manager in Section 4, Physitivity in Section 5.1 and Hourly Rounds in Section 5.2), they all have different requirements and therefore one may be more complex compared to the others in terms of the number of components it needs to be generated and the plethora of the generated files as well.

In order to compare the applications based on their complexity we constructed Table 5.5, where we can see the number of the different components generated for each one of the applications that have been discussed before. Based on these results, Physitivity seems to be the most complex application that was generated in terms of models, views and templates. For this application in particular, we had to use seven views in total to represent the different activities as well as the lists of the activities. On the other hand, the Contact Manager and Hourly Rounds applications have three and four views respectively. The Contact Manager was used as an example and as such is the least complex application of all composed of a Dashboard, a Form and a List view.

	Models	Views	Routers	Templates	HTML files	Adapters	Time (sec)
Contact Manager	1	3	1	7	1	1	307
Physitivity	5	7	1	12	1	0	955
Hourly Rounds	2	4	1	9	1	1	486

Table 5.5: Complexity comparison among the generated applications

For single page application a good practice is to have the templates in a single HTML file (where the libraries are also imported) and thus, for every application that is generated through the *WL++* plugin, there is a single HTML file along with a single Backbone router component. Finally, the Contact Manager and the

Hourly Rounds applications have supported back-ends and therefore they include a Worklight adapter each compared to the Physitivity which was developed for local use only.

Finally, we measured the time needed to generate the applications using the *WL++* application specification plugin. The time that was required to create the application diagram and specify the properties of each component is 307 seconds, 955 seconds and 486 seconds for the Contact Manager, Physitivity and Hourly Rounds respectively.

Therefore, for a developer who is familiar with the *WL++* plugin and already knows the constraints of the existing metamodel, the time that takes to generate a catalogue-based cross-platform mobile application is only few minutes. The least complex application (Contact Manager) took approximately 3 minutes whereas Physitivity (an application that had a more complex composition) took approximately 16 minutes. Usually, the time that takes to develop an application from scratch may take several weeks or even months. For example, the prototype of the Physitivity application took approximately 2-3 months to be developed. However, using the *WL++* plugin to create the application diagram that represents it, the time to generate about 66% of the application was significantly reduced to only 16 minutes.

On top of that, the files are structured into folders to facilitate the process of deploying the application within a Worklight project. If a developer had to construct from scratch all the application files, not only it would take significantly more time to write the source code but also the first version of the code would be more error prone and therefore, the necessary debugging would also be even more time consuming.

Chapter 6

Conclusions & Future Directions

In this thesis, we presented the *WL++* framework built by integrating three major tools: IBM Worklight, Backbone.js and the Graphical Modeling Framework. The *WL++* application specification plugin leverages model-driven engineering techniques to create cross-platform mobile applications along with supported backends. Before developing our *WL++* plugin, we first defined a general model that describes common features of a subset of applications falling under the catalogue-based category. Based on this model, we developed a set of widgets, through which developers are able to design the proper application diagrams and generate mobile applications.

Backbone.js played an important role as it separates the user interface from the business logic of the generated applications. Our proposed framework also offers flexibility to developers by deciding whether they want their application to be connected to external databases via RESTful APIs or not. RESTful APIs are automatically generated based on the models defined and their attributes. These APIs support basic CRUD/Filtering by specific attributes operations and the definition of the external database structure. Additionally, the reverse engineering component of the *WL++* plugin allows the automatic construction of models based on existing RESTful web services or XSD schemas.

The contribution of this work is threefold:

- **We have designed a modeling language to specify data models and user interface components of catalogue-style applications.** More specifically, the modeling language is composed of components to specify the data models of the application, their attributes through predefined user interface components such as Text, Numeric, Checkbox, Image etc. Moreover, the modeling language includes view components along with events such as the click event along with customized functions. Developers are able to define the navigations among views by using the *Navigate To* connection link. Finally, back-ends can be generated by specifying the proper Worklight Adapters.

- **We have developed a framework to allow developers generating easily and efficiently cross-platform applications.** The *WL++* application specification plugin offers developers to ability to specify the necessary components through drag and drop actions. The available widgets are grouped into clusters to facilitate the process of the application model construction. Two lines of code generation are available: (a) from scratch, and (b) based on existing RESTful APIs and XSD schemas. Through the reverse engineering process, the *Model Collection* component where the models are specified is generated in a semi-automatic way.

- **We have evaluated our framework by using three example applications: (a) a Contact Manager Application, (b) Physitivity: a Physical Activity Tracking Application and, (c) Hourly Rounds: an Application Operated by Nurses while Visiting their Patients.** The Contact Manager application was developed to illustrate the process a developer follows in order to generate an application. A prototype of the Physitivity app was developed from scratch prior to the *WL++* framework development and therefore we attempted to re-generate it. Finally, the Hourly Rounds application was requested by the nurses of the Glenrose Rehabilitation Hospital. As shown by the time measured to generate these application the manual effort needed by a developer in order to generate them from scratch is significantly reduced. The time needed to create the aforementioned applications ranges from approximately 5 minutes to 16 minutes whereas an application such as Physitivity make take up to several months to be developed.

In the future, we aim to add more essential widgets/attributes such as radio buttons and labels. Also we plan to integrate and allow the design of more complicated data model structures (e.g., nested models, inheritance from an abstract model). The design of multiple CSS files would be desired as it allows users to choose the user interface that better fits their needs. Finally, regarding the reverse engineering process, we would like to extend *WL++* in such a way where the integration of multiple RESTful web services would be possible.

Bibliography

[1] Smartphone Ownership 2013. Smartphone ownership 2013. http://www.pewinternet.org/2013/06/05/smartphone-ownership-2013/, 2013. [Online; accessed 8-March-2014].

[2] Fahad Aijaz, M Chaudhary, and Bernhard Walke. Performance comparison of a SOAP and REST mobile web server. *Context*, 2009.

[3] Nick Baetens. Comparing graphical DSL editors: AToM3, GMF, MetaEdit. *University of Antwerp*, 2011.

[4] G Botturi, E Ebeid, Franco Fummi, and Davide Quaglia. Model-driven design for the development of multi-platform smartphone applications. In *Specification & Design Languages (FDL), 2013 Forum on*, pages 1–8. IEEE, 2013.

[5] Coenraets C. The employee directory application. http://coenraets.org/blog/2011/10/sample-app-using-the-phonegap-database-api/, 2011. [Online; accessed 9-March-2014].

[6] Andre Charland and Brian Leroux. Mobile application development: Web vs. Native. *Communications of the ACM*, 54(5):49–53, 2011.

[7] Jason H Christensen. Using RESTful web-services and cloud computing to create next generation mobile applications. In *Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*, pages 627–634. ACM, 2009.

[8] Alberto Rodrigues Da Silva, João Saraiva, Rui Silva, and Carlos Martins. XIS-UML profile for extreme modeling interactive systems. In *Model-Based Methodologies for Pervasive and Embedded Software, 2007. MOMPES'07. Fourth International Workshop on*, pages 55–66. IEEE, 2007.

[9] Paul E Dickson. Cabana: a cross-platform mobile development system. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pages 529–534. ACM, 2012.

[10] Roy Thomas Fielding. *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine, 2000.

[11] Martin Fowler. Domain Specific Languages. http://martinfowler.com/bliki/DomainSpecificLanguage.html, 2004. [Online; accessed 17-September-2014].

[12] Martin Fowler. Flow synchronization. http://martinfowler.com/eaaDev/FlowSynchronization.html, 2004. [Online; accessed 16-September-2014].

[13] Martin Fowler. Observer synchronization. http://martinfowler.com/eaaDev/MediatedSynchronization.html, 2004. [Online; accessed 16-September-2014].

[14] Anar Gasimov, Chuan-Hoo Tan, Chee Wei Phang, and Juliana Sutanto. Visiting mobile application development: what, how and where. In *Mobile Business and 2010 Ninth Global Mobility Roundtable (ICMB-GMR), 2010 Ninth International Conference on*, pages 74–81. IEEE, 2010.

[15] Nathan Hazout. Handling backend responses in adapters. https://www.ibm.com/developerworks/community/blogs/worklight/entry/handling_backend_responses_in_adapters?lang=en, 2014. [Online; accessed 27-July-2014].

[16] Henning Heitkötter, Tim A Majchrzak, and Herbert Kuchen. Cross-platform model-driven development of mobile applications with MD2. In *Proceedings*

of the 28th Annual ACM Symposium on Applied Computing, pages 526–533. ACM, 2013.

[17] Adrian Holzer and Jan Ondrus. Trends in mobile application development. In *Mobile Wireless Middleware, Operating Systems, and Applications-Workshops*, pages 55–64. Springer, 2009.

[18] Hagen Höpfner, Jonas Pencke, David Wiesner, and Maximilian Schirmer. Towards a target platform independent specification and generation of information system apps. *ACM SIGSOFT Software Engineering Notes*, 36(4):1–5, 2011.

[19] Cowart J. Pros and cons of the top 5 cross-platform tools. http://www.developereconomics.com/pros-cons-top-5-cross-platform-tools/, 2013. [Online; accessed 9-March-2014].

[20] Stuart Kent. Model-Driven engineering. In *Integrated formal methods*, pages 286–298. Springer, 2002.

[21] Glenn E Krasner, Stephen T Pope, et al. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of object oriented programming*, 1(3):26–49, 1988.

[22] OpenDJ Contact Manager. OpenDJ Contact Manager. https://svn.forgerock.org/commons/mobile/contact-manager/trunk, 2013. [Online; accessed 9-March-2014].

[23] Craig McKeachie. Choosing a JavaScript MVC framework. http://www.funnyant.com/choosing-javascript-mvc-framework/, 2013. [Online; accessed 23-June-2014].

[24] Igor Minar. -. https://plus.google.com/+IgorMinar/posts/DRUAkZmXjNV, 2010. [Online; accessed 19-September-2014].

[25] Mobile-Spot-D-Project. Mobile-Spot-D-Project. https://github.com/DomenicoColandrea86/mobile-spot-d-project, 2013. [Online; accessed 9-March-2014].

[26] Gavin Mulligan and Denis Gracanin. A comparison of SOAP and REST implementations of a service based interaction independence middleware framework. In *Simulation Conference (WSC), Proceedings of the 2009 Winter*, pages 1423–1432. IEEE, 2009.

[27] Graphical Modeling Project. -. http://www.eclipse.org/modeling/gmp/, -. [Online; accessed 21-September-2014].

[28] Mitch Pronschinske. Cage match! Sencha Touch vs. jQuery Mobile. http://css.dzone.com/articles/sencha-touch-v-jquery-mobile, 2012. [Online; accessed 27-June-2014].

[29] André Ribeiro and AR Silva. XIS-Mobile: A DSL for mobile applications. In *Proceedings of SAC 2014 Conference, ACM*, 2014.

[30] Douglas C. Schmidt. Guest editor's introduction: Model-Driven Engineering. *Computer*, 39(2):25–31, 2006.

[31] Mathias Schäfer. JavaScript application architecture with Backbone.js. http://www.slideshare.net/molily/javascript-application-architecture-with-backbonejs, 2012. [Online; accessed 2-August-2014].

[32] Mobile Technology Fact Sheet. Mobile technology fact sheet. http://www.pewinternet.org/2013/06/05/smartphone-ownership-2013/, 2013. [Online; accessed 8-March-2014].

[33] P Smutny. Mobile development tools and cross-platform solutions. In *Carpathian Control Conference (ICCC), 2012 13th International*, pages 653–656. IEEE, 2012.

[34] Thomas Steiner. Automatic multi language program library generation for REST APIs. http://www.lsi.upc.edu/~tsteiner/papers/2007/automatic-multi-language-program-library-generation-for/rest-apis-masters-thesis-2007.pdf, 2007. [Online; accessed 6-August-2014].

[35] Eleni Stroulia, Shayna Fairbairn, Blerina Bazelli, Dylan Gibbs, Robert Lederer, Robert Faulkner, Janet Ferguson-Roberts, and Brad Mullen. Smartphone application design for lasting behavioral changes. In *Computer-Based Medical Systems (CBMS), 2013 IEEE 26th International Symposium on*, pages 291–296. IEEE, 2013.

[36] Eleni Stroulia, Dylan Gibbs, and Blerina Bazelli. Towards families of personalized mobile applications. In *Services (SERVICES), 203 IEEE Ninth World Congress on*, pages 166–169. IEEE, 2013.

[37] Arie Van Deursen, Paul Klint, and Joost Visser. Domain-specific languages. *Centrum voor Wiskunde en Informatika*, 5, 2000.

[38] Ivan Vasiljević, Gordana Milosavljević, Igor Dejanović, and Milorad Filipović. Comparison of graphical dsl. *University of Novi Sad*, 2013.

[39] W3.org. Web SQL database. http://www.w3.org/TR/webdatabase/, 2010. [Online; accessed 17-September-2014].

[40] Thomas Weigert and Frank Weil. Practical experiences in using model-driven engineering to develop trustworthy computing systems. In *Sensor Networks, Ubiquitous, and Trustworthy Computing, 2006. IEEE International Conference on*, volume 1, pages 8–pp. IEEE, 2006.