Politecnico di Milano

Scuola di Ingegneria Industriale e dell'Informazione

Master of Science
Engineering of Computer Systems

Model Driven Development of iOS Apps with IFML

By: Félix Javier Acero Salazar
Student ID: 795545

Supervisor: Prof. Marco Brambilla

December 2014

To my mum, my cousin and my girlfriend.
For their unconditional support and honest advice

Abstract

Multi-platform mobile development has never been as relevant as it is today. Every year, markets are flood with new and more powerful mobile devices, that pack several sensors, allow for richer user interactions, and run one of several operating systems. This scenario has put software developers in a troublesome situation, in which several versions of the same application — one for each major mobile platform – has to be developed, tested and supported.

To partially address this problem, an entire ecosystem of multi-platform development tools has appeared, promising to help developers reduce the overhead caused by the implementation of a cross-platform application. Even though these solutions operate under different assumptions and use different levels of abstraction, they ultimately require developers to program using an intermediate programming language, which is later transformed into its native equivalent, or run within a constraint environment, like a web browser.

Developers, however, have fewer options in the modeling world.

With the recent adoption by the Object Management Group (OMG) of a new standard [21] an opportunity for a Model Driven solution has arisen. The standard introduces the Interaction Flow Modeling Language or IFML, which can be used in tandem with UML, to describe the fundamental aspects of modern mobile, web and desktop applications. In this way, a set of code generators — one for each major mobile platform – could be developed, to transform the IFML and UML models of an application, into their native equivalents. A solution like this will benefit from the advantages offered by a Model Driven approach, allowing developers to focus in the design of the differential aspects of their applications, instead of having to worry about the implementation details.

To approach this Model Driven solution, our project focuses in the development of iOS applications with IFML, through the development of a code generation tool. Additionally, building on the experience and knowledge gathered in the generation process, we propose a set of extensions for the IFML metamodel, that aim to broaden the scope and modeling capabilities of the language.

Sommario

Lo sviluppo delle applicazioni mobili multi-piattaforma non aveva avuto la rilevanza che ha nella attualità. Oggi anno il mercato offre nuovi e più potenti dispositivi mobili con un numero importante di sensori che arricchiscono l'interazione con l'usuario e che si eseguono in un ampio numero di sistemi operativi. Questo scenario a lasciato gli sviluppatori in una situazione un poco problematica, richiedendo un esteso numero di versioni di una stessa applicazione – una per oggi principale piattaforma mobile.

Per affrontare parzialmente questa problematica e aiutare gli sviluppatori, è apparso un intero ecosistema di strumenti, che promettono ridurre il lavoro addizionale nella implementazione di una applicazione per diverse piattaforme. Queste soluzioni vengono svolte sotto diverse assunzioni e usano diversi livelli di astrazione. Nonostante, lo sviluppo richiede in definitiva l'utilizzo di un linguaggio intermedio che successivamente sarà trasformato nel linguaggio nativo equivalente o eseguito in un ambiente vincolato come un web browser.

Gli sviluppatori, tuttavia, hanno altre opzioni dentro il mondo della modellazione.

Con la recente adozione di un nuovo standard [21], da parte del Object Management Group (OMG), è sorta una opportunità per le soluzioni Model Driven. Questo standard introduce il Interaction Flow Modeling Language o IFML che può essere utilizzato di pari passo con UML per descrivere gli aspetti fondamentali delle applicazioni mobili. In questo modo una serie di generatori di codice – uno per oggi grande piattaforma mobile – potrebbe essere sviluppato per trasformare i modelli IFML e UML di una applicazione, nei loro codice nativo equivalente. Una soluzione come questa si beneficia dei vantaggi offerti da un approccio Model Driven, permettendo agli sviluppatori di concentrarsi nella progettazione degli aspetti delle loro applicazioni, invece di doversi preoccupare dei dettagli d'implementazione.

Per affrontare la soluzione Model Driven, il nostro progetto si concentra nella realizzazione d'applicazioni per iOS con IFML, attraverso lo sviluppo di uno strumento per la generazione di codice. Inoltre, basandoci sull'esperienza e le conoscenze riuniti nel processo di generazione, proponiamo una serie di estensioni per il metamodello di IFML, che mirano ad ampliare gli scopi e la capacità di modellazione del linguaggio.

Contents

List of Figures

Chapter 1

Introduction

1.1 Context

Developing multi-platform mobile applications has never been more relevant than it is today. Every year brings a new wave of technologies and devices, each more powerful that its predecessor, including more and smarter sensors and allowing for richer user interactions. These innovation cycles, have slowly flood the market with all kinds of devices, and has open the doors for a fragmented user population that interacts with applications using a wide variety of smart phones, tablets and other handheld devices. To complicate the matter even more, each of these devices includes a different set of capabilities, runs one of several operating systems, and comes in different form factors.

On the other side of this picture lie the developers, who need to offer their applications in the different market places, and carefully tweak them to guarantee that users have the same experience across their devices. All of this, without losing performance and preserving a premium visual design. In consequence, where before there was one application that had to be developed, maintained, improved and supported, developers now have several applications that frequently only share their name, their visual design and the fact that they have been created by the same company.

As a result, an entire ecosystem of muti-platform development tools has emerged. Project like PhoneGap [1] and Apache Cordova[2] encourage devel-

[1] http://phonegap.com/
[2] http://cordova.apache.org/

opers to program cross-platform applications through the usage of already familiar web technologies like HTML, Javascript and CSS. Other solutions like Titanium[3] and Xamarin [4], advocate the usage of a cross-platform tool, that allows the creation of native applications.

Developers however, have fewer alternatives to address the same issue at a modeling level.

1.2 Motivation

For years, software developers have used modeling tools to design their applications at a platform independent level. The Unified Modeling Language or UML [24], is one of these tools. However, as powerful and expressive as UML may be, there are still several aspects involved in the development of modern mobile, web and desktop applications that are not covered by it.

An important omission in UML, corresponds to the specification of the front-end of an application.

To bridge this gap, the Object Management Group has adopted a new standard, the Interaction Flow Modeling Language or IFML [21]. This modeling language can be used to describe the principal dimensions of an application's front-end. Moreover, since the entities and classes in the modeling language are based on a formal metamodel, code generators and modeling extensions can be proposed.

As a consequence, a Model Driven Development (MDD) approach for developing cross-platform mobile applications based on IFML could be proposed. In this approach, IFML could be used to describe and specify the front end of an application. In turn, UML could be used to describe the nuances of the business logic. And finally, a set of code generators could be used to translate the elements in the IFML and the UML models, into the equivalent native code of the major mobile platforms — Android, iOS and Windows Phone.

In contrast with other cross platform development solutions, like PhoneGap, Apache Cordova, Titanium and Xamarin, this MDD approach presents two advantages. First it would operate at a higher level of abstraction, allowing developers to focus in the differential aspects of their applications instead of

[3] http://www.appcelerator.com/titanium/
[4] http://xamarin.com/

worry about the implementation details. Second, it would produce, through the code generators, native code that could be later enhanced and optimized to obtain the best performance.

1.3 Objectives

The Model Driven Development approach of our project was based on the Interaction Flow Modeling Language (IFML) [21], and focused on iOS as the main target platform.

On one hand, we chose IFML, because with it, software developers can model the front end of their applications, at a platform independent level. Furthermore, since it integrates very well with other modeling languages, choosing IFML also meant that designers could use a language like UML to describe the nuances of the business logic of their applications.

On the other hand, we chose to focus on iOS since it remains as one of the more popular mobile operating systems in the market. In fact, according to a recent study conducted by the International Data Corporation [23], iOS is the second most popular mobile operating system with a share of 11.7% of all mobile devices shipped in 2014 — surpassed only by Android, which remains as the most popular operating system with a market share of 84.7%.

Particularly, in our project we focused in two objectives. First, to enhance the portability of the modeling language by developing a code generator for iOS. And second, to broaden the modeling capabilities of IFML by adding a set of mobile specific extensions to its metamodel.

1.4 Structure

This document is organized in six different chapters.

In Chapter 1 we introduce some of the challenges posed by multi-platform mobile development, and the different solutions that have been proposed in the industry to face them. Then, we motivate the opportunity for a model driven approach based on IFML and UML. Finally, we highlight the main objectives of the project.

In Chapter 2 we cover all the background information needed to grasp the

basic concepts of IFML and iOS. In the case of IFML, we introduce the core entities of the modeling language and provide several examples based on a simple dictionary app. For iOS instead, we briefly cover the main characteristics of the platform, the architecture of a typical iOS application and other traits that distinguish iOS from other mobile operating systems.

In Chapter 3, we provide an overview of the process followed during the project. We start by discussing the strategy we used to identify potential extensions for IFML. Then, we introduce the the seven milestones that lead to the creation of the code generators , namely: design and implementation of the prototype app, definition of the target metamodel, creation of the mapping rules, implementation of the static library, development of the code generators, creation of the integration tools and packaging. Besides the methodology we present, at the end of the chapter, the tools used to develop the deliverables of the project.

In chapter 4, we discuss at length some of the extensions that we proposed to broaden the modeling capabilities of IFML. However, a complete description of each of these extensions can be found in the appendices of this document.

In Chapter 5, we give a deeper look to the intermediate results that we achieved while working on the code generators. In this way, the "Prototype App" section starts describing the application that we used to guide the generation process. The "Target Architecture" section, introduces the architecture of the generated apps from a software driven perspective and a model driven point of view. In the "Mapping" section, we use some examples to illustrate the rules that map an IFML model into our target metamodel. Then we present the contents of the "Static Library", and discuss how they contribute to the shape of our solution. The "Code Generation" section instead, reviews the generation templates and provides a glimpse of the final project structure and the shape of generated Objective-C code. In the "Integration" section, we discuss the motivations that lead to the development of such tools and provide an overview of their functionality. The "Packaging" section describes the strategy we used to make the project outputs more accessible to potential users.

In Chapter 6, we present the results of the project, highlighting the different tools that were developed as well the proposed extensions. Then we discuss the project shortcomings, and the opportunities that should be explored in future research efforts.

Chapter 2

Background

2.1 IFML

2.1.1 Basic Concepts

IFML is a modeling language designed to tackle the main challenges of the front end specification of an app, through a small set of extensible entities based on a formal meta-model that provides them with rich semantics.

At its core, IFML is meant to be simple and flexible, but perhaps more importantly IFML is meant to be abstract. This allows the possibility of defining the main traits of an application's front end making as few visual commitments as possible. In contrast to other approaches like, wireframes or lo-fi mockups, IFML has the advantage of focusing on the principal aspects of the front end without dealing with their concrete visual implementation, which may vary radically from platform to platform. Similarly, IFML benefits from its formality by allowing the implementation of code generators – which could reduce the development time by automatically producing a great percentage of the code needed to implement an application.

Put in different words, IFML can be described as a language that formalizes the front end of modern web, mobile and desktop applications, providing an appropriate separation between the front end requirements and their concrete visual implementation.

Another important trait of IFML is its extensibility, which allows the language to remain small and general while permitting modelers and designers

to specialize certain components in order to enrich the semantics of their models and make the diagrams more readable.

As for the main motivations of IFML, the scenario is clear. The past decade has seen a huge change in the application development scene. While some years ago, applications were mostly being developed for a specific platform and run on devices that allowed for a small set of interactions, current applications need to target several platforms and offer support for a wide range of devices that allow users interact in many different ways. Moreover, the new focus on general consumer markets, brought by the appearance of centralized marketplaces, has made usability and user interface design primary concerns and a central part of the development life-cycle of an application.

More formally, IFML is a modeling language akin to UML, which has accompanied for years the community of software development. Like UML, IFML aims to define and model certain aspects of an application at an abstract level based on a formal meta-model that provides each of the entities of the language with clear semantics. In contrast, while UML offers a family of diagrams, each of which is meant to capture a particular aspect of an application, IFML focuses on the front end and does so by using a single type of diagram.

As stated previously, IFML's main objective is to formalize and specify the different dimensions that make up the front end of an application. In consequence, a good way to get acquainted with the main entities of the language is by dividing them according to the particular aspect of the front end that they intent to model, namely: View Composition, Navigation, Content, Interaction and Context.

With the aim of showing how each of the main entities of IFML may be used in practice, throughout this section we will model a dictionary application. The main requirements of the application are:

- The application must allow the user to enter a word to see its definition.
- The application must allow the user to see a list with the words the he has previously searched for.
- The application must allow a user to bookmark a word for later reference.
- The application must allow a user to see a list with the words he has bookmarked.

2.1.2 View Composition

A good place to start modeling this app is the view hierarchy. This hierarchy, identifies the main sections and top level containers, providing some hint regarding their visibility and rechability. According to the outlined requirements, the dictionary app has three main sections: Search, Recent and Bookmarks, as well as an independent section dedicated to show the definition of a word. To model this basic structure in IFML, we will need to use View Containers.

View Containers

View Containers are the most fundamental visual units of IFML. They are the foundation of the application's view hierarchy and can accommodate both View Containers and View Components [21]. As shown in Figure 2.1, we can use 6 View Containers to model the View Composition of the application. The first container (Dictionary App) acts as the root of the view hierarchy and will accommodate and manage all the other containers of the app. Under it, we find two View Containers, Main and Word.

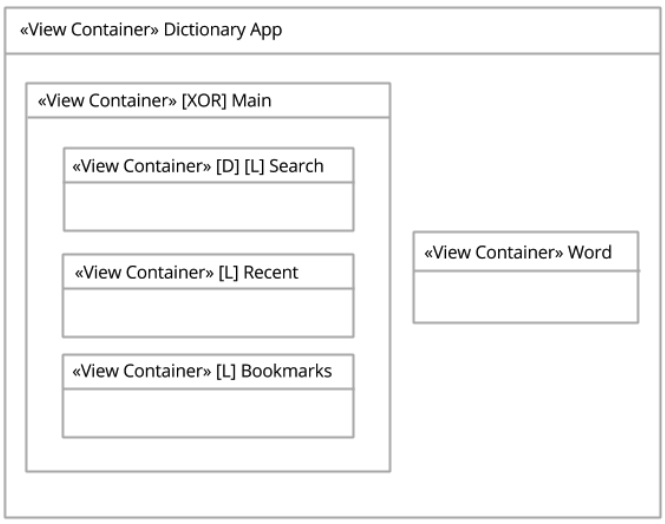

Figure 2.1: IFML Review - View Containers

While the Word View Container is rather simple, the Main View Container manages a set of mutually exclusive containers, namely Search, Recent and Bookmarks. To model this behavior, each of them has several tagged values that indicate their visibility and rechability. In this way, the Main View

Container is tagged as [XOR], to express the mutually exclusive behavior of its direct children, while the Search, Recent and Bookmarks containers are tagged with an [L] – landmark, to imply sibling navigation. Finally the Search View Container is tagged with a [D] – default, indicating that it should be the default section shown to the user.

2.1.3 Content

The next step will be to determine the information that needs to be shown in each of the previously identified containers, as well as the data that needs to be captured from the user.

Let's start with Search. This section, should prompt a search bar in which a user may enter a word in order to get its definition. To model this behavior we will need to make use of two of IFML's platform independent extensions: Form and Simple Field.

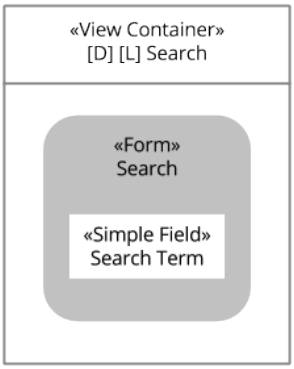

Figure 2.2: IFML Review - Search

Forms

Forms in IFML are an extension of a much more generic entity called View Component. As expected, they are used to model the capture of information from the user through a set of fields [21]. In the Search section, we will need a form entity with one single field in which the user may introduce the word she wants the meaning of.

Fields

As stated previously, Fields are each of the elements that make up a Form. They could be of two types Simple Field or Selection Field, depending on whether the user needs to introduce the data herself or if instead she can choose from a list of options [21]. The search bar of our application needs to be modeled with a Simple Field, since the user must introduce the word herself.

Now, lets analyze the Recent and Bookmarks sections (Figure 2.3). Both of them need to show a list of words accompanied of a snippet of their definition and an indication of whether they have been bookmarked. To model this content, we will make use of two core entities of IFML, namely: Data Binding and Visualization Attribute; and one entity that belongs to the platform independent extension of the language: List.

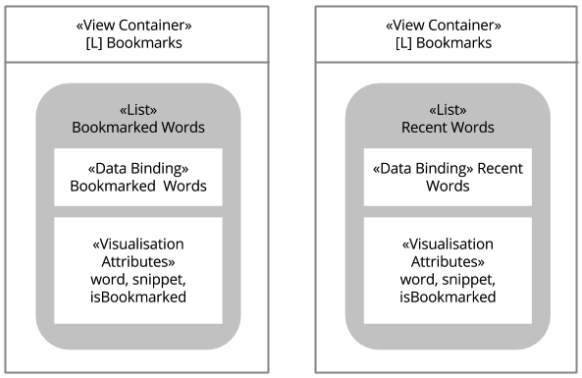

Figure 2.3: IFML Review - Bookmarked and Recent Words

Data Bindings

DataBindings are used to express the relations between the view and model layers of a MVC architecture. That is, a Data Binding should be used to imply that a particular View Component publishes the data of a given Domain Object [21]. In our case, the Recent Section shows words that belong to the Recent Words collection, while the Bookmarks Section shows words that belong to the Bookmarked Words collection. The previous distinction regarding the domain objects, is obviously more clear under the light of the application's domain model, which is not included in the current document for simplicity. However, in most of the examples available in the literature

a detailed Domain Model is provided and later referenced within the IFML models.

Visualization Attributes

Visualization Attributes like Data Bindings, are a specialization of a fundamental entity of IFML called View Component Part, and are meant to further specify the behavior of the View Components [21]. While a Data Binding may specify which particular Domain Object needs to be presented by a View Component, the Visualization Attributes instead, allow the software designer to specify explicitly which attributes and values need to be disclosed. In case of the dictionary app, both Recent and Bookmarked Words need to show the word along with a snippet of its definition. Similarly, an indication of whether the word has been bookmarked is needed.

List

The last entity depicted in Figure 2.3 is List. Lists are entities that extend the behavior of a View Component, and model a component able to display a collection of objects retrieved through a Data Binding [21]. Both, Recent and Bookmark View Containers include a List View Component to show the recently searched words and the bookmarked words respectively.

Finally, we describe how to model the contents of the Word View Container. This section of the app shows a word along with its definition, and appears every time a user types in a word in the search bar or selects one from the recent or bookmarked words list. To model this content, we use yet another platform independent extension of IFML: Details

Details

Details are also a type of View Component, that are used to model the publication of attributes from their bound Domain Object, as determined by a Data Binding entity [21]. In the dictionary app, we use a Details entity to model a component that shows a word along with its definition as shown in the Figure 2.4.

To finish this section, we now define the two generic classes from which most

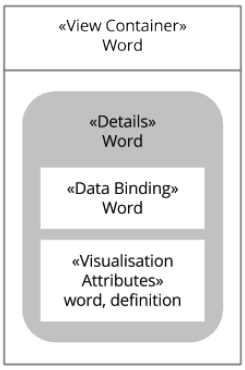

Figure 2.4: IFML Review - Details

of the previously defined entities extend from: View Components and View Component Parts.

View Components

View Components represent a visual unit that: may publish the attributes of a domain object; capture information from the user; or react to certain user interactions. In general, View Components define what is displayed in the view hierarchy defined by the View Containers [21]. To put it within the context of the dictionary app, this entity is the one from which List, Form and Details extend and get most of their semantics from.

View Component Parts

On the other hand, View Component Parts are each of the sub-elements that make up a View Component [21]. In the previously discussed models we presented the most frequently used View Component Parts: Data Binding, Visualization Attributes and Simple Fields.

2.1.4 Navigation

So far we have modeled, the view hierarchy and the content of each of the sections of the app. However, we still need to the describe the way a user may move around these sections and what are the dependencies, if any, between them.

To model these relationships IFML provides entities like Navigation Flows and Parameter Bindings that allow a software designer to model navigation and parametrized communications between the View Elements of an app.

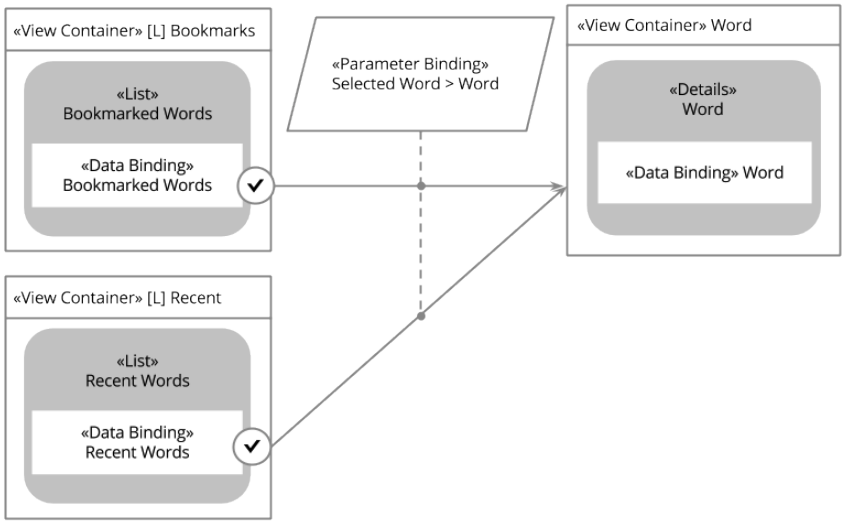

Figure 2.5: IFML Review - Navigation Flow

Navigation Flows

Navigation Flows are a special type of a more generic class called Interaction Flow, that model communications between a source and a target Interaction Flow Elements. Despite their broad nature, Navigation Flows are frequently used to model a visual transition from a source View Element to a target View Element, making them the most suitable construct for expressing navigation in IFML [21].

While the previously discussed Main View Container already describes the navigation dynamics that should govern the Search, Recent and Bookmarks View Containers, there is still one section that cannot be reached: Words.

According to our list of requirements, this section needs to be shown every time a user enters a word in the search bar, or every time a user selects a word from either the Recent Words or the Bookmarked Words list.

To make the Word Section reachable, we shall create three Navigation Flows starting from the Search, Recent and Bookmarks sections respectively, and targeting the Word View Container. For now, let's assume that a user has a way of triggering these transitions, and we will later dive deeper on the

specific entities that can be used to describe such behavior.

Finally, we need to formalize the previously defined navigation flows by means of parameters that will tell the Word View Container which word was typed in or selected, to show the appropriate definition. To achieve this, we need to make use of Parameter Bindings, as shown in Figure 2.5

Parameter Bindings

Parameter Bindings express the input / output dependencies that exist between the source and target of an Interaction Flow. That is, they stipulate how the outputs of the source entity, map into the inputs of the target one [21]. As shown in the Figure 2.5, we will need to stipulate a parameter binding for each of the navigation flows, providing the appropriate mapping in every case.

Before finalizing this section, we introduce the concept of Interaction Flows, since they are a generalization of the aforementioned Navigation Flows and constitute a fundamental part of IFML.

Interaction Flows

Interaction Flows represent the communication between a couple of View Elements or between a View Element and an Action [21]. Such communications are usually triggered by the occurrence of an event, and may imply a change of state in the user interface. In any case, an Interaction Flow is typically associated with a Parameter Binding to specify the dependencies between the the source and target entities. Finally, an Interaction Flow can be either a Navigation Flow, implying parameter passing and change of focus, or a Data Flow, implying parameter passing only.

2.1.5 Interaction

It is now time to present the possibilities that IFML offers to model the different interactions that a user may have with the front end of an application.

In the case of the dictionary a user may interact with the application in three different ways: introduce a string to perform a search; select a word

out of the Recent or Bookmarked word lists; and bookmark a particular word. Even though all of these interactions imply actions performed by the user, they also involve different reactions that are usually described by the application logic.

Using IFML a software designer is able to: model the specific interactions that must performed by a user; identify the Entity that is in charge of monitor the occurrence of such interactions; and determine the task that the application should perform in response to them. Specifically, IFML defines two entities that model the user's and the application's side of an interaction: Events and Actions.

Events

Events in IFML represent an occurrence that affect the state of the application causing parameter passing between entities and, in some, cases change of focus [21]. There are several kinds of events defined in IFML, like View Element Events, System Events and Action Events. Particularly, View Element Events are an abstraction for an action that is performed by the user — e.g click, submit, select – and is monitored by the event's target, which in case of the View Element Events happens to be a View Element.

In Figure 2.6, there is a simplified version of some of the containers of the dictionary app along with an indication of the events that the user can trigger in each of them. In this way, in the Bookmarked and Recent sections a user should be able to select one of the words to see its definition; in the Search form, a user should be able to introduce a string and start a search operation; and in the Word View Component the user should be able to update the bookmarked status of a word.

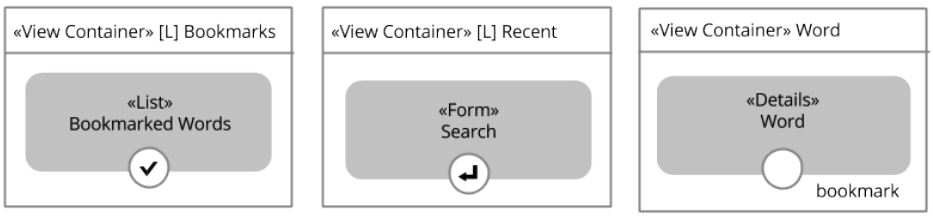

Figure 2.6: IFML Review - Events

In Figure 2.6, all the events are represented using a white circle, but some of them are further defined with a slightly different notation. Such is the case of the Selection and the Submit events. Both of them are part of the

platform independent extensions of IFML, and aim to refine the semantics of the language by providing a better description of the kind of interaction that is expected from the user. In this way, the Select Event, associated with the Bookmarked Words, implies that the user needs to select one of the items of the list; and the Submit Event, indicates that the user must introduce and submit some data.

As stated previously, there are always two sides in an interaction: one of them is the action performed by the user — which in IFML is modeled through Events —, and the other is the task that is performed by the system in response to it. To model the latter, IFML uses Actions.

Actions

Actions in IFML represent pieces of business logic that are triggered by the occurrence of an event [21]. In the case of the dictionary app, submitting a word in the search bar causes the system to trigger a look up task, that retrieves the definition of the word and navigates the user to the Word View Container once completed. Similarly, in the Word Details View Component, the bookmark event causes the execution of a piece business logic that updates the bookmark status of the word.

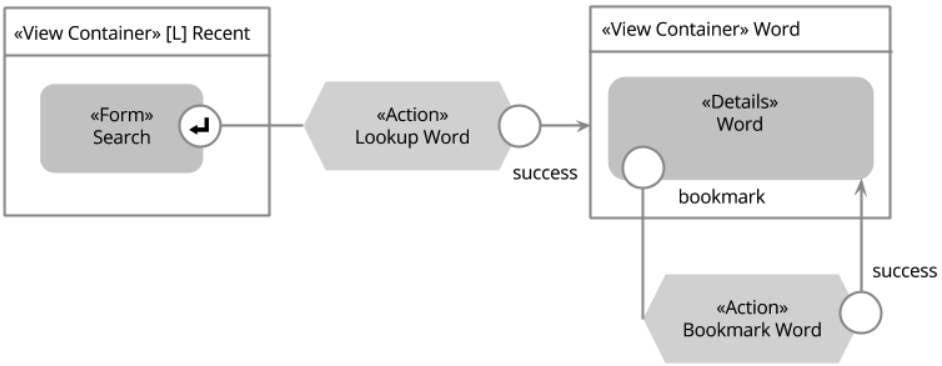

Figure 2.7: IFML Review - Actions

2.1.6 Context

The Context defines the particular View Composition, Content, Navigation and Interactions that are available to a user using a particular device. In IFML, these four dimensions of the front end make up an entity called View

Point.

Following the previous definition, IFML allows a software designer to define several Viewpoints for his application, identifying for each of them the particular context conditions needed to activate it. Such Context is modeled in terms of Context Dimensions that evaluate variables related with the User, the Position and the Device.

To show how the Context information may alter the actions available to a user in an application, let's suppose that in the dictionary app only the premium users are able to bookmark words. This additional requirement, will imply a runtime check in order to figure out whether the active user is allowed to update the bookmark status of a word. To model this feature, and express other traits of the Context, IFML has entities like the previously defined Context and Context Dimension. On the other hand the run-time evaluations are modeled through Activation Expressions.

Activation Expressions

Activation Expressions are used in IFML to determine whether a particular element of the interface should be active or not. These entities come handy in cases where run-time checks need to be applied to figure out if a particular action or content section is available to the user. Such is the case of the dictionary app, in which we can use an Activation Expression to determine if the user is a premium user and if so, enable the bookmark interaction.

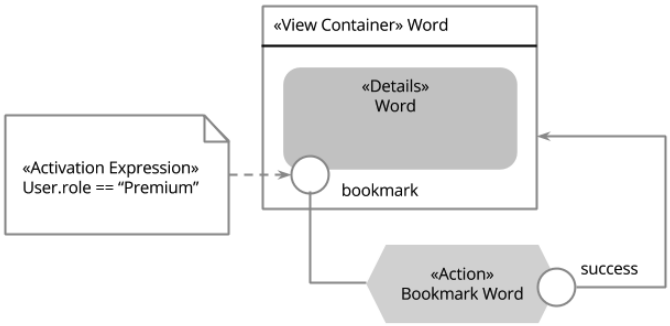

Figure 2.8: IFML Review - Activation Expressions

2.2 iOS

2.2.1 Technologies

One of the reasons why programming for iOS is regarded as a daunting task, is because there are several concepts and abstractions that a programmer needs to understand, before starting to program applications. A good starting point for this journey, is the study of the very technologies that power the platform, as well as the services that they provide to the developers.

The architecture of the platform can be seen as a four layered system, organized in ascending level of abstraction. At the lowermost level we find Core-OS [8] that provides several APIs to: communicate with external devices, manage bluetooth communications, manage security, provide managed file-system access, support concurrency and threads, memory allocation and management, among others.

Following the stack, we encounter the Core Services [8] layer that encompasses APIs for networking operations, model management and arguably the most important framework of the platform: Foundation. This framework, contains Objective-C wrappers for basic classes like: NSObject, NSDate, NSData, NSString, NSInteger, NSNumber as well as collections like NSArray, NSDictionary and NSSet.

At the third level we find the Media Layer [8], that basically contains frameworks for image, video and audio management. Finally at the top of the stack, we find Cocoa Touch [8]. This framework promotes the Model-View-Controller (MVC) architecture, and provides the foundation for most iOS apps. In consequence, Cocoa Touch contains the family of classes that every iOS programmer needs to be familiar with.

2.2.2 Objective-C

Another defining characteristic of the platform is the programming language that powers it: Objective-C. The language is currently supported and continuously enhanced by Apple and is used for natively developing apps for the OSX and iOS operating systems. At a general level, Objective-C is an object oriented superset of C [9] and inherits many of its traits, like the support of two files types (≪ .m ≫ or implementation files and ≪ .h ≫ or header files), or the required tasks expected from the programmer to manage memory allocation and deallocation of objects and data structures –although recent

enhancements on the compiler, have moved some of this burden away from the developer shoulders.

A defining trait of Objective-C is certainly its syntax, which mixes the use of the dot notation for property access (akin to Java attributes) and square brackets for message passing (similar to method invocation in Java). Moreover, public, and private modifiers are managed through the distribution of methods and properties between the ".m" and ".h" files (in the case of the public and private modifiers) while instance and class methods are indicated through the use of " - " and " + " respectively.

Other than the syntax, there are several constructs that differentiate Objective-C from other object oriented languages like Java, and even from the ANSI C programming language. Some examples of such constructs are: categories, blocks and protocols.

- **Categories:** Categories provide a way for extending the capabilities of a class without the need of inheritance or composition. They allow the programmer to modify and enhance the behavior of any custom or system provided class (e.g NSString) and add methods that will be available at runtime for all the classes within the app [9].
- **Blocks:** Blocks are similar to the closures or lambda expressions present in other languages like Javascript and Haskell. They allow a function to be executed outside of its scope while still accessing the variables available in it. Blocks are used very frequently when developing applications for iOS, because they provide a very handy way to implement object communication as well as completion handlers for asynchronous and concurrent operations [9] .
- **Protocols:** Protocols are similar to Java Interfaces, and they are widely used across all the frameworks that power the platform. They are the main building block of two important communication patterns: Delegates and Data Sources. Protocols, provide a way to loosely relate objects by defining a contract that specifies a set of functions that should be supported by the implementing class. Given that Objective-C doesn't allow multiple inheritance, but allows classes to implement as many protocols as needed, the possibilities offered by protocols in terms of decoupling, reuse and collaboration are widely used. In contrast to Java interfaces, protocols in Objective-C can distinguish between required and optional methods, which removes the need for providing empty implementations for uninteresting optional methods [9].

Finally, additional enhancements to the compiler as well higher level APIs have been added to make easier cumbersome tasks like memory management and multithreaded programming. Such is the case of ARC and GCD. ARC or Automatic Reference Counting, is a compiler level enhancement that releases some of the burden caused by the memory management operations and specially for the need of deallocating unused objects [11]. Similarly, Grand Central Dispatch (GCD), provides a managed approach to multithreaded programming based on dispatch queues, in which asynchronous, synchronous, sequential and concurrent tasks can be scheduled and are seamlessly managed by the framework. This approach abstracts away the complexity implied by thread management, while giving the developer enough control to finely tune the execution of his code [3].

2.2.3 Apps' Architecture

Most of iOS applications are built using a Model-View-Controller (MVC) architecture [2]. In this architecture, classes are to be divided into three different groups according to their role and responsibility. This architecture favors separation of concerns and reusability by decoupling the models – that represent the domain entities and contain a good part of the business logic – from the GUI used to present them. Such decoupling is achieved by means of a third group of objects called controllers that are in charge of translating the user inputs into the corresponding updates in the models, as well as updating the user interface every time something changes in the underlying models.

From the software engineering point of view the MVC architecture is the combination of three design patterns, namely: Observer, Strategy and Composite. Whereby, the Observer pattern is used by the models to keep interested objects updated on any changes of its values. The Strategy pattern is used by the views to let its associated controller respond to certain events – like user interactions. Finally the Composite pattern is evident in the view hierarchies whereby a view can have subviews, that are views themselves and can have more subviews.

- **Model objects:** Represent the domain entities, and encapsulate the business logic [2].
- **View objects:** Represent the visual components that make up the graphic user interface (GUI) of an application. Typically a view object knows how to draw itself and is able to effectively capture the user interactions. Moreover, a view should provide a way to communicate

certain events to a controller as well as provide clear APIs that can be used to update the information it shows [2].

- **Controller objects:** Controller are the less reusable objects of an application. They act as the glue between the views and the models, keeping the information stored in the models aligned with the information displayed by the views and performing actions after a user interaction is captured [2].

Even though MVC is a very common software architecture, the implementation used in iOS is somehow particular. Particulary, the communication mechanisms that allow objects collaborate with each other, deserve a brief description.

- **Delegate [View / Controller]:** Delegates are an implementation of the Strategy pattern that is frequently used by views to communicate with their controllers. The interesting thing about delegates, is that they imply a decoupled communication between objects. Using delegation, an object A can transmit the responsibility of performing some operation to object B, as long as object B implements the required methods of some protocol that is known by both A and B [2].
- **Data Source [View / Controller]:** Similar to delegates, data sources are typically used by views to let other objects provide the information that they should show. It is very usual to find a View Controller that becomes both a delegate and a data source of a particular View and does so by implementing both the specified delegate and datasource protocols [2].
- **Target-Action [View / Controller]:** This type of communication allows control views (buttons, sliders, checkboxes, etc) to communicate to a target object. The communication happens via a defined action that is triggered when a particular control event has taken place. Typically, a View Controller registers himself as the target object of several actions that occur within the view – touch, drag, value change, etc. Moreover, it specifies for each action a method that should be triggered every time the action is captured [2].
- **Outlets [Controller / View]:** Every View Controller has a reference to the view it controls. However a view is usually composed of several subviews that need to be updated in response to user interactions or changes in the underlying model. To provide easy access to such view components, a controller could declare several properties, marked as IBOutelts. In this way, every time a view controller needs to update a

subview, it could do it directly instead of having to examine first the subviews of its view. Similarly, if there are subviews that do not need to be updated, the view controller could ignore them by not providing any outlets for them. From the software engineering point of view, an outlet relation is equivalent to a ≪has-object≫ relation; however, the reason IBOutlets are treated specially, is because the actual relation is archived on the view's nib file and is established only when needed [2].

- **Key-Value Observer [Model / Controller]:** According to the MVC architecture, a controller may hold a reference to both its model and its view, allowing it to directly communicate with them. Since the opposite is not true for a model, there is a need for a decoupled mechanism that will allow a model to inform its controller every time there has been and update on its values, so that the controller can refresh the view accordingly. Such mechanism is known as Key-Value Observer or KVO. Through KVO, a controller could observe a particular property of the model and get notified every time such property has changed [2]. **Segues [Controller / Controller]:** A typical storyboard based iOS app has have several scenes. Each of these scenes, has a View Controller, which in turn has a reference to its view and possibly to a model object. To transition between the scenes of a storyboard, segue objects can be used. There are several kinds of segues offered by the UIKit framework, and should the programmer need it, custom segues can also be defined through inheritance [13].

Core Objects

Typically, iOS applications use the Cocoa Touch Framework. Like any other framework, Cocoa needs several objects to establish bidirectional communications with the application's custom code. Some of this objects are usually provided by the programmer, while others are already supplied by the framework.

- **Main:** Just like in regular C programs, the main file contains the entry point of the application. When developing applications using Cocoa Touch this file is rarely modified [7].
- **UIApplication:** This object controls the execution of the application and is the one in charge of managing the app's run loop [7].
- **AppDelegate:** Implements the UIAppDelegate protocol in order to

act as the UIApplication delegate. Through this link, an app delegate is notified after every change in the state of the application [7].

- **UIWindow:** Is the top of the view hierarchy and as such is in charge of managing the views that are displayed to the user [7].

2.2.4 Persistence

There are several ways to provide persistence for an iOS app, and the decision of which of them is the right one depends on the kind of the application being developed. It is not rare, however, to come across applications that need to make use of more than one persistence strategy due to a particular business requirement, or simply due to the constraints imposed by a mobile environment – having to deal, for instance, with unreliable network access. The following list presents some persistence alternatives that are offered in iOS.

- **Property Lists:** Property lists provide a good alternative to store simple data as XML or in a binary representation. Only a few Foundation classes are property-list compliant and therefore can be serialized and deserialized as property-list's data items, namely: NSArray, NSDictionary, NSString, NSDate, NSData and NSNumber. This persistence strategy is advised for use cases in which small quantities of simple data needs to be persisted locally. Additionally, since property lists are often used to store user configurations, the Foundation Framework offers an API that covers common management tasks of these files [10].

- **File System:** Through this method, an application is able to store data in files that live within the application's sandbox. This persistence method is frequently used directly to provide caching for externally acquired objects, store additional application objects and assets (e.g in-app purchases) or save basic user configuration files [5].

- **Archives:** In contrast with property lists, the object archival strategy offers the possibility of storing complete and complex object graphs to secondary storage. Almost any object can be archived as long as it implements the required methods of the NSCoding protocol. Through archives arbitrary objects can be encoded as byte streams and stored, as well as decoded and turned back into the objects they were originally encoded from. Object archival offers a viable persistence strategy for the model layer of an application [1].

- **Core Data:** Core data is, in essence, a model layer technology. It provides a managed approach to deal with persistent object graphs, and offers operations like lazy object loading, relationship management, undo changes support, as well as a powerful query system. Core Data is traditionally used on top of a SQLite database that uses a private format, and that is only accessible through the APIs offered by the framework [6].

- **SQLite:** This alternative allows a developer to create the persistent layer of an application relying on a familiar technology like SQLite [4]. In contrast with Core Data, working directly with SQLite allows a developer to finely tune every aspect of the data access logic. However, the level of abstraction of this approach is much lower, and as result most of the management services need to be implemented by the developer.

2.2.5 Application Sandbox

Every iOS application runs within its own sandbox, that isolates it from the other applications installed in the running device. This feature aims to enhance the security of the system by allowing an application to access, manage and manipulate only the files that concern it, as opposed to allow it manipulate arbitrary files across the file system [5]. The sandbox of an application has a predefined structure that classifies the files managed by the application according to their purpose and provenance. The sandbox of a typical application contains the following folders and files:

- **AppName.app:**:The application's executable file.

- **Documents:** iTunes backed up folder. User produced objects as well as user configurations should be stored here.

- **Documents / Inbox:** This folder stores external documents. Examples of files that can be found in it are the attachments of an email that need to be viewed using a different application.

- **Library / App Resources** In this folder user created documents and additional application assets are stored. This folder is backed up by iTunes.

- **Library / Caches**: An application may use this folder to cache expensive and externally obtained objects. This folder is not backed up by iTunes.

- **Library / Preferences** User configuration files and other settings related documents should be archived under this folder.
- **tmp:** A folder where non essential files and documents can be cached. The reason why only non-essential objects are supposed to be archived under this folder is because the OS may decide to purge it in case of needing additional disk space. In consequence, this folders is not backed up by iTunes.

2.2.6 Networking

Very frequently, mobile applications need to communicate with external entities using protocols like http or https to access, download or store data and files. Likewise, to provide multi-platform deployments and enhance the user experience, application developers usually rely on a centralized server architecture that offers a unified service interface (e.g. a REST API).

To address the previous networking scenarios, Apple offers several low and high level implementations that can be used to suit very specific use cases. In particular, for communications over HTTP and HTTPS there is a group of classes known as the URL Loading System that together offer a complete set of APIs to transfer data as well as upload and download files from an external HTTP server [12].

2.2.7 Views Management

iOS offers several alternatives when it comes to creating and managing the views that make up the user interface of an application. Some of them are very intuitive and allow developers to layout and connect the main screens that make up an application, while some others require a deeper understanding of the available APIs. In the following list we provide a brieff overview of the existing alternatives for Views Management offferd by iOS.

- **Storyboards:** Storyboards provide a way for graphically laying-out the scenes that make up the graphic user interface of an application. The negative aspect of storyboards is that their underlying textual representation is hard to read making the merge operations, needed when working in teams using a code repository, quite difficult to carry out. Moreover, laying out very complex applications using storyboards can quickly get very hard to manage [13].

- **Nib files:** Just like with storyboards, NIB files allow developers to compose the a view of the application through a graphic editor. In this scenario, a view can be a specific component or a complete screen of the app. In contrast with storyboards, NIB files do not describe the transitions between the views of an app, requiring the programmer to manage this aspects through code [14].

- **Programatically:** Views can also be created entirely from code; and sometimes to achieve certain behaviors there is no way around it. The caveats of programmatically creating all the views of an application regard the complexity implied by coding aspects of the view that are simpler to achieve using a graphical tool, e.g layouts and constraints management [14].

Chapter 3

Approach

3.1 Overview

The development of our project was divided in three moments (Figure 3.1). First, we identified a set of candidate extensions that could be added to IFML, using the knowledge we had gained about the iOS platform during the background review. Then, we focused on creating the code generator for iOS, which would produce the Objective-C implementation of an app modeled with IFML. Finally, using the insights gained during the generation process, we revised some of the extensions we had proposed, and drafted our final proposal.

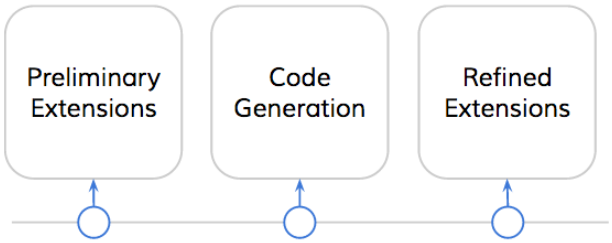

Figure 3.1: Approach - Overview

In this chapter, we will give an in depth look to the particular methodologies that we used to achieve the two objectives of our project. We also present, at the end of the chapter, a brief overview of the main tools and programming languages that enabled our final solution.

3.2 Extensions

To tackle the first objective of the project, we strated by analyzing the type of apps that could be built in iOS, and more specifically observing the type of features and functionalities they offered. We began looking at the gestures that were available, the types of sensors that could be used, and the strategies that enabled inter-app collaborations. We then looked at IFML to see if there was a way in which a software designer could express these traits in his models — whether a form could be submitted by shaking the phone, or if a particular action could be triggered by an update on the device orientation. Every time we couldn't find a way to represent one of these features, we proposed an extension.

The other place that proved to be a good source of inspiration, was the actual code generation effort that we had undergone to achieve the second goal of our project. The approach we followed in this case, was simple. We reviewed all the assumptions that we had made about the implementation of a particular IFML entity in iOS, and then analyzed whether an extension could have helped to disambiguate the matter. For instance, while developing the code generators, we assumed that all the user triggered events – Select Event, Submit Event, and plain View Element Events — were captured after a user touch. This was an over simplification of what can be achieved in a mobile environment — where a user may touch, tap, slide and even shake his device to trigger an event. In all these cases we added entities with more specific semantics, that would allow designers express their intent more precisely – in our previous example, we added a set of extensions to the event hierarchy of IFML. We repeated the same exercise with other entities, and produced the final extensions proposal.

To organize our extensions, we grouped them into three different packages according to their scope, and the core entities they extended from. In this way, the General Extensions Package contained classes that were valid beyond a mobile environment, like an Application Lifecycle Event, a Multiline Field or an Image Attribute. The Mobile Extensions Package instead, clustered entities that were more tightly coupled to the constraints and possibilities offered by mobile technologies, like the tactile and sensor events. Finally, in the Private Extensions Package, we grouped classes that extended the Context entity of IFML, which as of this writing, is not allowed outside a private scope by the IFML standard [21].

3.3 Code Generation

The diagram shown in Figure 3.2 provides a clear view of the methodology that guided our work during this part of the project.

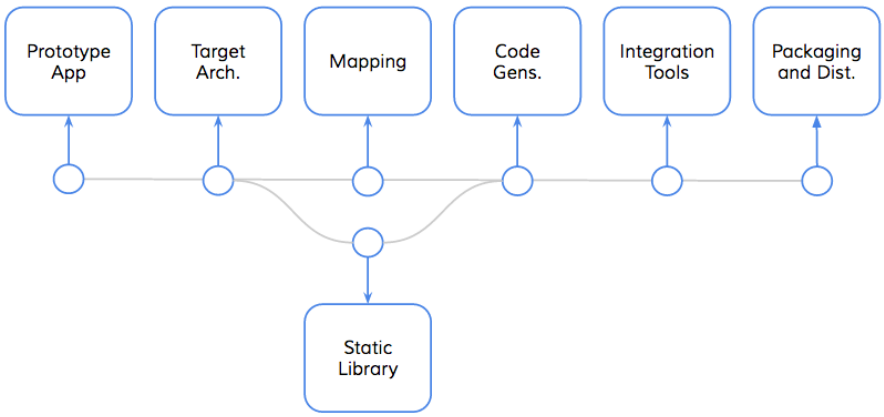

Figure 3.2: Approach - Code Generation

3.3.1 Prototype App

Our first step was to choose a suitable prototype that could guide the generation process. We looked for an application that used a meaningful subset of the IFML entities, and whose model had already been designed with the model editor. After some analysis, we chose one of the sample applications that shipped with the IFML Model Editor. We applied some minor modifications to its model, and then moved forward to provide an iOS implementation for it.

Choosing this prototype was very important. It defined which entities were supported and which were not. But the next step, was the one that truly defined the shape of our solution.

3.3.2 Target Architecture

We defined the architecture of the generated apps in two ways. First, we used a software driven approach to described the main application components and their relations. Then, we used model driven approach, to formalize the structure of this architecture in a metamodel.

The software driven approach was concerned with creating a taxonomy for the objects that composed the application, assigning responsibilities for each of them, and describing the way different types of objects collaborate with each other. In contrast, the model driven approach focused on representing the application components at a level of abstraction that was farther away from the implementation details.

This approach allowed us to think about the transition that takes the IFML model of an application to its iOS implementation, as a two step process. First the IFML entities had to be mapped into the metamodel that defined the target architecture — a Model to Model transformation. Then, the mapped entities had to be translated into the code of the target platform — a Model to Text transformation.

In this way, our next step was to understand the equivalences between IFML and the entities in the newly defined target metamodel.

3.3.3 Mappings

There were two ways in which we could perform the mapping between IFML and the entities of the target metamodel. The first one, was to write a set of rules using a model transformation language like ATL. The second one, was a more flexible approach based on diagrams.

The first strategy had the advantage of being formal, allowing the model transformation to be done automatically through the tools provided by the Eclipse Modeling Framework (EMF). On the other hand, it increased the technical demands of the project, reducing the time that we could devote to other aspects of the research.

The other strategy, instead, had the benefit of being lightweight and faster to perform. On the flip side, however, it moved most of the transformation burden into the code generators, which instead of having to generate code out of the entities of the target metamodel, had to do so from the entities of the IFML model.

At the end, we opted for a hybrid approach. We preserved the formality of the two step transformation process provided by the model driven approach, as a theoretical tool that described the shape of our solution. While the actual implementation was achieved by describing the mappings using a set of diagrams and placing all the transformation logic into the code generators.

Once the equivalences between IFML and the target architecture were well understood, we focused on the code related tasks. Particularly, we set out to discover how much code had to be generated.

3.3.4 Static Library

To understand how much code had to be generated during the M2T transformation step, we started by analyzing the code of the prototype app. We separated the pieces of code that were model independent — those that would remain the same across the generated applications—, from those that depended on the input model. The model independent pieces were wrapped into a static library; the other sections of code instead, were singled out as the ones that would be dynamically generated by the code generators.

We wrapped the model independent pieces into a static library because in this way we could reduce and simplify the code that had to be generated. The library contained a set of basic classes, a collection of default views and other utilities. The basic classes would be extended by the generated code, allowing us to factor out into parent classes all the complicated logic and control flow. The default views and the utilities instead, allowed us to provide the generated apps with a basic user interface, that will further reduce the time between modeling and the first compilation of the apps. Additionally, since the dependencies between the generated code and the classes in the library were resolved at compilation time, we were able to test and debug the library independently, which gave us a lot of flexibility.

At the end of this stage, we had a good grasp of what needed to be generated, and could start working on two parallel fronts. One that was be focused in writing and testing the code contained in the static library, and other that focused on the development of the actual generators.

3.3.5 Code Generator

There are several patterns that can be used to implement a code generation tool — Markus Voelter [25] has several good advices on this topic. In our case, we decided to use the "Template and Filtering" [25] approach. In consequence, all the generators we developed had a very similar logic. They would first query the IFML model, to find the subset of entities that would be transformed, and then, through the usage of templates and simple logic, they would generate the target source code.

An important advantage we had, was given by the IFML metamodel, which had been developed using ECORE. This implied, that instead of having to deal with cumbersome and time consuming tasks like parsing the model file, and providing an object oriented representation of the elements in it, we could use the APIs provided by the Eclipse Modeling Framework (EMF)[1], which made trivial these operations.

Our technical stack was completed by Xtend[2] — a dialect of Java, that compiles to regular Java code —. We chose this tool, to develop the actual generators because of its powerful template capabilities and very concrete syntax.

In combination, the EMF infrastructure, Xtend, and the "Template and Filtering" [25] strategy made the implementation of the transformation tool much easier, and kept the focus on the real objective: producing Objective-C code out of an IFML model.

The only caveat we found while using these tools, was the management of static files. In our project this meant, copying into the final output the classes of the static library and a set of boilerplate files that remained unchanged across the generated applications — e.g the Main file.

We addressed this issue through a combination of strategies. First, since the number of boilerplate files was not too high in our case, we added generators whose only purpose was to create the same files every time. As for the files in the static library, since they also had to be linked to the classes in the generated code, we opted for a more robust strategy that involved the usage of cocoapods[3] — a popular dependency manager for iOS and OSX development.

3.3.6 Integration Tools

The goal of this final step was to integrate the generated files into the tools used by iOS developers. This implied the generation of the Xcode project file and the development of a script that retrieved the static library and linked it to the generated project.

This additional effort allowed us iterate much faster in the development of

[1] http://eclipse.org/modeling/emf/
[2] http://eclipse.org/xtend/
[3] http://cocoapods.org/

the generators. Instead of having to manually create a new Xcode[4] project and add the generated files along with the static library to it, we could only execute the scripts and have it done for us in a couple of seconds.

Moreover, we could tighten up even more the final output project, so that the folder structure of the generated app will be indistinguishable from the structure of an app that had been manually created.

3.3.7 Packaging

Up to this point, we had created three different tools that were tightly related. First, the Code Generators, which had been distributed as a standalone jar. The project builder script, which had been developed using ruby . And Finally, the Library Linker that would download the Static Library from a centralized repository and add it to the compilation path of the generated Xcode project, leveraging the capabilities offered by cocoapods.

The biggest issue caused by having these separate applications was usability. An IFML designer interested in generating the iOS version for one of his models, would have to launch each of these tools, execute them in the right order and remember to provide the appropriate parameters for each of them. To solve this issue, and ultimately, make the leap from IFML into iOS, as frictionless as possible, we developed a desktop app with a graphic wizard to guide users through the generation process.

3.4 Tools

The following table summarizes the IDE's and programming languages used for developing the project deliverables.

Tool	Programming Language	IDE
Code Generators	Xtend / Java	Eclipse
Static Library	Objective-C	Xcode
Project Builder	Ruby	Xcode
Library Linker	Bash	Xcode
Generation Wizard	Objective-C	Xcode

[4] https://developer.apple.com/xcode/ide/

Chapter 4

Extensions

Currently, IFML provides software designers with a set of entities that can be used to model desktop, web and even the fundamental aspects of mobile applications. There are, however, several features of the mobile environment that are not covered yet by the language. For instance, a software designer doesn't have a way to query the hardware capabilities of the device in which his application is running on, to understand which sensors are available or what communication networks are currently active. Similarly, the existing Event architecture of IFML doesn't consider interactions beyond simple mouse and keyboard events, leaving uncovered tactile and sensor driven events. Finally, there are few, and very general, entities that can be used to model apps that can cope with the changing nature of the mobile context or blend nicely with the resource optimization strategies used by modern mobile operating systems.

For this reason, our goal during this part of the project was to provide a suitable set of extensions for IFML, that would allow software designers model modern mobile applications, considering the particular requirements that arise in this platform. Similarly, these extensions also had the benefit of potentially increase the effectivety of future code generators, since having more specific entities mean that better assumptions can be made regarding the functionalities expressed by software designers in their models.

As a result, we proposed 90 extensions that we divided into three packages: the General Extensions Package, the Mobile Extensions Package, and the Private Extensions Package. Figure 4.1, shows these packages, the number of entities they contained and the IFML entity they extended from. As for the specific extensions they included, in the next sections we provide a

brief overview describing some of them. A more complete description can be found in the appendices of this document.

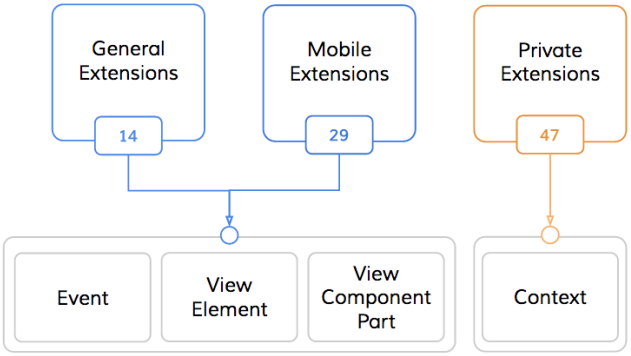

Figure 4.1: Extensions - Packages

4.1 General Extensions

The General Extensions Package, contains extensions that go beyond the mobile scope. Hence, the extensions discussed in this section could also be used for modeling desktop and web applications. Figure 4.2 shows an overview of the main entities contained in this package.

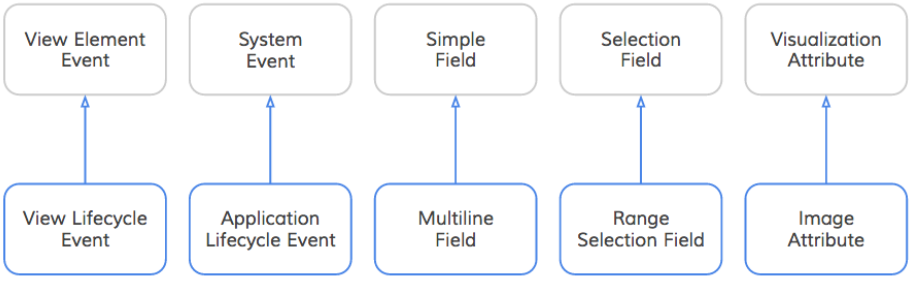

Figure 4.2: General Extensions - Overview

4.1.1 View Lifecycle Event

Traditionally, one of the main roles of the controller layer within a MVC architecture, is to keep the alignment between the information shown in the user interface and the data stored in the underlying models. Consequently,

this synchronicity needs to be enforced by the controller throughout the lifespan of the view it manages, including the following milestones:

- The creation and allocation of a view in memory.
- After a user interacts with the view aiming to change the state of the models.
- Every time a view is displayed, after the user has visited other sections of the app.

While the second case is somewhat covered by IFML (through View Element Events), the other two remain unsupported. To address the missing cases, we propose an extension to the System Event entity called View Lifecycle Event. This entity will allow software designers to specify the actions that should be triggered after the occurrence of an important milestone of the lifecycle of a particular view. As shown in Figure 4.3, the extension defines five milestones: View Did Load, View Will Appear, View Did Appear, View Will Disappear and View Did Disappear.

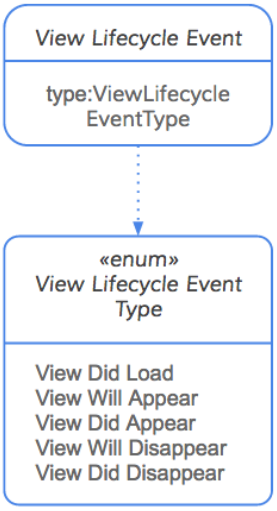

Figure 4.3: General Extensions - View Lifecycle Event

Using this extension:

- A social media app, could start a network request to obtain the posts of a user's news feed as soon as the view that displays this information has been loaded into memory.

- A task manager app, could update the main task list to reflect the changes applied by the user in a different section of the app.

4.1.2 Application Lifecycle Event

In order to optimize resource usage and provide the possibility of executing several applications concurrently, operating systems use several strategies. One of this strategies consist in assigning different states to the running applications depending on whether they are running in the foreground or in the background [7]. In consequence, to guarantee the best user experience, applications must supply mechanisms to take appropriated actions after being notified about a change in their state.

Even though, the current specification of IFML includes a general System Event that could be used in this scenario, it is not specific enough to allow software designers model applications that blend nicely with the resource optimization strategies of modern operating systems. For this reason, we propose an extension called Application Lifecycle Event, whose main purpose is to group together all the events that are related the with state transitions of an application, which in turn, will allow for the definition of more resilient apps. Figure 4.4 shows the 10 application states that can be modeled with this extension.

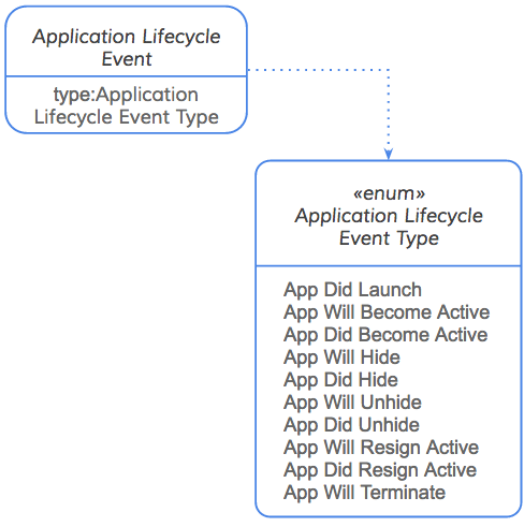

Figure 4.4: General Extensions - Application Lifecycle Event

Using this extension:

- A note taking application can save locally the user's work, once it is notified about being swapped into a background state, foreseeing a possible future termination.
- A productivity app, that allows several users to collaborate in the creation of a document, may notify the server before a user terminates the application, so push notifications are suspended for this user.
- A casual gaming app, may choose to pause the game timers when the application is moving into the background.

4.1.3 Multiline Field

Currently a software designer doesn't have a way for indicating if a Simple Field may allow single or multiple-line inputs, yet this distinction may lead to very different implementations. To disambiguate these cases, an extension called Multiline Field is proposed.

Using this extension

- A social network may model the user registration form using a Simple Field for capturing the user's name, and a Multiline Field for capturing the user's bio.
- An ecommerce app may use a Multiline Field to allow users write reviews about the products offered in the platform.

4.1.4 Range Selection Field

Range Selection Fields are meant to allow users select a particular value within a range of discrete or continuous values. To fully model this type of fields, this entity contains the following attributes

- minValue: the lower boundary of the range
- maxValue: the upper boundary of the range
- stepValue: the increment by which a user may update the field value.

Using this extension:

- An application for order processing may use a Range Selection Field with its minValue set to 0, a maxValue equals to 100 to model a discount percentage field.
- A hotel booking application, may use a Range Selection Field with its stepValue set to 1, to model the number of guests that will be staying in a room.

4.1.5 Image Attribute

Currently designers don't have a way to differentiate between a Visualization Attribute that needs to be rendered as text from a piece of data that needs to be shown as an image. To disambiguate these cases, an extension of the Visualization Attribute, called Image Attribute was be introduced.

Using this extension,

- An email application, may model the user profile section using a Visualization Attribute for the user's name field, and an Image Attribute for the user's picture.
- An ecommerce application, may model a catalog of products using a List View Component that shows for each product a Visualization Attribute for the product's name, and an Image Attribute for the product's picture.

4.2 Mobile Extensions

One of the strongest appeals of mobile applications comes from their ability to exploit the interaction capabilities of the devices they are executed on. For this reason, allowing users to interact with an application using tactile gestures or by physically moving their devices has become an important concern of the front end development of mobile apps. In contrast, one of the biggest challenges faced by developers comes from the inherently tougher environment in which mobile applications are executed on.

While IFML allows software designers to model certain user interactions, and respond to some system triggered events, there are still several uncovered aspects that are needed to enable the definition of resilient applications that can take full advantage of the interaction capabilities offered by the mobile platforms.

On the other hand, mobile operating systems have an active role and interact frequently with applications. Hence, mobile operating systems are not only in charge of managing access to shared resources like memory and storage, but also of tasks like delivering sensor readings to the applications that require them. Additionally, due to the relatively short amount of available resources in mobile platforms and the optimization strategies put in place by the operating systems to cope with such scarcity, mobile apps need to be able to receive system notifications and react appropriately to them in order to provide a consistent and reliable user experience.

System Events in IFML are meant to model occurrences that are not directly initiated by the user, but that are instead triggered directly by the operating system. Even though these general events can be used to model some of the previously described cases, a more subtle taxonomy will allow software designers to specify how their mobile applications should react to specific occurrences. In particular, we propose a taxonomy shaped by two entities: Mobile Sensor Event, and Mobile Resource Events.

Mobile Sensor Events model all occurrences generated by the device sensors. Mobile Resource Events instead describe the notifications triggered by the operating system regarding the shortage and changes on the availability of certain shared resource. Furthermore, we have proposed an extension for the View Container class of IFML, called Screen, that represents each of the scenes that makeup the user interface of a mobile application. Figure 4.5, illustrates the main extensions of this package, while the following sections describe briefly their purpose and utility.

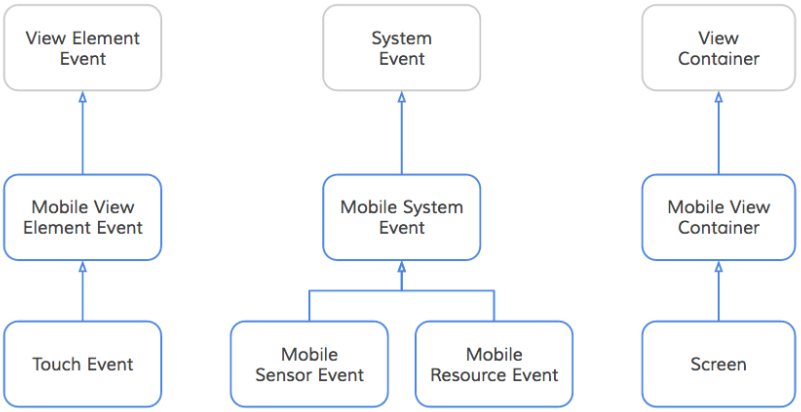

Figure 4.5: Mobile Extensions - Overview

4.2.1 Touch Event

Mobile applications need to allow a richer set of user interactions that involve tactile interfaces. While in a desktop environment, a user can interact with an application by means of a mouse and a keyboard, in a mobile context a user can interact using a tactile surface that could leverage several standard gestures like Tap, Pan, Swipe, Flick, Pinch, Rotate, Drag and Long Press.

To support this interactions, we propose an extension to the View Element Event called Touch Event. This entity will group together the set of standard gestures that are supported by the major mobile platforms. Figure 4.6 shows the six interactions that are modeled through this extension.

Using this extension a software designer may specify the specific tactile events that can be captured by a View Element and the action that may be triggered in response. For instance:

- An application that shows a gallery of pictures may allow users to swipe horizontally in order to navigate through the pictures.
- The same gallery application may allow users to zoom in and out within a picture using a pinch gesture.

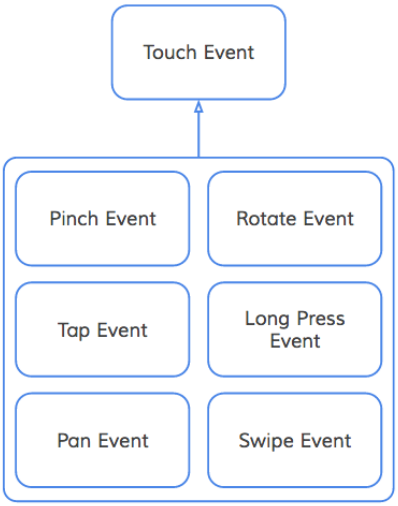

Figure 4.6: Mobile Extensions - Touch Event Hierarchy

4.2.2 Mobile Sensor Event

Mobile applications need to allow interactions with the outside world beyond user inputs, that is considering as sources of events other agents like the device sensors. The events produced by these sensors are usually captured by the operating system and delivered to the application upon user permission. For this reason, we propose an extension to the System Event called Sensor Event that identifies three general categories of events, namely: Proximity Event, Location Events and Motion Events.

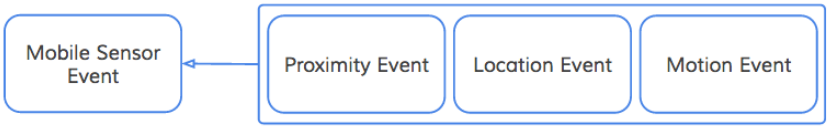

Figure 4.7: Mobile Extensions - Mobile Sensor Hierarchy

- Proximity Events are produced by the proximity sensor of the device, is frequently used by applications that require the user to place the device near to his ear such as telephone apps.
- Location Events refer to changes in the geographical position of the user. They are captured by the device GPS and are frequently used in applications that involve maps and navigation.

- Motion Events group events triggered by the Accelerometer, Gyroscope and Magnetometer of the device, which identify if the device has been rotated, if there has been a change in its orientation or if variations on its linear velocity have been detected.

Using this extension,

- An application that wants to show the linear acceleration of the device in the user interface, may have a Mobile Sensor Event of type Accelerometer, that will be fired every time there is a change on the linear velocity of the device.
- A productivity application may update the configuration of the user interface, every time the user rotates his device.

4.2.3 Mobile Resource Event

The OS in mobile devices like in desktop platforms, is in charge of managing shared resources like memory, secondary storage and network connections. While desktop applications usually run on devices with very large and stable resources, mobile applications need to cope with resource scarcity and instability.

The Mobile Resource Event is meant to encompass all these occurrences. Figure 4.8 shows the types of events that could be captured using this extension.

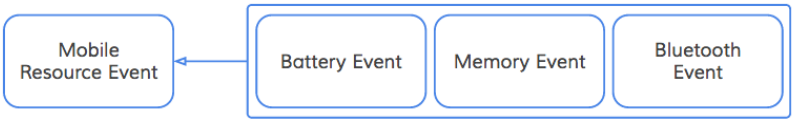

Figure 4.8: Mobile Extensions - Mobile Resource Event

Using this extension

- An application may decide to relase unused resources after receiving a memory warning from the operating system
- A podcast application may choose to reschedule an expensive network operation, in case the device battery is running low.

4.2.4 Screen

Akin to the Window extension of IFML, the Screen entity is aimed to represent the basic container unit of a mobile application. Furthermore, to cover the most frequent usage scenarios of this entity, we have given it two attributes:

- isModal: To explicitly model wether a screen should be shown modally or modeless, software designers may update the Boolean value of this attribute.
- hasNavBar: While some platforms like Android use default navigation stacks to keep track of the screens that have been visisted by the user, other platforms like iOS, require an explicit declaration. For this reason, the Screen Entity contains a hasNavigationBar attribute that allows designers to declare whether a particular screen should be part of a navigation stack.

Using this extension,

- A newsreader app, may model the share section of the application as a modal Screen to streamline the social features of the app.
- The Window containers of the dictionary app that we introduced in Chapter 2, will be replaced by Screens, which will allow us to specify aspects like back-navigation through simple attribute values.

4.3 Private Extensions

To address the wide variety of devices in which a mobile application may be executed, IFML needs a better model of the mobile context. Particularly, a model that takes into account variables that affect the user experience, like the screen size, the available sensors, and the communication networks that are active while an application is running.

Currently, IFML offers entities like the Context Dimension and the Context Variable, which are meant to capture information about the running environment. These entities, however, are too general and therefore more specific extensons may be needed.

Figure 4.9, shows in blue a subset of the entities we have proposed to address the previously described scenarios. These entities are grouped under the Private Package, since as of this writing, the IFML standard [21] doesn't allow extending the Context entity outside a private scope.

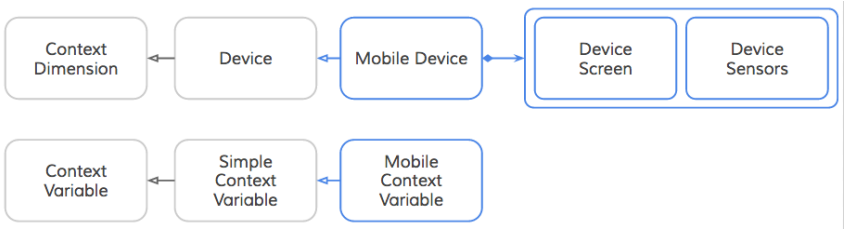

Figure 4.9: Private Extensions - Overview

4.3.1 Mobile Device

The Device entity in IFML focuses on the characteristics of the hardware in which an application is running on. This entity, however, is too general to describe the nuances of modern mobile devices – leaving uncovered aspects like sensor availability and the main screen features. To bridge this gap, we propose the introduction of a new entity, the Mobile Device.

As shown in Figure 4.10 a Mobile Device entity could be associated with Sensors like: Magnetometer, Accelerometer, Proximity, Gyroscope, GPS, Camera, Video and Microphone, which are the hardware appliances supported by most of the current mobile devices. Similarly, a Mobile Device could be described using a Screen entity, to report the width, height and the pixel density of the device screen needed by a particular Viewpoint.

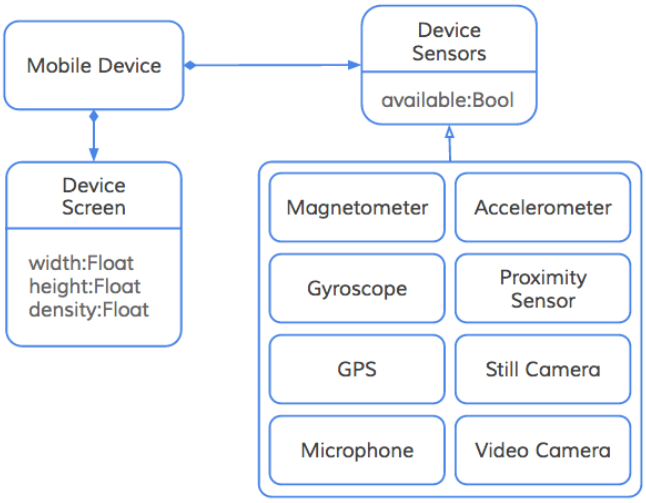

Figure 4.10: Private Extensions - Mobile Device

Using this extension a software designer may define different Viewpoints for his application based on the sensors available in the running device, and the characteristics of the device screen.

4.3.2 Mobile Context Variable

The Mobile Context Variable entity models a set of run-time variables that provide developers access to the readings of each of the sensors packed in modern mobile devices. Figure 4.11, shows the entities that can that can be used to query information about the running environment, as provided by the sensors of the device.

Since, by definition, the Context Variables can only hold a single value, more specific extensions had to be added to capture multivalued information. As shown in Figure 4.12, the Acceleration Entity was further specified into Acceleration-X, Acceleration-Y, and Acceleration-Z. Similarly, the Rotation entity is divided into three subclasses: Rotation-X, Rotation-Y and Rotation-Z. And the Attitude entity is extended by the Yaw, Pitch and Roll classes, which store the device rotation across the main spacial axes.

Other Context Variables (Figure 4.13), like the Battery Status and the Orientation take their values from enumerations. In this way, the Orientation of a mobile device could be described as Unknown, Portrait, Landscape, Portrait Upside Down, Landscape Right, Landscape Left, Face Up and Face

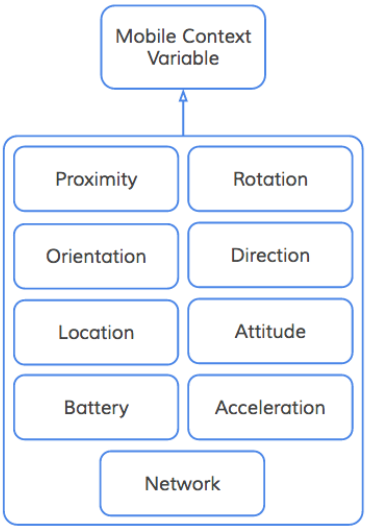

Figure 4.11: Private Extensions - Mobile Context Variable

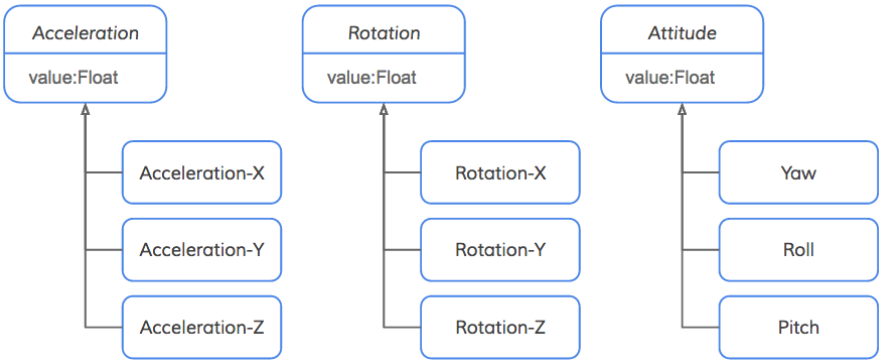

Figure 4.12: Private Extensions - Acceleration, Rotation and Attitude

Down. And the Battery Status as Unknown, Unplugged, Charging or Full.

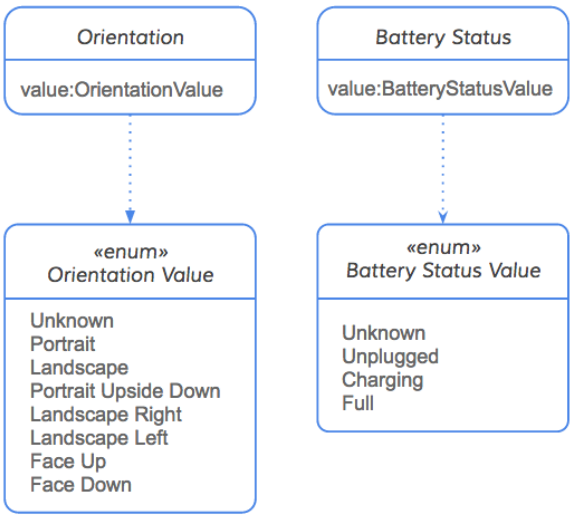

Figure 4.13: Private Extensions - Orientation and Battery Status

In contrast, the Network and the Proximity variables shown in Figure 4.14, hold boolean values. In the first case, a boolean value indicates whether a particular communication network is active and ready to be used. The value of the Proximity variable instead, determines whether the device is close to the user's face.

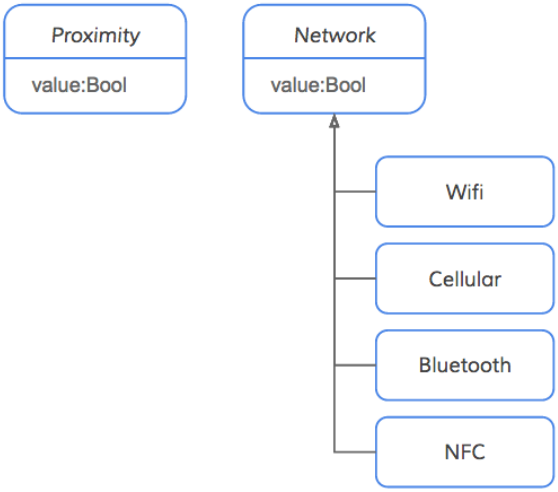

Figure 4.14: Private Extensions - Proximity and Network

Finally the Direction and Location variables, which take their values from

the GPS and magnetometer sensors, are divided into the entities showed in Figure 4.15.

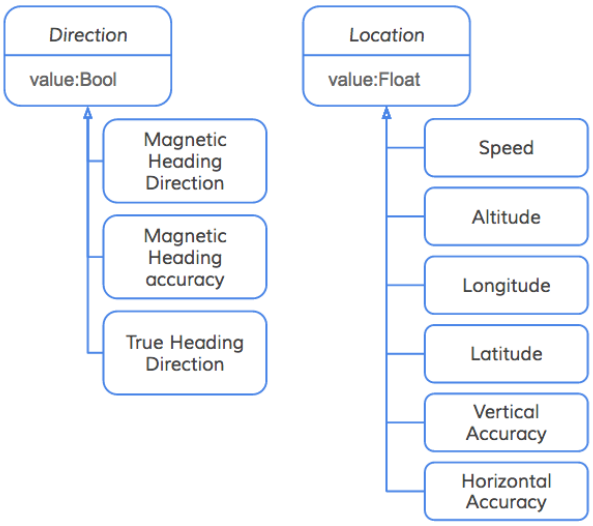

Figure 4.15: Private Extensions - Direction and Location

Using these extensions in combination with Activation Expressions, IFML designers will be able to model applications that react to their environment, and make a better usage of the available resources.

Hence, using these extensions:

- The download button of a podcast application may be enabled only when the user is connected to a Wifi Network.
- A recommendation app could show certain information only if the location readings of the device conform to certain constraints.
- An application may show a notification to the user about the battery status, before starting a long and delicate operation.

Chapter 5

Code Generation

To transform the IFML model of an application into Objective-C Code, we followed the Model Driven Engineering (MDE) approach described in Figure 5.1. First, the IFML model of the app was mapped into a target model, which described the concepts of IFML in an MVC fashion. Then, the model of the app was transformed into equivalent Objective-C code through a set of Xtend templates. Finally, the generated code was packed as an Xcode project linked to a Static Library – from which the generated code inherited most of its functionality and behavior.

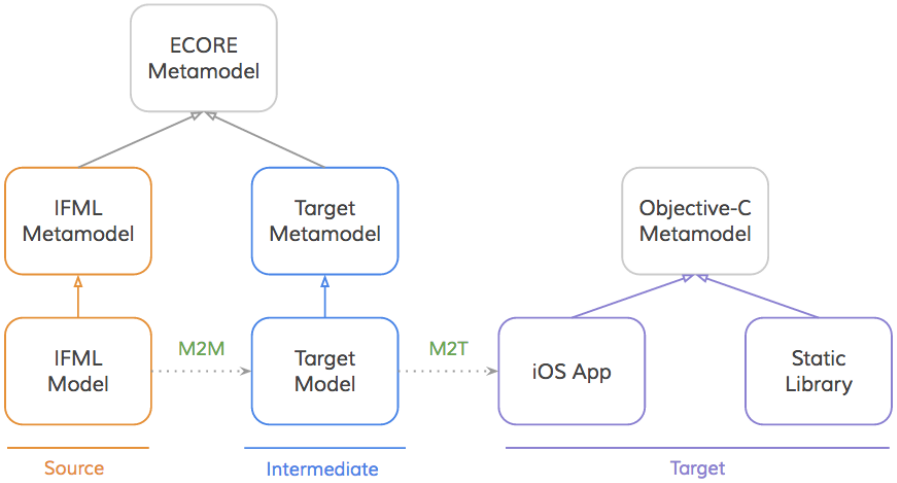

Figure 5.1: Transformation Overview

The figure, however, fails to illustrate the different steps that we took in order to achieve the transformations. For this reason, in the following sections

we describe in depth the results produced after completing each of the seven steps of the methodology we introduced in Chapter 3.

5.1 Prototype App

Using the tools developed during the project we were able to produce the iOS version of the “Movie Manager” app – a simple application that ships with the IFML Model Editor to showcase its capabilities.

Despite its apparent simplicity, the ”Movie Manager” app contains a representative subset of the IFML entities — Lists, Details, Forms, Windows, Fields, Data Bindings, Visualization Attributes, Actions, Parameters, Navigation Flows and others are all included in it. Moreover, the model of the application had already been implemented using the IFML Model Editor – which was an important requirement for us, because the tools developed during the project were highly dependent on the EMF platform, and therefore having well formed models that followed closely the metamodel of IFML, was very important.

As we can see in Figure 5.2, the “Movie Manger” app has three windows. The “Favorite Movies” window, which has a List that shows the title and year of each movie. A “Movie Details” window, which displays detailed information about a particular movie. And the “New Movie” window, that contains a simple form to allow users introduce new movies into the list.

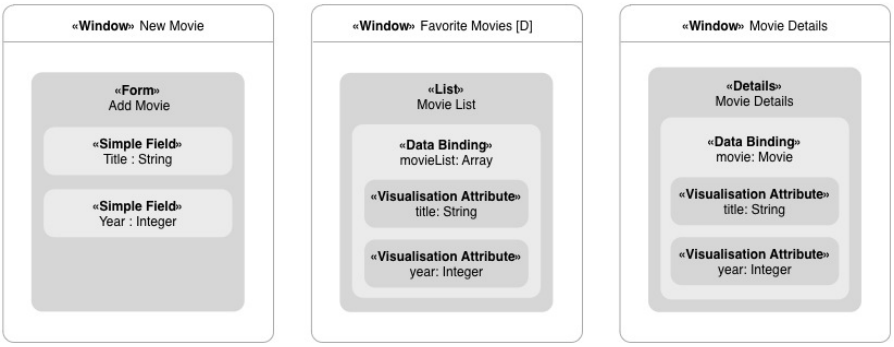

Figure 5.2: Movie Manager - Main windows

The “Favorite Movies” window is the application’s default screen, which indicates that it is the first window that is presented to the users. Furthermore, it contains an event called “Fetch Movies” that is fired after the window has been loaded. This event, triggers the execution of an action

with the same name, that in turn refers to a UML sequence diagram that describes its implementation.

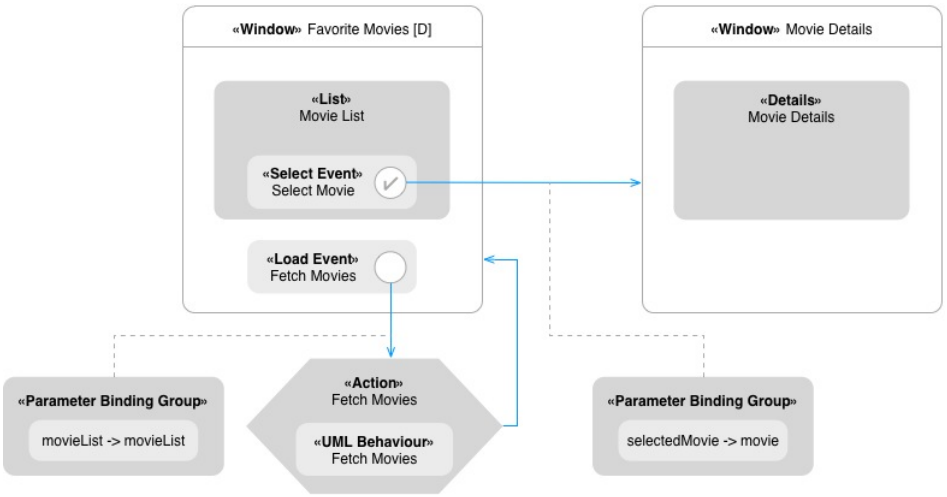

Figure 5.3: Movie Manager - Movie Details

In Figure 5.3, we can see that within the "Favorite Movies" window, lays a List component that publishes the information of a set of movies. This List includes an event of its own, called "Select Event", that is fired every time the user taps on one of the items in the list. As a result of this event, the "Movie Details" window becomes the active screen of the application, where users can see detailed information about the selected movie.

As for the "Add Movie" window, the initial screen of the application – "Favorite Movies", also contains an event called "Add Movie" that navigates users to it (Figure 5.4). Once there, users can use a simple form to introduce the year and title of a new movie, and then submit this information to add it to the list. This last step is achieved through a submit event, that takes the values captured in the form and transfers them to an action that takes care of the rest.

This concludes our brief review of the IFML model that describes the prototype application. However, a good way to conclude our discussion would be to show the result we obtained after applying our code generation tool to the model. Figure 5.5, contains the main screens of the application after being generated by our transformation tool. We have , however, replaced the default views generated by the tool, to give the app a more polished visual design.

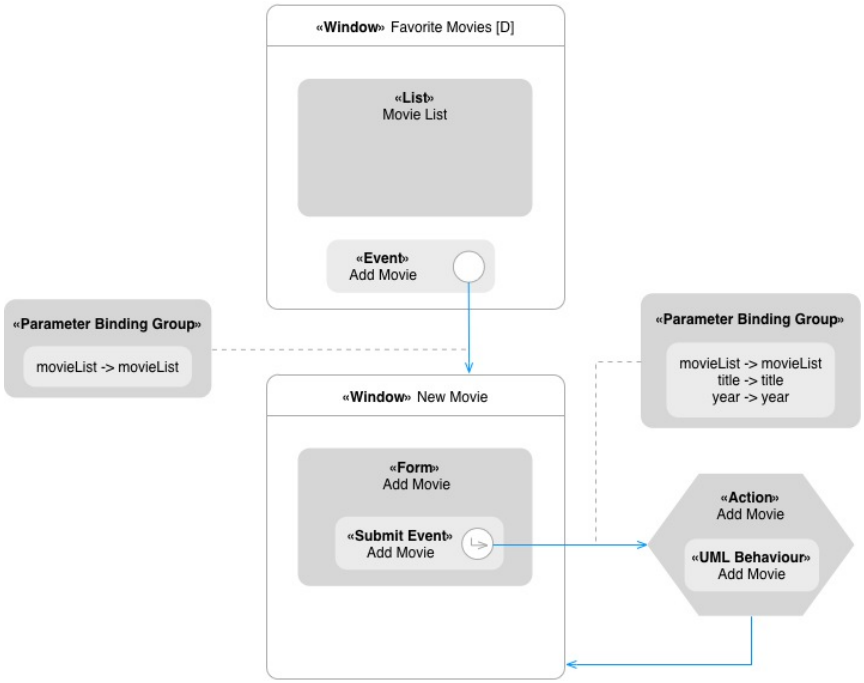

Figure 5.4: Movie Manager - Add Movie Form

5.2 Target Architecture

Defining a suitable architecture for the generated apps was an important milestone in the project. It described the shape of the the output code. And it defined a target model, against which the IFML entities should be mapped into.

When designing the architecture, we used two different approaches. One focused on assigning responsibilities and providing a classification for the classes that will power the application – a software engineering approach. And other, focused on describing the target architecture as a metamodel – a model centered approach.

The software engineering perspective helped us shape and organize the code produced at the end of the transformations. Instead, the model centered approach gave us the theoretical tools needed to map the IFML entities into our architecture.

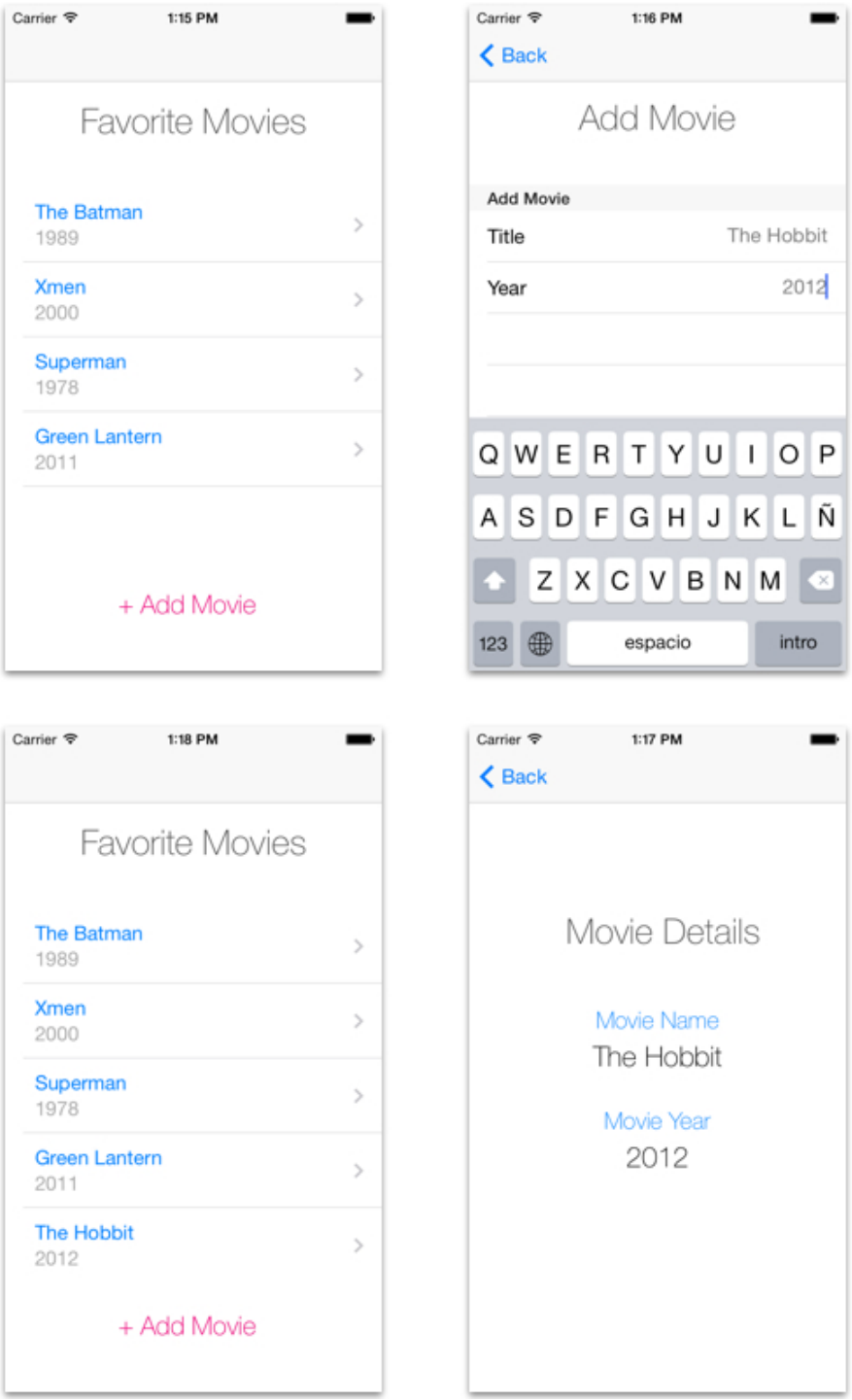

Figure 5.5: Movie Manager - Generated App

5.2.1 Software Driven Perspective

The shape of the code in the generated apps was guided by the architectural design shown in Figure 5.6

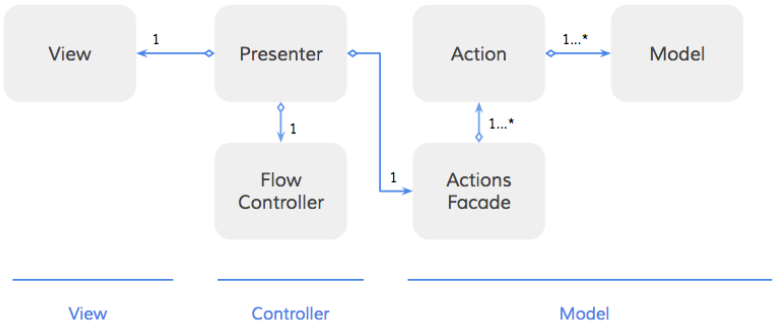

Figure 5.6: Target Architecture - Oveview

An object in this architecture can belong to one of four different categories: Views, Presenters, Actions and Models. Each of which has the following responsibilities:

- Views: Refer to the visual aspects of the application. In this category we can find the GUI widgets that define the user interface, as well as the logic that captures user interactions.
- Presenters: Gather the data that should be published, and prepare it for display. Presenters, also contain the logic that is triggered after a user interaction is captured by the view. Such logic often implies invoking an action, triggering navigation, or updating the presenter's view.
- Actions: Contain the application logic as described by the application's use cases.
- Models: Represent the business entities.

The second ingredient of the architecture is given by a couple of singleton objects. The Flow Controller and the Actions Facade.

- The Flow Controller manages the navigation between the different views of the application, taking care of the dependencies between them.

- The Actions Facade reduces the application dependencies on specific actions. It exposes a set of methods whose implementation consist of creating and configuring an Action object.

Using an architecture like this, our "Movie Manager" app would have: One Presenter for each Window; two Actions; and a single Movie model. Each presenter would be associated with a suitable View, while the the Actions will be shadowed behind the Actions Facade. Figure 5.7 shows a diagram that illustrates the described architecture.

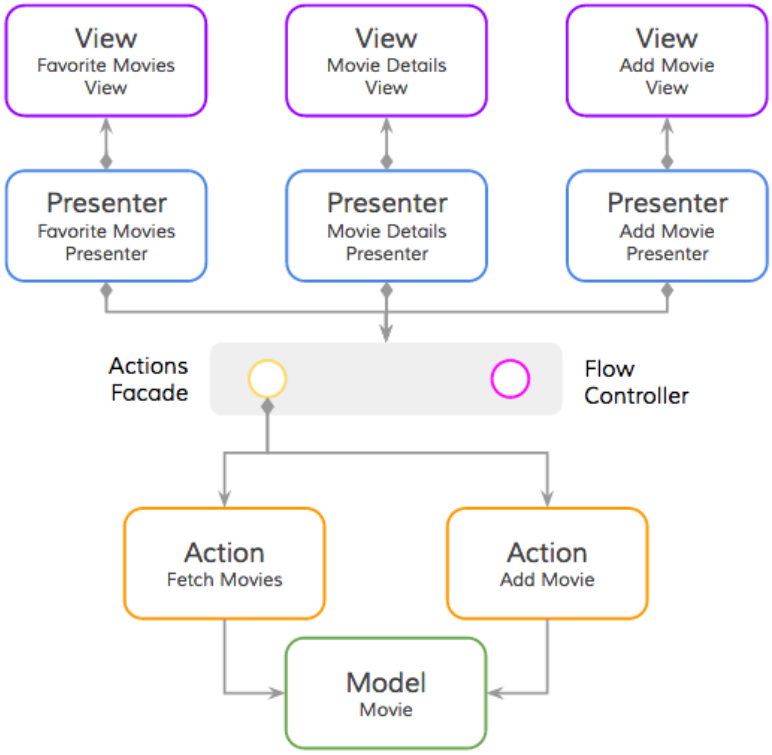

Figure 5.7: Target Architecture - Movie Manager

5.2.2 Model Driven Perspective

The model driven perspective allowed us to think of the architecture as being the target model of a model to model transformation.

Our ultimate goal was to transform the model of an application that conforms to the IFML metamodel into an implementation that conforms to the Objective-C metamodel. A good way to achieve this purpose was by first

transforming the IFML model of the application into an intermediate representation that was closer to the target implementation, and then transform the entities in this intermediate representation into the code of the target platform.

Looking at the architecture from this new perspective we discovered, an application could be described through the core entities shown in Figure 5.8

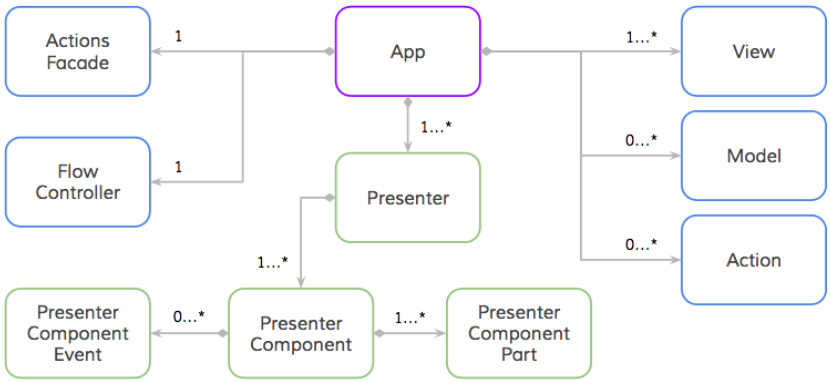

Figure 5.8: Target Architecture - Core Entities

Using this metamodel, an application is composed of an ActionsFacade, a Flow Controller, a set of Presenters, a set of Actions, several Models and several Views. This structure is very similar to the one we have presented in the software perspective. However this model introduces three new entities, the Presenter Component, the Presenter Component Event and the Presenter Component Parts.

As we discussed before, Presenters are in charge of two tasks: preparing the data for display and deciding how to react after a user interaction. In our metamodel, the presenters also have several Presenter Components each of which may have several Presenter Component Parts, and several Component Events as shown in Figure 5.9. This design is intentionally similar to the view hierarchy used by IFML, with the only difference that the concrete entities are lighter versions of their IFML counterparts, and only the attributes that are relevant for the M2T transformation were preserved during the mapping.

Up until now, we have covered entities that are either included in the architecture of the generated apps, or that are mock versions of the IFML classes. However, to completely describe the target metamodel, we need to introduce an additional family of entities: the method hierarchy (Figure 5.10).

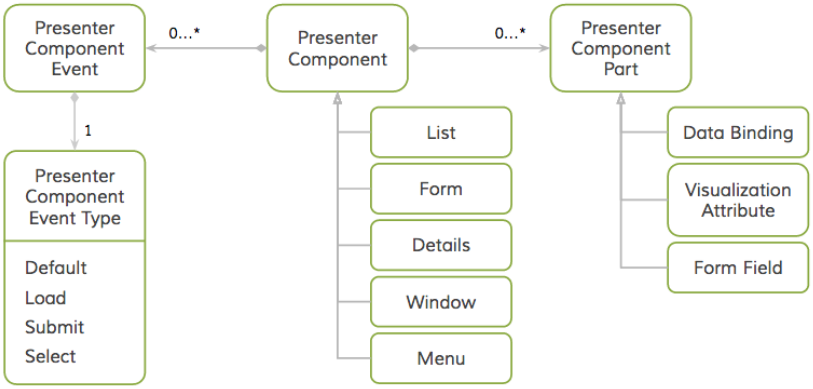

Figure 5.9: Target Architecture - Presenter's Hierarchy

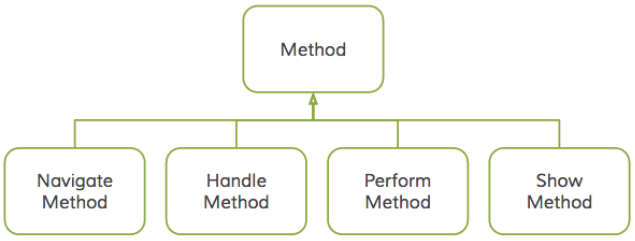

Figure 5.10: Target Architecture - Method's Hierarchy

The classes in this hierarchy can be of four different types:

- Navigate Methods, which implement the transition between views.
- Handle Methods, which pack the logic that is triggered after a user interaction.
- Perform Methods, which execute a piece of the application's logic.
- Show Methods, which are in charge of publishing data to the user through the GUI.

Figure 5.11 shows how these methods relate to the core entities. Together they provide a better picture of our design solution.

An Action Facade can have several Perform Methods, each of which has reference to an Action that implements a piece of the business logic. A Flow Controller may have several Navigate Methods that take the user interface to

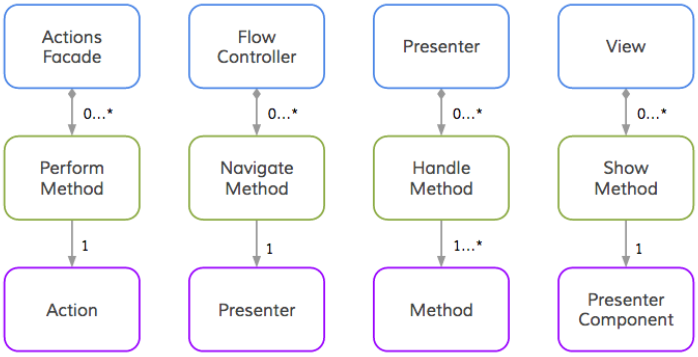

Figure 5.11: Target Architecture - Core entities and methods

a target Presenter. A Presenter has several Handle Methods each of which defines how the application should react to a user interaction. And each View has several Show Methods that publish the information contained by a particular Presenter Component.

To provide a better idea of how this architectural elements are translated into code, Figure 5.12 shows the code produced by the code generators for the Flow Controller, and Figure 5.13 shows the corresponding code for the Actions Facade.

```
#import <Foundation/Foundation.h>
#import "IFMLAppFlowController.h"
#import "IFMLAppFlowControllerImpl.h"
#import "Movie.h"
@class FavoriteMoviesWindowPresenter;
@class MovieDetailsWindowPresenter;
@class AddMovieWindowPresenter;

@interface FlowController:IFMLAppFlowControllerImpl <IFMLAppFlowController>

@property (strong, nonatomic) FavoriteMoviesWindowPresenter *favoriteMoviesWindowPresenter;
@property (strong, nonatomic) MovieDetailsWindowPresenter *movieDetailsWindowPresenter;
@property (strong, nonatomic) AddMovieWindowPresenter *addMovieWindowPresenter;

- (void)navigateToMovieDetailsWindowWithMovie:(Movie *)movie;
- (UIViewController<IFMLAppUI> *)initialViewController;
- (void)navigateToAddMovieWindowWithMovieList:(NSMutableArray *)movieList;
- (void)navigateToFavoriteMoviesWindowWithMovieList:(NSMutableArray *)movieList;

@end
```

Figure 5.12: Target Architecture - Flow Controller

This model driven perspective is meaningful because it provides us with a formal definition of our target. It allows us to describe through a set of entities and relationships between them, our final solution. This was also the case of IFML —describing the application through a set of entities and relationships— however, this intermediate representation makes the subsequent transformation process easier to grasp, and implement.

```
#import <Foundation/Foundation.h>
#import "IFMLAppActionFacade.h"

@interface ActionsFacade:NSObject<IFMLAppActionFacade>

- (void)performAddMovieWithMovieList:(NSMutableArray *)movieList
                               title:(NSString *)title
                                year:(NSNumber *)year;

- (void)performFetchMoviesWithMovieList:(NSMutableArray *)movieList;

@end
```

Figure 5.13: Target Architecture - Actions Facade

In the following sections we will illustrate how exactly the entities of IFML are mapped into the entities of these simpler metamodel. We will then cover the code generation process and will finally show some of the generated code.

5.3 Mapping

Perhaps the best reason for looking at the architecture of the generated apps as a metamodel, was how clear it made the mappings between the IFML entities and the proposed architecture. The effort introduced by the creation of the target metamodel produces its benefit at this mapping step, because it allowed us to describe in a simple way our design solution for the generated applications.

To illustrate how the mappings were achieved during the project, let us introduce some examples drawn from the "Movie Manager" App.

Once we are finished discussing the mappings, the code generation process would be easier to grasp. Because the application has already been described in terms that are closer to the final implementation.

5.3.1 IFML Model

The applications described with our target metamodel, have a root object of type App, that among other entities, references a single Actions Facade and a single Flow Controller. In contrast, an application described with the IFML metamodel has an IFML Model object as root object. The mapping we used in these case is shown in Figure 5.14.

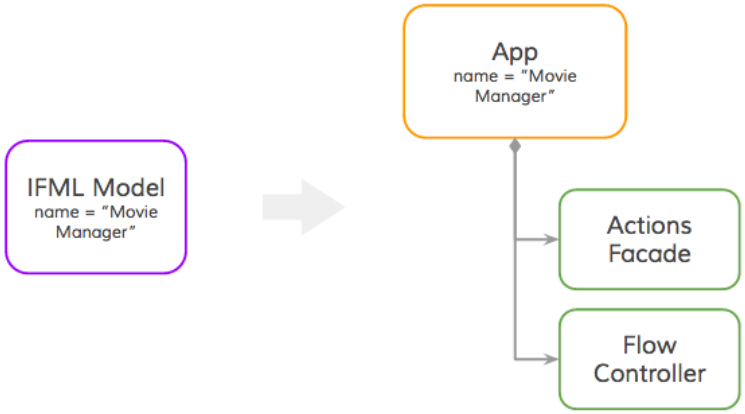

Figure 5.14: Mapping - IFML Model

5.3.2 Actions

Now lets look at the case of Actions. Actions in IFML have several attributes – Name, In Interaction Flows, several Parameters as well as Navigation Flows. Actions in our target metamodel are simpler. They only preserve the name attribute and the parameter list. However, in our target metamodel each action is referenced by one of the Perform Methods of the Actions Facade.

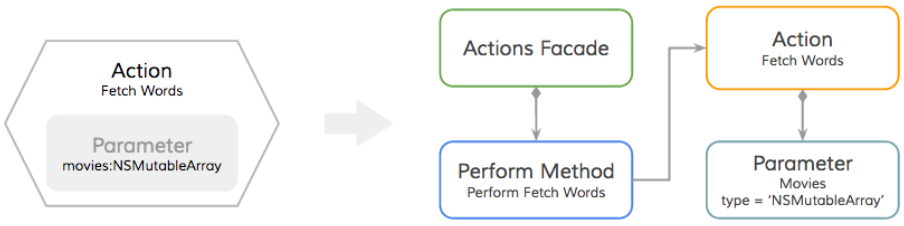

Figure 5.15: Mapping - Actions

In Figure 5.15, we present an example of how the "Fetch Movies" action of the "Movie Manager" app, was mapped into our target meta model. An action object and a Perform Method were created while the parameters of the IFML Action were mapped into equivalent parameter objects. Additionally, the Perform Method was associated to the Actions Facade.

This example starts to give us a better idea of the final outcome. That is,

at the end of the mapping effort, we would expect the Actions Facade to have several Perform Methods — one for each of the actions in our sample model. And our model will have one Action object for each IFML Action, preserving the parameters of the source object.

5.3.3 View Containers

At the front of any mobile application we have its screens. They describe the flow of information, and the different user interactions that can be captured in them. IFML uses a Window object to represent them. Windows act as containers of other components, and include the events that can be triggered in them.

The abstraction level of IFML, however, blurs out some implementation details. Specifically how responsibilities should be distributed among the application objects. Who should capture user interactions? who should display the data in the user interface?. In contrast, our target metamodel makes these details more explicit. A View entity is in charge of publishing data and capturing user interactions. A Presenter contains the logic to react to the captured interactions, and communicates the application data back and forth from the user interface.

To put this in context, lets look at Figure 5.16 that shows the mapping of an IFML window into our target metamodel.

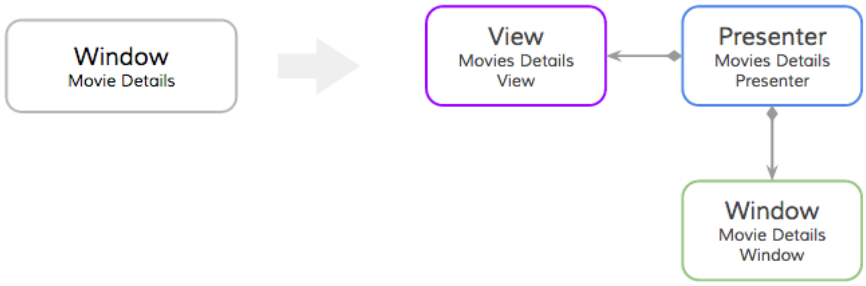

Figure 5.16: Mapping - Window

A window in IFML is turned into a triad of objects in the target metamodel: A View, a Presenter and a Window. The view represents the graphic dimension of the initial Window Object. While the Presenter connects the View with the rest of the application structure. Finally the Window object is preserved as one of the Presenter's Components.

While the presence of the View and Presenter can be sustained on the basis of the single responsibility principle that we discussed previously, the presence of the Window object requires an additional explanation. Though this explanation takes us back to the initial goals of our project.

One of the goals of our project was to reduce the time to compilation of the generated apps. That is, to create apps that had to be slightly modified before they could be compiled and executed for the first time. To achieve this level of simplicity we had to preserve some of the IFML infrastructure. And then use the introspection capabilities of Objective-C, to provide appropriated views. In other words, the Window entity , and similar classes preserved from the IFML metamodel, exist in the target metamodel so that we could provide default views and other default behaviors for the generated applications.

This also means that if we decided to compel IFML designers to provide the view layer of the generated apps, then many of the IFML classes that are preserved in the transformation would be no longer needed. This, however, was not the case of our project, and therefore we decided to keep the required entities.

5.3.4 View Components

If windows are relevant because they sit at front of mobile applications, then the content they display is just as important. IFML View Components describe the content that is displayed in each of the screens of the application. In the "Movie Manager" app, there is a View Component that shows a list of favorite movies; an other in charge of displaying detailed information about the movies in the list.

Like in the case of the View Containers, IFML abstracts away several implementation details, that become important when trying to implement the application in Objective-C. For this reason the IFML View Components are mapped into several entities in the target metamodel.

In Figure 5.18, a Details View Component adds a Show Method to the Presenter's View. This method holds a reference to the Details object and implies that the view should provide a mechanism to display the information gathered by it. The other elements and their respective mappings should look familiar since they follow our discussion about IFML Windows.

We don't provide examples for the List and Form elements since the rational

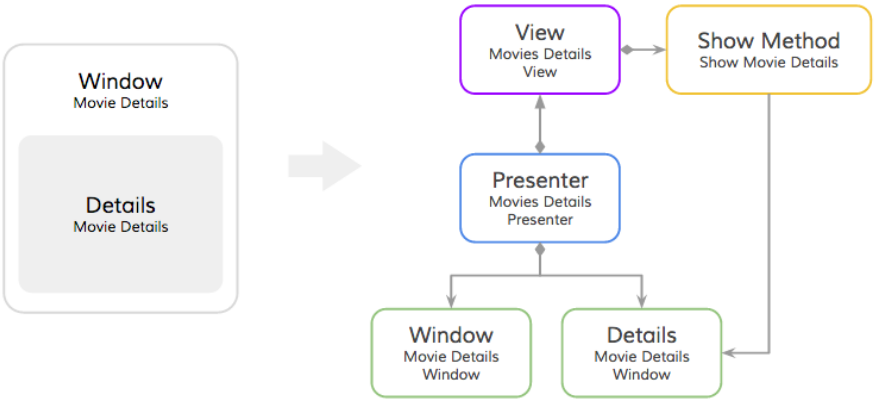

Figure 5.17: Mapping - Details

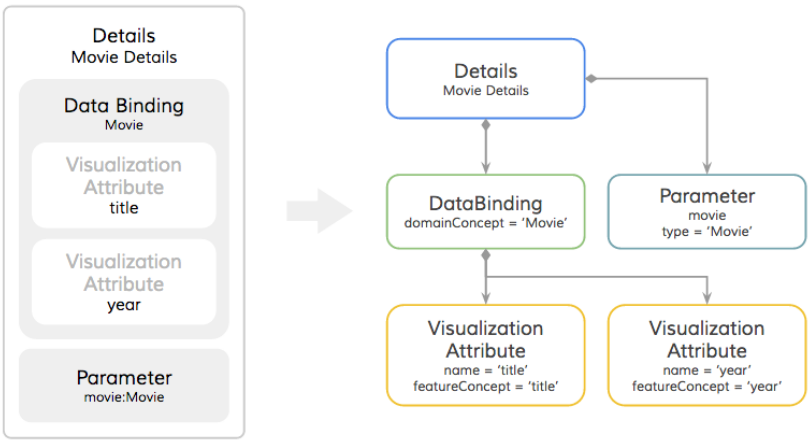

Figure 5.18: Mapping - Details

used to map them into the target architecture follows closely the strategy applied in the case of the Details object.

5.3.5 Interaction Flows

A second axis of any application is given by the navigation patterns it implements. That is, how each screen relates to the next one, and what are the dependencies between them. IFML uses Interaction Flows as the abstraction for these aspect of the mobile applications.

In our metamodel, Interaction Flows are mapped into Navigate Methods that are grouped in the Flow Controller. Like their IFML counter part,

each of these methods imply a change of screen towards the target, or an update in the data published by the current View. The Flow Controller is an entity that groups all the navigation paths of the application. It manages the dependencies between the different screens of the application. And more importantly, it provides a unified interface that evens out the communication patterns of the application.

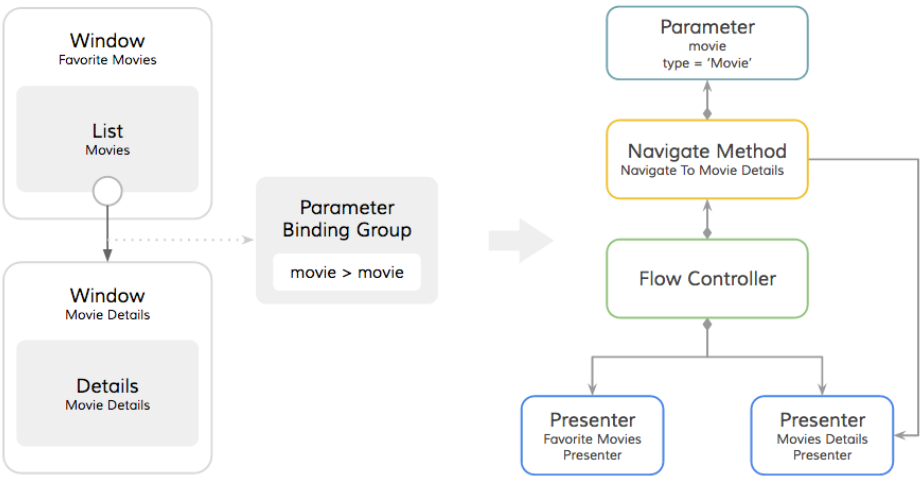

Figure 5.19: Mapping - Interaction Flow

In Figure 5.19 we can see how the Interaction Flow, that is triggered from the Select Method of the List Component, is transformed into a Navigate Method with the Movie Details Presenter as a target. Additionally we can see that the View Element Event of the List Component was mapped into a Handle Method included in the Presenter.

Up until now, we have discussed some of the the mappings of the main IFML entities in the context of the "Movie Manager" App. And we can see much better now the relations between IFML and the intermediate metamodel that we designed. In some cases the relations are simple, like in the case of Actions. Other entities, instead, require additional effort — as we saw in the case of the of Details and Windows.

In total 22 IFML entities were included in our mapping — not all of which were discussed in the document. As reference, we include a list of the IFML entities that were mapped.

- Action
- Data Flow
- Data Binding
- Navigation Flow

- Parameter
- Parameter Binding
- Parameter Binding Group
- UML Structural Feature
- UML Behavioral Feature
- UML Domain Concept
- Visualization Attribute
- Details
- Form
- List
- Menu
- OnLoadEvent
- OnSelectEvent
- OnSubmitEvent
- Selection Field
- Simple Field
- Slot
- Window

5.4 Static Library

Before developing the code generators we analyzed the code of the prototype app in order to understand how much code had to be dynamically generated. We started by filtering out the pieces of code that were model dependent — those that were deduced from the elements in the IFML model –, from the sections that would remain the same across the generated applications – or model independent. We then took the model independent sections, and packed them into a static library. As for the model dependent pieces, we singled them out as the ones that would have to be dynamically generated.

We called the Static Library, the IFML Kit. And we distributed it as a "private" cocoapod[1].

The classes in the library helped us reduce the code that had to be dynamically generated, and provided the proper vessel for the default views of the applications. As shown in Figure 5.20, the library was divided into three main packages, each of which is discussed in the following sections.

5.4.1 Simplified Meta-model Package

One of our goals during this part of the project was to provide a set of default views for the generated code. In this way, software designers could compile

[1] available at https://github.com/acerosalazar/ifml-ifmlkit.git

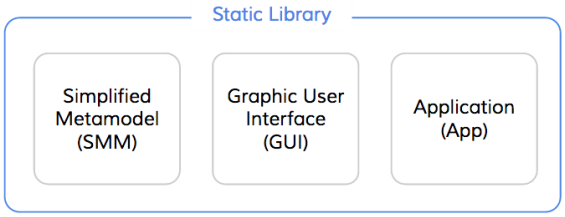

Figure 5.20: Static Library - Package Structure

and execute their applications soon after transforming their IFML models. To achieve this, we decided to preserve in the target metamodel a subset of the IFML entities, for which we provided an Objective-C implementation.

Following this analysis, the Simplified Meta-model Package or SMM, contained the Objective-C representation for a subset of IFML entities, like IFMLList, IFMLForm, IFMLDetails, IFMLWindow, IFMLDataBinding, IFML FormField and others, which not only represented their IFML equivalents, but also preserved some of their semantics and implemented a subset of their behaviours.

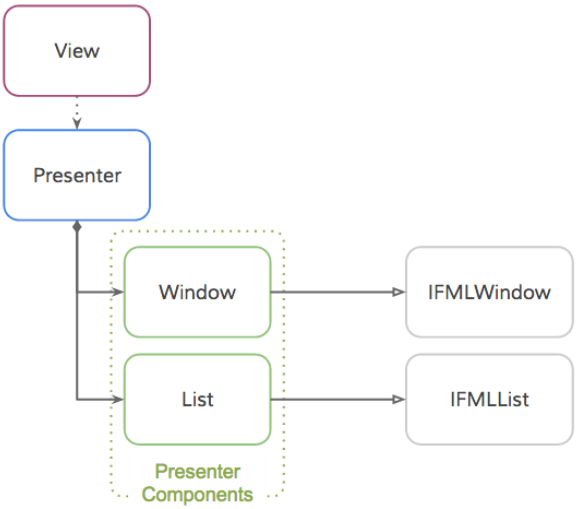

Figure 5.21: Static Library - SMM Classes

As shown in Figure 5.21, the Presenter Components, that we discussed in the "Target Architecture" section, extend the functionality of the classes in the SMM package. In consequence, the code that had to be produced by the code generators in this case was very simple – because having a set of base

classes with predefined behaviours, meant that part of the dynamic code would only be concerned with configuring these objects through inheritance or simple method calls.

Preserving these entities from the IFML metamodel was also very valuable for the classes in the GUI package, which could use them to deduce the widgets that needed to be shown in each of the screens of the application.

5.4.2 Graphic User Interface Package

The Graphic User Interface Package contained a set of default Views and View Controllers for the classes in the SMM Package. Classes like the IFMLListController, IFMLDetailsController and IFMLFormController, packed the logic needed to present the information hold by Lists, Details and Form components respectively. Moreover, while some of the Presenter Components like an IFMLList, could be presented using generic user interface widgets like a UITableView, others required the implementation of custom views that were also grouped into this package – for instance the IFMLSwitchCell, IFMLPickerCell, IFMLDateCell and others, were developed to provide default views for all the types FormFields supported by IFML.

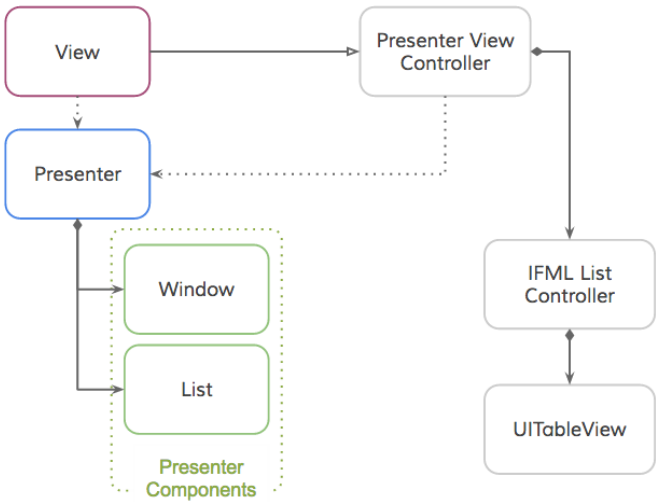

Figure 5.22: Static Library - PresenterViewController

To keep the logic of the code generators lean and simple, we created an additional class called the PresenterViewController. As shown in Figure 5.22, the View classes created by the code generators inherit from the PresenterViewController, who in turn holds a reference to the Presenter object associated

with the View. Using this relation, and the powerful run-time capabilities of Objective-C, the PresenterViewController can dynamically instantiate the Views and Controller needed to show the Components of the Presenter object. In the example shown in Figure 5.22, the PresenterViewController will introspect the properties of the associated Presenter object, and will instantiate an IFMLListController, who in turn will create and manage a UITableView to publish the data gathered by the List component.

What is interesting about this solution, is that all of this logic is wrapped by PresenterViewContoller and doesn't have to be handled by the code generators. In consequence, the code that needs to be generated for the Views of the application is minimal. Figure 5.23 shows an excerpt of the code generated for the View Controller of the example we discussed before.

```
1  //
2  // Created by IFMLGen on 2014/11/14 17:23:57
3  //
4
5  #import "MovieDetailsWindowViewController.h"
6
7  @implementation MovieDetailsWindowViewController
8
9  - (void)setMovieDetailsWindowPresenter:(MovieDetailsWindowPresenter *)
       movieDetailsWindowPresenter {
10
11     _movieDetailsWindowPresenter = movieDetailsWindowPresenter;
12       self.presenter = _movieDetailsWindowPresenter;
13 }
14
15 - (void)showMovieDetails:(NSArray *)values {
16
17     // Empty Implementation
18 }
19
20
21 @end
22
```

Figure 5.23: Static Library - Generated Code

The combination of classes in the SMM and the GUI packages, allowed us to achieve our goal of providing default views for the generated applications, while keeping the logic of the code generators lean and simple. The cost we had to pay for this simplicity was seen in the increased complexity of the classes in the GUI package, which had to resource to the run-time capabilities of Objective-C to provide appropriate views for the Components of a Presenter. This price was, however, mitigated by the fact that these classes could be implemented and tested independently using the development tools offered by Xcode.

5.4.3 Application Package

Up until now, all the classes and packages we have discussed, contribute to the creation of a set of default Views for the generated applications. The Static Library, however, contained other classes whose objective was very different. Such is the case of the classes in the Application Package.

Following the premise of reducing as much as possible the amount of code that had to be dynamically generated, the Application Package grouped the base classes for the main architectural components of the generated applications, thus classes like the IFMLAppFlowController, IFMLAppPresenter and the IFMLAppActionsFacade are included in it. In consequence, some of the classes created by the code generators inherit from them. Figure 5.24 shows a subset of the generated classes and their relation with the classes in the Application Package.

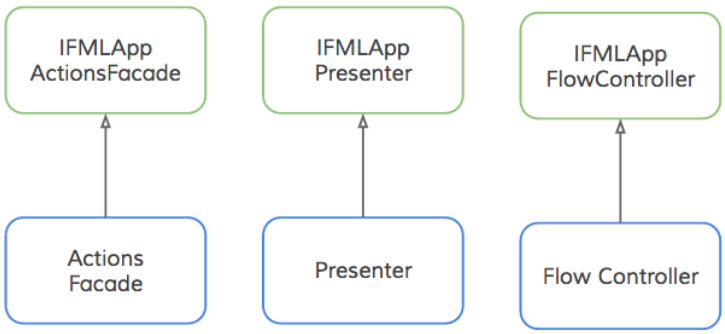

Figure 5.24: Static Library - Application Package

5.5 Code Generators

Before going into the details of the generation let's review how the code is organized in a typical iOS application.

The first thing to know is that Objective-C preserves several traits of its ancestor C. One of the more noticeable similarities is that it requires every class to declare a header and an implementation file. The header, contains the declaration of public methods and attributes — also known as properties in Objective-C. The implementation files instead, contain the implementation of the methods declared in the header.

As for the boilerplate code, every iOS application contains a Main file, an App Delegate, a Prefix header and a Property List file. The main file is seldom modified. It creates a unique UIApplication object and registers a delegate for it. The App Delegate instead, is the entry point for the custom code of the application. This is where the first screen of the user interface should be instantiated and where the code that triggers in response to changes in the application state and other notifications should be placed. The Prefix header provides compile time optimizations that need to be seldom modified. Finally, the Property List file contains meta-data about the application: supported orientations, required device capabilities and localization information among others, are the kind of aspects that are described in this file.

The rest of the files of a project are application dependent. As shown in Figure 5.25 the files of the generated applications were organized into five groups, following very closely the structure of our target architecture. Hence: Application, Views, Presenters, Actions, Models and Supporting Files were the names of folders used to group the generated files.

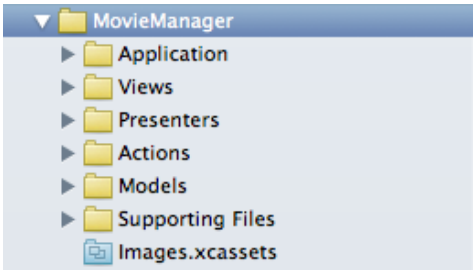

Figure 5.25: Code Generation - Folder Structure

While the groups represented by the Views, Presenters, Actions and Models contained classes with similar responsibilities, the Application group included classes like the Flow Controller and the Actions Facade, that many other classes in the architecture were dependent on. The Supporting Files group instead, was home to the Main, Prefix and the Property List file of the application.

5.5.1 Generators

The M2T transformation was done using Xtend. A dialect of java that provides, among other features, very powerful templates. The generation effort was divided into 2 types of files: boilerplate and dynamic.

The boilerplate files corresponded to classes like the Application's Main, shown in Figure 5.26, and the App Delegate whose contents remained unchanged across the applications.

```
4  #import <UIKit/UIKit.h>
5  #import "AppDelegate.h"
6
7  int main(int argc, char * argv[])
8  {
9      @autoreleasepool {
10         return UIApplicationMain(argc, argv, nil, NSStringFromClass([AppDelegate class]));
11     }
12 }
13
```

Figure 5.26: Code Generation - Main file

The dynamic package instead, dealt with the model dependent elements of the application. In this case, we used one generator file for each core entity in the target metamodel.

- ActionsFacadeGenerator
- FlowControllerGenerator
- ActionsGenerator
- PresentersGenerator
- PresentersComponentGenerator
- ModelGenerator
- ViewGenerator

Additionally there were three generators that handled the creation of files whose importance is better appreciated from the implementation perspective. The ViewFactoryGenerator, the PresenterUIGenerator and the AppDependenciesGenerator. The first two, produced components that make the view layer of the generated apps decoupled and therefore easy to replace. In contrast, the AppDependenciesGenerator was in charge of creating the AppDependencies class whose role consisted in instantiating and configuring the objects that assemble the app. The contents of these files can be considered implementation details, and therefore, we do not discuss them any further in the document.

5.5.2 Templates

Each of the generators used one template for the header, and one template for the implementation file of the classes they had to generate. Moreover, each template was populated through a simple sequence: first add the imports, then the properties and finally the methods.

Figure 5.27: Code Generation - Template Structure

Since the header file contained the method signatures, and the implementation file their corresponding implementation, the generation of methods deserved a special management. Specifically, we used a key-value pair data structure in which the method signatures were used as keys, while the implementations were stored as their corresponding values. This strategy allowed us to keep the contents of both files synchronized while removing the hazards caused by duplicated method declarations or omitted implementations. In contrast, the generation of the imports and properties was achieved through simple arrays.

To have a better idea of the templates used during the project, lets have a look a the template used to create actions. Figure 5.28 shows a snippet with two methods two methods: "generateHeader" and "generateImplementation". The first one, declares the public imports of the class, and then proceeds to generate its properties based on the parameters associated with the IFML Action. The second one is arguably simpler; it imports the generated header file and then invokes the extension method "implementations" on the methods map to print successively the signatures and implementations of the action's methods.

This basic structure was preserved among all the generators. Although, in some cases —like the FlowControllerGenerator Figure 5.29 — the logic used to populate the templates was more complicated.

```
override generateHeader(IFMLAction it) {

        '''
                #import <Foundation/Foundation.h>
                #import "IFMLAction.h"

                @interface «it.name.toFirstUpper» : NSObject <IFMLAction>

                «it.parameters.map['''@property (strong, nonatomic) «type.name» *«name»;''']».join("\n")»

                @end
        '''
}

override generateImplementation(IFMLAction it) {
        '''
                #import "«it.name.toFirstUpper».h"

                @implementation «it.name.toFirstUpper»

                «methods.implementations()»

                @end
        '''
}
```

Figure 5.28: Code Generation - Actions Templates

```
override protected prepareGeneration(Iterable<ViewElement> it) {

    // Generate Imports
    imports = it.map [ v |
        v.inInteractionFlows.map [ i |
            i.parameterBindingGroup.parameterBindings.map [ pB |
                if (!(pB.targetParameter.type instanceof PrimitiveType)) {
                    pB.targetParameter.type.name
                }
            ]
        ].flatten
    ].flatten.filter[!isNullOrEmpty]

    // Generate Methods
    val defaultWindow = it.filter(typeof(IFMLWindow))?.filter[isIsDefault]?.head()
    if (defaultWindow instanceof IFMLWindow) {
        methods.put(methodInitialViewController(defaultWindow).key, methodInitialViewController(defaultWindow).value)
    }

    it.forEach[inInteractionFlows.forEach [ i | methods.put(methodNavigateToPresenter(i).key, methodNavigateToPresenter(i)
}
override protected generateHeader(Iterable<ViewElement> it) {

    var windows = it.filter(typeof(IFMLWindow))

    '''
        #import <Foundation/Foundation.h>
        #import "IFMLAppFlowController.h"
        #import "IFMLAppFlowControllerImpl.h"
        «imports.map[i|'''#import "«i».h"'''].join("\n")»
        «windows.map[w|'''@class «w.name.toFirstUpper»Presenter'''].join(";\n")»;

        @interface «fileName»:IFMLAppFlowControllerImpl <IFMLAppFlowController>

        «windows.map[w|'''@property (strong, nonatomic) «w.name.toFirstUpper»Presenter *«w.name»Presenter'''].join(";\n")»

        «methods.signatures»

        @end
    '''
```

Figure 5.29: Code Generation - Flow Controller Template

5.5.3 Generated Code

To have a better understanding of how the application's code is generated, lets look at Figure 5.30 that shows the classes that are generated from the Movie Details screen of the "Movie Manager" app.

Figure 5.30: Code Generation - Generated Code

On the top left we have the IFML diagram used to describe the selected screen of the application. On top right, we have the same app excerpt but described using our target metamodel. Finally, the figure shows that the M2T transformation for this case would generate four classes, namely: MovieDetailsPresenter, MovieDetailsView, MovieDetailsWindow and MovieDetails.

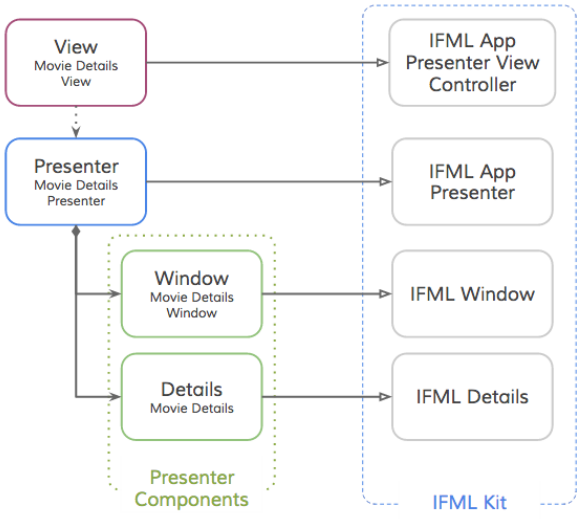

Figure 5.31: Code Generation - Inherintance from the Static Library

Notice, how in Figure 5.31 becomes clear that all the generated classes inherit from the classes contained in the Static Library . Using this mecha-

nism, the generated code was greatly reduced and simplified, as we can see in Figures 5.32 and 5.33, which contain the generated header and implementation files for the Movie Details Presenter class.

```
#import <Foundation/Foundation.h>
#import "IFMLAppPresenter.h"
#import "IFMLAppPresenterImpl.h"
#import "FlowController.h"
#import "ActionsFacade.h"
#import "MovieDetailsWindowPresenterUI.h"
#import "MovieDetailsWindow.h"
#import "MovieDetails.h"

@interface MovieDetailsWindowPresenter : IFMLAppPresenterImpl <IFMLAppPresenter>

@property (strong, nonatomic) FlowController *flowController;
@property (strong, nonatomic) ActionsFacade *actionsFacade;
@property (strong, nonatomic) UIViewController<MovieDetailsWindowPresenterUI> *userInterface;
@property (strong, nonatomic) MovieDetailsWindow *movieDetailsWindow;
@property (strong, nonatomic) MovieDetails * movieDetails;

- (void)didLoadView;
- (void)updateView;

@end

```

Figure 5.32: Code Generation - Movie Details Presenter Header

```
#import "MovieDetailsWindowPresenter.h"

@implementation MovieDetailsWindowPresenter

- (void)didLoadView {

}

- (void)updateView {

    [self.userInterface showMovieDetails:self.movieDetails.viewComponentParts];
}


@end
```

Figure 5.33: Code Generation - Movie Details Presenter Implementation

5.6 Integration

Our project went beyond the mere generation of source code files. It also provided support for integrating the produced files into Xcode — Apple's flagship IDE. We achieved this integration through a set of scripts that allowed us create the project file, and to subsequently download and link the static library to the project.

The Project Generation script was based on a ruby gem called Xcodeproj[2]. The script received as input the location of the source files produced by the

[2] http://www.rubydoc.info/gems/xcodeproj

code generators and it produced as output an Xcode project file. The generated project had a unique compilation target and organized the code into the six groups that we discussed previously — Views, Presenters, Models, Actions, Application and Supporting Files. Additionally, using the capabilities offered by the Xcodeproj gem, we added a collection of common build options for the project.

In contrast, the second script we developed was used to download and link the static library to the generated project. To do this, we distributed the static library using cocoapods — a popular dependency manager for iOS and OSX development.

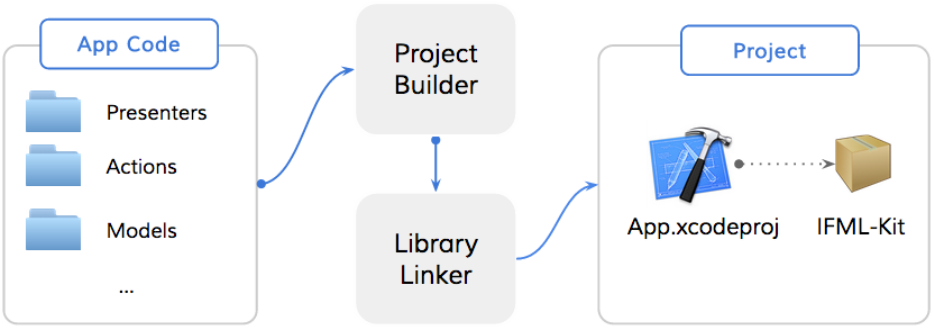

Figure 5.34: Integration - Automation Scripts

After executing the previous scripts, we obtained an Xcode project that could be used not only to visualize and modify generated code but also to compile and execute it.

These tools had a positive effect on the productivity of the project. They allowed us to quickly test and modify the generated code using Xcode without having to perform this process manually. The tools also provided a robust solution for the management of static files. Similar projects like MD2 [22], had solved this issue by creating additional modules into the transformation tools whose responsibility was the management of static files. This solution implied, however, that the transformation tool had to be distributed in tandem with the static library, which made the update cycles of the static library more cumbersome.

Finally, these tools managed to reduce the friction caused by the change of technical spaces, improving in this way the experience of the users of the transformation tools. With the generation tools in place, IFML designers could use the transformation tool as an eclipse plug-in right next to the model editor, while native iOS developers could use Xcode to update, com-

pile and execute the generated code. This way every actor gets to work using their everyday tools, instead of having to learn how to use something new.

5.7 Packaging

The tools discussed previously were organized into several software packages. A good way to visualize them is in a pipeline whereby, the input — an IFML model — is transformed several times until reaching the desired output — a compilable Xcode project.

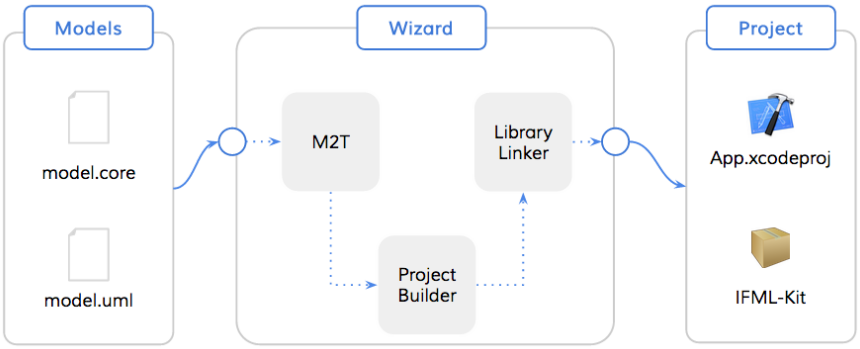

Figure 5.35: Packaging - Generation Wizard

As shown in Figure 5.35, several tools are involved in the generation process. First, the transformation tool takes an IFML model and produces from it a set of equivalent Objective-C files. Then, the Project Builder script creates an Xcode project file. Finally, the Library Linker communicates with cocoapods and, leveraging the command line tools of the dependency manager, adds the Static Library to the compilation path of the project.

While on its own, the proposed pipeline was very powerful, it suffered from usability issues. For instance the intermediate steps between the generation steps had to be done manually, and the parameters for the scripts needed to be provided in the right order. To solve this issue, we developed a generation wizard. This application sat on top of the generation tools and provided a GUI to insert the inputs and control the generation process. Although the application was developed as an OSX app, the automation script used to connect the generation tools is a simple bash that can be executed in other operating systems.

In summary, the output of the project was packaged into six different tools, from which the the Code Generator, the Static Library and the Project Builder work as standalone apps. In contrast, the Library Linker and the Generation Wizard are dependent on the previous three and were added as a way of making the generation process more convenient. The links to the repositories that contain the source code of all these tools can be found in this site: http://acerosalazar.github.io/ifml-gtk/.

Chapter 6

Conclusion

6.1 Results

At the end of the project we managed to reach our objectives. Specifically, to enhance the portability of IFML by developing a code generator for iOS, and to broaden the modeling capabilities of the language, by adding a set of extensions to its metamodel.

In the first case, we developed a toolset composed of a Model to Text transformation tool, a Project Builder and a Library Linker scripts. Using these tools, we could produce a compilable Xcode project, starting with the IFML and UML models of an application. To make things even simpler, we provided a graphic wizard that made the transformation process a matter of a couple of clicks.

As for the language extensions, we proposed 90 new entities grouped into three different packages. The General Extensions Package, which is composed of entities whose scope goes beyond mobile environments — like in the case of the Image Attribute and the Map View. The Mobile Extensions Package, whose classes are meant to broaden the types of user interactions that can be modeled with IFML – Pan, Pinch, Swipe, Tap, Touch and others —, as well as the type of events that can be captured using the sensors included in modern mobile devices – Location Events, Orientation Events, Accelerometer Events, Gyroscope Events, among others. Finally the Private Extensions Package, was home to the entities that extended the Context classes, which, as of this writing, is not allowed outside a private scope by the current version of the IFML standard [21].

In addition to these results, we believe that the methodology used to conduct the research was an important byproduct of the project. This is specially relevant in the case of IFML, since there are several research teams working in projects with similar goals, for whom this insight could be of great value.

There is, however, room for improvement.

6.2 Critical Analysis

The current version of the code generators supports a subset of 22 entities from the total amount of classes defined by the IFML standard [21]. Leaving uncovered important entities like the UML Behaviours, and a great portion of the Expressions hierarchy. Moreover, data persistence, which is an important aspect in modern mobile applications, was narrowly addressed in our project and therefore deserves a deeper look.

As for the language extensions, the proposed entities need to be further discussed and evaluated, to understand their practical value. Then, a suitable graphic syntax should be proposed, so that they could be added into the IFML Model Editor. Allowing code generators to provide mappings for them, and software engineers to start using them in the models of their applications.

6.3 Future Work

Finally, to keep our project within the scope and time constraints, we had to steer clear from some opportunities that should be, however, explored in future efforts.

The first opportunity is provided by Swift – Apple's new programming language. Its simpler syntax, and stronger type system may lead to several improvements in the code generators, and specially to simpler templates and a more straightforward generation logic.

The second opportunity lies in the possibilities offered by the Model Driven world. In particular, we consider that an intermediate metamodel that brings the concepts of IFML closer to a traditional MVC architecture, should be designed. This metamodel would formalize the architecture of the generated applications, and since it would be defined at a platform independent

level, existing code generators for IFML could use it as the starting point for the model to text transformation process. This strategy would have positive implications in several fronts. On one hand, cross-platform developers will have to be familiar with a unique architecture, that will be preserved across the code generated for the different platforms. On the other hand, the developers in charge of the code generators would be able to collaborate in the creation of more robust generation strategies. And finally, external observers and other interested people, would be able to understand at a higher level of abstraction, the way code generation is achieved in IFML.

Our final words, are dedicated to the IFML Model Editor.

During the first stages of our project we had access to an early version of the editor, which we used to visualize and modify the IFML model of our prototype application. While this early version had some rough edges, the tool was very important in achieving our final results, and helped us in the creation of well formed models that could be used by our code generators. We believe that, once finished, this tool will be of great value for software designers, and that the development team behind it should continue improving its features and broadening in its capabilities. A good place to start this improvements, would be adding support for the introduction of private extensions into the editor, which will allow for a more dynamic evolution of the modeling language.

Appendix A

General Extensions Package

Application Lifecycle Event

Abstract: No
Generalization: Mobile System Event

Application Lifecycle Events model each of the notifications triggered by operating system to inform applications about changes in their life-cycles. For instance, operating systems can inform an application when it has been launched, or when it is about to be terminated. Similarly, an application can receive updates on whether it is executing in the foreground or in the background.

Expanded Selection Field

Abstract: No
Generalization: Selection Field

Expanded Selection Fields can be used to indicate that all the options offered by a Selection Field should be displayed to the user at once rather than one at the time. In contrast with regular Selection Fields, code generators could map Expanded Selection Fields into more suitable and specific GUI components like button groups instead of using more generic components like a combo-box.

Enumeration View Lifecycle Event Type

The values of this enumeration describe the notifications that are typically raised by the system after there has been an update on the state of a particular view.

Literals

- View Did Appear: This type of event is triggered after a view has been allocated memory and has been successfully rendered to the user [13].
- View Did Disappear: This event is triggered after a view has been removed from the screen, but not necessarily so from main memory [13].
- View Did Load: This event is triggered after a view has been allocated memory. This is the earliest point at which a view or its corresponding controller may execute additional initialization routines [13].
- View Will Appear: This event is triggered when a view is ready to be rendered to the user. While the View Did Load event may be triggered only once (after a view and its corresponding controller have been allocated memory), the View Will Appear event should be triggered every time the view is ready to be shown [13].
- View Will Disappear: This is event is triggered just before a view is removed from the screen [13].

Enumeration Application Lifecycle Event Type

The values of this enumeration describe the notifications that are raised by the system after there has been an update on the application state.

Literals

- App Did Launch: This event is raised right after the user launches the application [17].
- App Will Become Active: This event is triggered just before an application is switched to foreground mode and will become the active application. In contrast with the background mode, applications in the foreground can receive user interactions [17].

- App Did Become Active: This event is triggered right after an application has changed its state from inactive to active, and its being executed in the foreground [17].
- App Will Hide: This event is triggered just before an application is minimized. Since mobile nor web applications are susceptible to this kind of transitions, the usage of this event is restricted to Desktop applications.
- App Did Hide: This event is triggered right after an application has been minimized. Since mobile nor web applications are susceptible to this kind of transitions, the usage of this event is restricted to Desktop applications.
- App Will Unhide: This event is triggered right before an application is maximized. Since mobile nor web applications are susceptible to this kind of transitions, the usage of this event is restricted to Desktop applications.
- App Did Unhide: This event is triggered right after an application has been maximized. Since mobile nor web applications are susceptible to this kind of transitions, the usage of this event is restricted to Desktop applications.
- App Will Resign Active: This event is raised just before an application transitions form an active state to an inactive state. Such transition happens every time a user switches to another application [17].
- App Did Resign Active: This event is triggered right after a user has just switched focus to a different application[17].
- App Will Terminate: This events is triggered before an application is terminated and removed from main memory [17].

Image Attribute

Abstract: No
Generalization: Visualization Attribute

Image Attributes allow IFML designers to explicitly indicate that a particular attribute should be rendered as an image. In contrast with Image Attributes, generic Visual Attributes would rely on the capabilities of the code generators to infer the type of data that needs to be shown in the UI based on the associated Data Bindings, which in turn may not be so accurate.

Map Annotation

Abstract: No
Generalization: View Component

Map annotations describe points of interest in a map, and can naturally be included only within a Map View.

Constraints

- Map Annotations must have a Data Binding that provides them with a geographic coordinate that indicates the exact location of the map where they need to be rendered. Optionally, the Data Binding could also include the additional information that should be shown to the user in these points.
- They can only be added into Map Views.

Map Overlay

Abstract: No
Generalization: View Component

Map Overlays represent poly lines, shapes and other graphical elements that can be shown on top of a map. IFML designers could use map overlays to model maps that convey information in a more convenient way – e.g using poly lines to describe routes or shapes to identify areas of interest for the user.

Constraints

- Map Overlays require a Data Binding to provide them with the geographical information and the description of the shape, if any, used to represent them.
- They can only be added into Map Views.

Map View

Abstract: No
Generalization: View Container

Map Views are off-the shelf interactive map visualizations that are frequently used in mobile, web and desktop applications. They can only contain Map Annotations and Map Overlays which, in turn, are used to display points of interest and map related information in a meaningful way.

Constraints

- Map views can contain Map Annotation and Map Overlay components only.

Modal Menu

Abstract: No
Generalization: Menu

A Modal Menu represents a set of commands that can be dismissed. Similar to Menus, the commands available within a Modal Menu can be modeled by means of View Element Events, each of which may trigger a different business logic action.

Constraints

This type of View Containers cannot contain any View Elements.

Multi Line Field

Abstract: No
Generalization: Simple Field

Multiline Fields allow IFML designers to indicate explicitly that a particular field would be used for multiple line inputs. Such refinement will allow code generators to make better decisions and apply more meaningful mappings. In this way, Multiline Fields could be translated into more suitable GUI components like text areas rather than into, the more general, text fields.

GL View

Abstract: No
Generalization: View Component

GL Views model interface components that are rendered using advanced graphics libraries like OpenGL or WebGL.

Range Selection Field

Abstract: No
Generalization: Selection Field

Range Selection Fields are selection fields whose possible values lay within are a range. Users can pick any continuous or discrete value as defined by a step. An IFML designer may use this view components to represent frequent GUI widgets like sliders and value steppers.

Attributes

- minValue [float] : the lower bound of the value range.
- maxValue [float] : the upper bound of the value range.
- stepValue [int]: the difference between two adjacent values. If the value of this attribute is NULL, the user will be allowed to select from a set of continuous values, thus without a defined step.

Constraints

- The minValue must be greater than the maxValue.
- The stepValue must be an integer greater than zero.

View Lifecycle Event

Abstract: No
Generalization: Mobile System Event

View Lifecycle Events are triggered after a change in the state of a view's lifecycle. Such events can be used by IFML designers in order to model responsive and usable applications that react appropriately to such changes, and blend nicely with the resource management strategies put in place by the operating systems.

Attributes

- Event Type [View Lifecycle Event Type]: The specific type of lifecycle event triggered.

Relevant iOS Resources

General Extensions Package Entity	iOS Resources
Application Lifecycle Event	UIApplicationDelegate
Expanded Selection Field	UISegmentedControl, UITableViewContoller
Enumeration View Lifecycle Event Type	UIViewController
Enumeration Application Lifecycle Event Type	UIAppDelegate
Image Attribute	UIImageView
Map Annotation	MKAnnotation
Map Overlay	MKOverlay
Map View	MKMapView
Modal Menu	UIAlertController
Multiline Field	UITextView
GL View	GLKView
Range Selection Field	UISlider, UIStepper
View Lifecycle Event	UIViewController

Appendix B

Mobile Extensions Package

Accelerometer Event

Abstract: No
Generalization: Sensor Event

This event will be fired every time there is a change on the device acceleration along any of the spatial axis. This event may be used, for instance, in order to allow users to interact with the application by physically moving their device. Since acceleration events can be considered as global events (meaning that they are perceived and affect equally all the elements of the application), they are categorized under the Catching Events entity. Because of it, they can be associated with Activation Expressions, Interaction Flow Expressions and Navigation Expression.

Ad Banner View

Abstract: No
Generalization: Mobile View Component

The Ad Banner View Component is used to display banner advertisements as supported in the SDKs of several mobile platforms [16] [20]. In general IFML designers will simply include these components in their models in order to allow Code Generators to produce the appropriate source code. Thus, without adding any events or navigation flows that affect them.

Battery Event

Abstract: No
Generalization: Resource Event

Battery events will be raised when any changes related with the device battery level or battery status occur. Examples of such changes could be an update on the remaining battery power or a notification informing that the device has been plugged to the electrical outlet.

Bluetooth Event

Abstract: Yes
Generalization: Resource Event

This kind of events will be raised upon device discovery and updates on connection status as well as after receiving / completing requests related with service publishing and consumption. Since mobile devices can behave as bluetooth peripherals as well as a bluetooth centrals, further extensions to provide a greater level of granularity are encouraged.

Enumeration Swipe Direction

Each of the directions in which a swipe gestures can be identified.

Literals

- Right: The user taps the screen and dragging his finger from left to right.
- Left: The user taps the screen dragging his finger from right to left.
- Up: The user taps the screen dragging his finger in direction down - up.
- Down: The user taps the screen dragging his finger in direction up - down.

Gyroscope Event

Abstract: No
Generalization: Sensor Event

Gyroscope Events are raised every time there is a change on the device's rotation rate along any of the three spatial axis. IFML designers may use this

kind of events to allow users interact with their applications by physically moving and rotating their device.

Location Event

Abstract: No
Generalization: Sensor Event

Location Events are triggered every time there is a change on the geographic position of the user. IFML designers may use Location Events to model applications that offer navigation services, maps or that simply rely on user positioning to deliver certain functionalities.

Long Press Event

Abstract: No
Generalization: Touch Event

Long Press Events are raised when the user touches a particular view leaving its finger on the screen for certain amount of time. This type of events are frequently used to display contextual menus, or activate alternative actions – e.g item sorting.

Attributes

- pressDuration [Float]: a value in seconds that specifies the period of time that a user should press a view before a Long Press Event is fired.
- fingersCount [Int]: the number of simultaneous touches that the user is expected to perform so the event can be effectively captured.

Magnetometer Event

Abstract: No
Generalization: Sensor Event

This type of events are triggered by changes on the orientation of the device measured against the magnetic or the true north. This type of events are typically used in apps that offer navigation services to identify the user's heading direction.

Memory Event

Abstract: No
Generalization: Resource Event

Memory events are usually warning notifications triggered by the system to inform mobile apps about the scarcity of memory resources and allow them to reduce their memory footprint if possible [17].

Motion Event

Abstract: No
Generalization: Sensor Event

Motion events model changes on the rotation, orientation or linear velocity of the device. Motion events are a general classification for Magnetometer Events, Accelerometer Events and Gyroscope Events.

Mobile Resource Event

Abstract: No
Generalization: Mobile System Event

Mobile Resource Events are occurrences related with resources managed by the system. This entity provides a general category for Memory Events, Bluetooth Events and Battery Events.

Mobile Sensor Event

Abstract: No
Generalization: Mobile System Event

Mobile Sensor Events model notifications triggered by the sensors included in modern mobile devices. This entity provides a general classification for Location Events, Proximity Events, Motion Events, Magnetometer Events, Accelerometer Events, Gyroscope Events and Shake Events.

Mobile System Event

Abstract: Yes
Generalization: System Event

A Mobile System Event is an Event produced by mobile operating systems, which is related with tasks and resources that they manage. This class pro-

vides a general classification for Mobile Sensor Events and Mobile Resource Events.

Mobile View Component

Abstract: Yes
Generalization: View Component

Mobile View Components are View Components that are somewhat unique to mobile platforms. This abstract class provides a generalization for other classes like Web Views, Search Views and Ad Banner Views.

Mobile View Container

Abstract: Yes
Generalization: View Container

A Mobile View Container is an abstract entity provides a generalization for container classes that are unique to mobile platforms like the Screen container.

Mobile View Element Event

Abstract: Yes
Generalization: View Element Event

Mobile View Element Events are Events that are somewhat unique to views in mobile platforms. This abstract class provides a generalization for classes like Touch Events.

Orientation Event

Abstract: No
Generalization: Motion Event

Orientation Events are triggered by changes in the orientation of a mobile device. A typical scenario in which this event will be fired is given by a user that rotates his device to use it in landscape mode. IFML designers could consider this kind of events in their models to appropriately adapt the UI layout according to the current device orientation.

Pan Event

Abstract: No
Generalization: Touch Event

A Pan Event occurs when a user touches the device screen and swipes in any direction. In contrast with Swipe Events, Pan events are triggered regardless of the direction of the swipe, while Swipe Events are only raised if the performed gesture happened in the registered direction. In this way, every Swipe Event could be also considered a Pan Event, while the opposite is not always true.

Attributes

- fingersCount [Int]: the number of simultaneous touches that the user is expected to perform so the event can be effectively captured.

Pinch Event

Abstract: No
Generalization: Touch Event

A Pinch Event denotes a gesture that requires at least two simultaneous touches, following either of the following descriptions: the fingers start separated and move towards each other; or, they start together and move away from each other. This type of events are typically used in mobile applications to allow zooming in and out from a particular view.

Proximity Event

Abstract: No
Generalization: Sensor Event

Proximity Events are triggered every time the user is close to the proximity sensor of the device. This type of events is frequently used in applications that allow users make and receive phone calls which, to avoid unintended interactions, disable the touch events of the screen when the user's ear is perceived to be close to the device.

Rotate Event

Abstract: No
Generalization: Touch Event

Rotate Events are triggered in response to a tactile gesture in which the user touches the screen with two fingers dragging them in opposite directions describing a circular path.

Search View

Abstract: No
Generalization: Mobile View Component

A Search View is a special kind of view component that allows users to introduce a query term and perform a search with it. While a regular Form component with a Simple Field could achieve a similar result, very often mobile platforms offer ready to use search components that provide additional functionalities – e.g. submitting the query to search provider, or a carefully crafted interface that improves usability.

Screen

Abstract: No
Generalization: Mobile View Container

The Screen entity is aimed to represent the basic container unit of a mobile application. Typically a mobile app will have several screens, that can be shown modally or modeless as well as belong to a navigation stack.

Attributes

- isModal: If true, this attribute indicates that the screen should be shown modally. By default this attribute should be false, declaring that the screen should be shown modeless.
- hasNavBar: If true, this attribute will indicate that the screen is part of a navigation stack and therefore should provide a mechanism to navigate back (i.e. removing the current screen out of the stack)

Shake Event

Abstract: No
Generalization: Motion Event

A Shake Event is triggered when a user physically move his device from side to side a couple of times. This event is captured using the accelerometer of

the device, which implies that alternative interaction strategies will need to be provided in case the device in use does not have an accelerometer.

Swipe Event

Abstract: No
Generalization: Touch Event

A Swipe Event is triggered by a gesture in which the user touches the screen with one or more fingers and slides them together towards a single direction.

Attributes

- swipeDirection[Swipe Direction]: the direction in which the user should swipe the device screen.

Tap Event

Abstract: No
Generalization: Touch Event

A Tap Event is triggered every time a user touches the screen of the device using one or more fingers. This is arguably the simplest and most common Touch Event used in mobile applications.

Attributes

- fingersCount [Int]: the number of simultaneous touches that the user is expected to perform so the event can be effectively captured.

Touch Event

Abstract: No
Generalization: Mobile View Element Event

A Touch Event encompasses a family of gestured based interactions that are enabled by the tactile capabilities of the screen of most modern mobile devices. This class provides a generalization for other classes like Tap Event, Swipe Event, Pan Event, Long Press Event, Pinch Event and Rotate Event.

Web View

Abstract: No
Generalization: Mobile View Container

Web Views represent a special kind of containers that are able to render and navigate through web content – based on HTML, CSS and JavaScript.

Relevant iOS Resources

Mobile Extensions Package	iOS Resources
Accelerometer Event	CMMotionManager (startAccelerometerUpdatesToQueue: withHandler:)
Ad Banner View	ADBannerView
Battery Event	UIDevice (UIDeviceBatteryLevelDidChangeNotification, UIDeviceBatteryStateDidChangeNotification)
Bluetooth Event	CBCentralManagerDelegate, CBPeripheralDelegate
Enumeration Swipe Direction	UISwipeGestureRecognizerDirection
Gyroscope Event	CMMotionManager (startGyroUpdatesToQueue:withHandler:)
Location Event	CLLocationManager (startMonitoringSignificantLocationChanges, locationManager:didUpdateLocations:), CLLocationManagerDelegate (stopMonitoringSignificantLocationChanges)
Long Press Event	UILongPressGestureRecognizer (initWithTarget:action:)
Magnetometer Event	CMMotionManager (startMagnetometerUpdatesToQueue: withHandler:)
Memory Event	UIViewController (didReceiveMemoryWarning)
Motion Event	CMMotionManager (startDeviceMotionUpdatesToQueue: withHandler:)
Mobile Resource Event	N/A
Mobile Sensor Event	N/A

Mobile System Event	N/A
Mobile View Component	N/A
Mobile View Container	N/A
Mobile View Element Event	N/A
Orientation Event	UIDevice (beginGeneratingDeviceOrientationNotifications, endGeneratingDeviceOrientationNotifications), UIDeviceOrientationDidChangeNotification
Pan Event	UIPanGestureRecognizer (initWithTarget:action:)
Pinch Event	UIPinchGestureRecognizer (initWithTarget:action:)
Proximity Event	UIDevice, UIDeviceProximityStateDidChangeNotification
Rotate Event	UIRotationGestureRecognizer (initWithTarget:action:)
Search View	UISearchBar, UISearchDisplayController
Screen	UIViewController, UINavigationController
Shake Event	UIResponder (canBecomeFirstResponder, becomeFirstResponder, motionBeganWithEvent:, motionEndedWithEvent:), UIEventSubtypeMotionShake
Swipe Event	UISwipeGestureRecognizer (initWithTarget:action:)
Tap Event	UITapGestureRecognizer (initWithTarget:action:)
Touch Event	UIGestureRecognizer, UIGestureRecognizerDelegate
Web View	UIWebView

Appendix C

Private Extensions Package

Accelerometer

Abstract: No
Generalization: Sensor

This entity models the accelerometer sensor that comes with most modern mobile devices. It measures the linear velocity of the device and can be used to allow users to interact with applications by physically moving their devices. As part of the Mobile Device dimension, the Accelerometer could be used to decide whether a particular ViewPoint should be activated. Finally, because this entity only models the availability of the sensor on the device, it should not be used to query the actual acceleration values.

Acceleration

Abstract: Yes
Generalization : Mobile Context Variable

This abstract entity groups together the linear acceleration values produced by the accelerometer of the mobile device.

Attributes

- isAvailable [Bool]: this attribute is inherited by all the specialization classes, and indicate the acceleration of the mobile device on the x, y or z dimensions. If the sensor is not available all the acceleration

values will be NULL.

Acceleration-X

Abstract: No
Generalization: Acceleration

Acceleration of the device along the x axis. If the accelerometer is not available the value of this variable will be NULL.

Attributes

- value [Float]: A measure in m/s2 that represent the acceleration along the x-axis. If the sensor is not available this attribute's value will be NULL.

Acceleration-Y

Abstract: No
Generalization: Acceleration

Acceleration of the device along the y axis. If the accelerometer is not available the value of this variable will be NULL.

Attributes

- value [Float]: A measure in m/s2 that represent the acceleration along the y-axis. If the sensor is not available this attribute's value will be NULL.

Acceleration-Z

Abstract: No
Generalization: Accelerometer

Acceleration of the device along the z axis. If the accelerometer is not available the value of this variable will be NULL.

Attributes

- value [Float] : A measure in m/s2 that represent the acceleration along the z-axis. If the sensor is not available this attribute's value will be NULL.

Altitude

Abstract: No
Generalization: Location

This variable indicates the altitude of the device with reference to the sea level. Consequently, positive values indicate above sea level measures, while negative values will indicate below sea level measures.

Attitude

Abstract: Yes
Generalization: Mobile Context Variable

Attitude values combine the measures of several sensors to determine the orientation of the device within a reference framework. The measures of this variable are characterized by the device Yaw, Pitch and Roll.

Attributes

- value [Float] : A floating point number representing the Yaw, Pitch or Roll of the device.

Battery

Abstract: Yes
Generalization: Mobile Context Variable

The device battery as described by its current level and status. In contrast with desktop applications, mobile apps are expected to deal with the challenges posed by a finite power supply as well as to contribute to its optimal usage. Because of this, applications may be in interested in learning the current Battery Level and decide, for instance, to execute a power intensive operation (e.g. involving network access) when the battery levels are higher or the device is plugged.

Battery Level

Abstract: No
Generalization: Battery

The remaining percentage of battery power.

Attributes

- value [Float]: A number between 0.0 and 1.0 that represents the remaining percentage of battery power. This value will be NULL in case the device is unable to report the current battery level.

Battery Status

Abstract: No
Generalization: Battery

A Mobile Context Variable that captures the state of the device battery as described by the following values: Unknown, in case the device is unable to report the battery level; Unplugged, indicating that the device is running on its battery; Charging, when the device is plugged to an electrical outlet; and Full, equivalent to a 1.0 Battery Level value.

Attributes

- value [Battery Status Description]: Can only take any of the following values: Unknown, Unplugged, Charging or Full.

Bluetooth

Abstract: No
Generalization : Network

The Bluetooth Mobile Context Variable, represents whether bluetooth connections can be established based on the availability and activation status of the hardware component.

Attributes

- value [Bool] : This value will be true if Bluetooth networks are available and the device is able to use them for data transfers. In any other case, this variable will be false.

Cellular

Abstract: No
Generalization: Network

The Cellular Mobile Context Variable represents whether cellular networks are activated and can be used for data transfer.

Attributes:

- value [Bool] : This value will be true if Cellular networks are available and the device is able to use them for data transfers. In any other case, this variable will be false.

Device Screen

Abstract: No
Generalization: Named Element

This entity encompasses the screen features of a mobile device that are required to activate a particular Viewpoint.

Attributes

- height[Float]: the height of the screen measured in pixels
- width[Float]: the width of the screen measured in pixels.
- density[Float] : the resolution of the screen measured in pixels per inch.

Device Sensor

Abstract: Yes
Generalization: Named Element

This entity encompasses the family of sensors that is included in modern mobile devices, namely: Proximity Sensor, Magnetometer, Gyroscope, Accelerometer, GPS, Video Camera, Microphone and Still Camera.

Direction

Abstract: Yes
Generalization: Mobile Context Variable

The Direction variable, describes the readings of the magnetometer sensor which can report the orientation of the device in relation with the magnetic north as well as the accuracy of such measures.

Enumeration Orientation Description

This enumeration describes the possible orientation modes that can be reported by mobile Operating Systems.

Literals

- Unknown: If the device is not able to report the current orientation.
- Portrait: If the device is held vertically with the front camera at the top and the menu button and the bottom. This is the natural position of the device.
- Landscape: If the device is held horizontally regardless of the position of the front camera and menu button.
- Portrait Upside Down: If the device is held vertically with the front camera at the bottom and the menu button at the top.
- Landscape Right: If the device is held horizontally with the front camera in the right and the menu button in the left.
- Landscape Left: If the device is held horizontally with the front camera in the left and the menu button on the right.
- Face Up: If the device screen is facing the user.
- Face Down: If the device screen is not facing the user.

Enumeration Battery Status Description

This enumeration describes the possible battery status values

Literals

- Unknown: If the device is not able to report the battery status.

- Unplugged: If the device is currently running on battery.
- Charging: If the device is plugged to an electrical outlet and its battery is currently charging.
- Full: If the device battery is fully charged.

GPS

Abstract: No
Generalization: Sensor

This Entity models the GPS sensor of mobile devices.

Attributes

- isAvailable [Bool]: A true value indicates that the device in use has GPS capabilities.
- isEnabled [Bool]: Since mobile users can decide whether they want applications to access their current location, this variable indicates if the users have allowed GPS tracking.

Gyroscope

Abstract: No
Generalization: Sensor

The Gyroscope entity models the availability of a sensor that is able to measure and report changes in the angular momentum of the device.

Attributes

- isAvailable [Bool]: A true value indicates that the device in use has a sensor that is able to report changes in the angular momentum of the device.

Magnetic Heading Direction

Abstract: No
Generalization: Direction

The Heading Direction describes the orientation of the Mobile device in relation with the magnetic north.

Attributes

- value [Float] : Device orientation measured in degrees. A value of 0^o degrees will indicate north, 90^o east, 180^o south and 270^o west. This value will be NULL in case the device is unable to report the heading direction.

Magnetic Heading Accuracy

Abstract: No
Generalization: Direction

The Magnetic Heading Accuracy variable, models the perceived error in the measurements captured by the magnetometer present in the device.

Attributes

- value [Float] : The perceived error in the magnetic heading direction measurement. The lower this value, the more accurate the reported direction.

Horizontal Accuracy

Abstract: No
Generalization: Location

The location services in Mobile Applications are achieved by means of a combination of sensors that include GPS, cellular and wifi networks. Like in any other positioning systems, depending on reference points available, the reported data may contain some errors. This Context Variable contains the horizontal accuracy of the reported location data.

Attributes:

- value [Float]: A value measured in meters, that describes the radius of the error ellipse of the reported location data. Negative values

indicate that the device is unable to report the horizontal accuracy of the location measurements.

Latitude

Abstract: No
Generalization: Location

This Context Variable describes the north-south component of a geographical coordinate.

Attributes:

- value [Float]: A value measured in degrees, that describes the north-south coordinate of the user's geographic position.

Location

Abstract: Yes
Generalization: Mobile Context Variable

This abstract entity encompasses all the sensor readings that describe the user location in terms of geographic position, speed and altitude along with the respective measurement errors.

Longitude

Abstract: No
Generalization: Location

This Context Variable describes the east-west component of a geographical coordinate.

Attributes

- value [Float]: A value measured in degrees, that describes the east-west coordinate of the user's geographical position.

Magnetometer

Abstract: No
Generalization: Sensor

The Magnetometer entity models the availability of a sensor that is able to measure the orientation of a mobile device with respect to the magnetic and true norths.

Attributes

- isAvailable [Bool]: A true value indicates that the device in use has a sensor that is able to report its orientation with respect to the magnetic and true norths.

Microphone

Abstract: No
Generalization: Sensor

This entity models the availability of an input device capable of capturing audio.

Attributes

- isAvailable [Bool]: A true value indicates that the device in use has a microphone.

Mobile Context Variable

Abstract: Yes
Generalization: Simple Context Variable

This entity provides a generalization for all the information produced by the sensors of a mobile device. Each of its specializations is concerned with the data produced by a particular sensor and because such readings are valid throughout the different sections of the app, all the specializations have Application Scope.

Mobile Device

Abstract: No
Generalization: Device

This entity models a modern mobile device considering the available sensors and the main features of its screen.

Association Ends

- screens [Device Screen] (0..*): Each of the device screens supported by a particular Viewpoint as described by their dimensions and pixel density.
- sensors [Device Sensor] (0..*): Each of the device sensors that are required on a particular Viewpoint.

Network

Abstract: Yes
Generalization: Mobile Context Variable

The Network Context Variable comprises all modern network connections that can be established from a mobile device. Its specializations, namely Wi-Fi, Cellular, NFC and Bluetooth networks, can be queried throughout the application and used by IFML designers to specify, for instance, the preferred network that should be used to perform certain task.

Attributes

- isAvailable [Bool]: this attribute is inherited by all the specialization classes, and indicate whether certain network is available for data transfer.

NFC

Abstract: Yes
Generalization: Mobile Context Variable

A key-value that describes the availability of NFC networks for data transfer.

Attributes

- value [Bool]: This value will be true if NFC networks are available and the device is able to use them for data transfers. In any other case, this variable will be false.

Orientation

Abstract: No
Generalization: Mobile Context Variable

A key-value pair that describes the current orientation of the mobile device through a set of fixed values.

Association Ends

- value [Orientation Description] (0..1) : The current device orientation as described by the values of the Orientation Description enumerator, namely: Portrait, Landscape, Portrait Upside Down, Landscape Right, Landscape Left, Faced Up and Face Down.

Pitch

Abstract: No
Generalization: Attitude

A key-value pair that describes the rotation around the lateral axis of the device –i.e. an axis that goes from side to side [15].

Attributes:

- value[Float] : Measure in radians of the rotation of the device along its lateral axis.

Proximity Sensor

Abstract: No
Generalization: Sensor

This entity represents a type of sensors that is able to recognize if the user is close to the device. Such sensor is typically used in applications that allow users to make and receive calls because it can perceive when the device is close to the user's ear and trigger an event that deactivates all the touch events avoiding in this way unintended interactions.

Attributes

- isAvailable [Bool]: This value will be true if the device in use has a proximity sensor. In any other case, the attribute's value will be false.

Proximity

Abstract: No
Generalization: Mobile Context Variable

A key-value pair that can be queried to verify whether the user is close to the proximity sensor of the device.

Attributes

- value [Bool] : This value will be true if the user is close to the proximity sensor, and false when the user is far from it. In case the device in use doesn't have a proximity sensor or its unable to report the proximity status, the attribute's value will be NULL.

Roll

Abstract: No
Generalization: Attitude

A key-value pair that describes the rotation along the longitudinal axis of the device – i.e. an axis that goes from top to bottom [15].

Attributes

- value[Float] : Measure in radians of the rotation of the device along its longitudinal axis.

Rotation

Abstract: Yes
Generalization: Mobile Context Variable

The rotation of the device along the three spatial axis as measured by the gyroscope.

Attributes

- isAvailable [Bool]: this attribute is inherited by all the specialization classes, and indicate the rotation measured along each spatial axis.

Rotation-X

Abstract: No
Generalization: Rotation

The rotation of the device along the x-axis.

Attributes

- value[Float] : Measure in radians of the rotation of the device along the x-axis. If the gyroscope is not available the value of this variable will be NULL.

Rotation-Y

Abstract: No
Generalization: Rotation

The rotation of the device along the y-axis.

Attributes

- value[Float]: Measure in radians of the rotation of the device along the y-axis. If the gyroscope is not available the attribute's value will be NULL.

Rotation-Z

Abstract: No
Generalization: Rotation

The rotation of the device along the y-axis.

Attributes

- value[Float]: Measure in radians of the rotation of the device along the y-axis. If the gyroscope is not available the attribute's value will be NULL.

Speed

Abstract: No
Generalization: Location

The ground speed of the device as measured by the location change rate over a period of time.

Attributes

- value[Float]: Measure in meters per second of the device ground speed. If the location services are not available this attribute's value will be NULL.

Still Camera

Abstract: No
Generalization: Sensor

This entity models the availability of rear and back facing, still cameras.

Attributes

- front [Bool]: A true value indicates the availability of a front facing camera that is able to snap pictures. In case the device doesn't have a front facing camera or it is not capable of snapping pictures this variable will be false
- rear [Bool]: A true value indicates the availability of a back facing camera that is able to snap pictures. In case the device doesn't have a rear facing camera or it is not capable of snapping pictures this variable will be false.

True Heading Direction

Abstract: No
Generalization: Direction

The Heading Direction describes the orientation of the Mobile device in relation with the true north.

Attributes

- value [Float] : Device orientation measured in degrees. A value of 0^o degrees will indicate north, 90^o east, 180^o south and 270^o west. This value will be NULL in case the device is unable to report the true heading direction.

Video Camera

Abstract: No
Generalization: Sensor

This entity models the availability of rear and back facing video cameras.

Attributes

- front [Bool]: A true value indicates the availability of a front facing camera that is able to shoot video. In case the device doesn't have a front facing camera or it is not capable of shooting video this variable will be false.
- rear [Bool]: A true value indicates the availability of a back facing camera that is able to shoot video. In case the device doesn't have a rear facing camera or it is not capable of shooting video this variable will be false.

Vertical Accuracy

Abstract: No
Generalization: Location

The accuracy of the reported altitude measure.

Attributes

- value [Float]: A value measured in meters, that describes measurement error of the reported device altitude with respect to the level of the sea.

Wi-Fi

Abstract: No
Generalization: Network

This key-value pair describes the availability of Wi-Fi networks for data transfers.

Attributes

- value [Bool]: This value will be true if Wi-Fi networks are available and the device is able to use them for data transfers. In any other case, the attribute's value will be false.

Yaw

Abstract: No
Generalization: Attitude

A key-value pair that describes the rotation of the device along an axis that is perpendicular to its surface [15].

Attributes

- value[Float]: Measure in radians of the rotation of the device along an axis that is perpendicular to its screen.

Relevant iOS Resources

Private Extensions Package Entity	iOS Resource
Accelerometer	UIRequiredDeviceCapabilities, CMMotionManager (accelerometerAvailable, accelerometerActive)
Acceleration	CMAccelerometerData, CoreMotion
Acceleration-X	CMAccelerometerData, CMMotionManager (accelerometerData)
Acceleration-Y	CMAccelerometerData, CMMotionManager (accelerometerData)
Acceleration-Z	CMAccelerometerData, CMMotionManager (accelerometerData)
Altitude	CLLocation, CLLocationManager
Attitude	CoreMotion, CMAttitude
Battery	UIDevice
Battery Level	UIDevice (batteryLevel)
Battery Status	UIDevice (batteryLevel)
Bluetooth	CoreBluetooth, CBCentralManager
Cellular	Rechability
Device Screen	UIDevice (model, orientation), UIUserInterfaceIdiom, UIDeviceOrientation
Device Sensor	UIRequiredDeviceCapabilities
Direction	CoreLocation, CLLocationManager, CLHeading
Enumeration Orientation Description	UIDeviceOrientation
Enumeration Battery Status Description	UIDeviceBatteryState

GPS	UIRequiredDeviceCapabilities, CLLocationManager (locationServicesAvailable)
Gyroscope	UIRequiredDeviceCapabilities, CMMotionManager (gyroAvailable, gyroActive)
Magnetic Heading Direction	CLHeading (magneticHeading)
Magnetic Heading Accuracy	CLHeading (headingAccuracy)
Horizontal Accuracy	Horizontal Accuracy
Latitude	Latitude
Location	CoreLocation, CLLocationManager
Longitude	CLLocation (coordinate)
Magnetometer	CMMotionManage (magnetometerActive, magnetometerAvailable), UIRequiredDeviceCapabilities
Microphone	UIRequiredDeviceCapabilities
Mobile Context Variable	CoreLocation, CoreMotion, CoreBluetooth
Mobile Device	UIRequiredDeviceCapabilities, UIDevice
Network	Rechability
NFC	N/A
Orientation	UIDevice, UIDeviceOrientation
Pitch	CoreMotion, CMAttitude(pitch)
Proximity Sensor	UIDevice (proximityMonitoringEnabled)
Proximity	UIDevice (proximityState)
Roll	CMAttitude (roll)
Rotation	CoreMotion, CMGyroData

Rotation-X	CMGyroData (rotationRate)
Rotation-Y	CMGyroData (rotationRate)
Rotation-Z	CMGyroData (rotationRate)
Speed	CLLocation (speed)
Still Camera	UIRequiredDeviceCapabilities, UIImagePickerController (isSourceTypeAvailable), UIImagePickerControllerSourceType, UIImagePickerControllerCameraCaptureMode
True Heading Direction	CLHeading (trueHeading)
Video Camera	UIRequiredDeviceCapabilities, UIImagePickerController (isSourceTypeAvailable), UIImagePickerControllerSourceType, UIImagePickerControllerCameraCaptureMode
Vertical Accuracy	CLLocation (verticalAccuracy)
Wi-Fi	Rechability
Yaw	CMAttitude (yaw)

Note about References

Although in the Bibliography section we enumerate all the references used during the project, in this section we list the more valuable resources and the role the played in shaping the final result of our project.

For the theoretical review we used the technical guides developed by Apple Inc [17] [8] [7] [9] [2], and the OMG standard of IFML [21]. These resources provided us with enough information and examples to grasp the main concepts of the target platforms.

The books authored by Martin Fowler[19], Markus Voelter [26] [25] and Marco Brambilla [18] provided the theoretical foundations that guided the code generation process. These books were a great source of good practices and advices for conducting a Model Driven Development effort.

Finally, the MD2 [22] project, developed at the Department of Information Systems in the University of Münsterin, provided some inspiration for the prototype driven approach used during the project.

Bibliography

[1] Apple Inc. Archives and Serializations Programming Guide. https://developer.apple.com/library/ios/documentation/cocoa/conceptual/Archiving/Articles/archives.html.

[2] Apple Inc. Concepts in Objective-C Programming. https://developer.apple.com/library/ios/documentation/General/Conceptual/CocoaEncyclopedia/Introduction/Introduction.html.

[3] Apple Inc. Concurrency Programming Guide. https://developer.apple.com/library/ios/documentation/General/Conceptual/ConcurrencyProgrammingGuide/Introduction/Introduction.html.

[4] Apple Inc. Data Management - SQLite. https://developer.apple.com/technologies/ios/data-management.html.

[5] Apple Inc. File System Programming Guide. https://developer.apple.com/library/ios/documentation/FileManagement/Conceptual/FileSystemProgrammingGuide/Introduction/Introduction.html.

[6] Apple Inc. Introduction to Core Data Programming Guide. https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CoreData/cdProgrammingGuide.html.

[7] Apple Inc. iOS Programming Guide. https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Introduction/Introduction.html.

[8] Apple Inc. iOS Technology Review. https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html.

[9] Apple Inc. Programming with Objective-C. https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html.

[10] Apple Inc. Property lists Programming Guide. https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/PropertyLists/Introduction/Introduction.html.

[11] Apple Inc. Transitioning to ARC Release Notes. https://developer.apple.com/library/ios/releasenotes/objectivec/rn-transitioningtoarc/introduction/introduction.html.

[12] Apple Inc. URL Loading System Programming Guide. https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/URLLoadingSystem/URLLoadingSystem.html.

[13] Apple Inc. View Controller Programming Guide. https://developer.apple.com/library/ios/featuredarticles/ViewControllerPGforiPhoneOS/AboutViewControllers/AboutViewControllers.html.

[14] Apple Inc. Views Programming Guide. https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/PropertyLists/Introduction/Introduction.html.

[15] Apple Inc. CMAttitude Class Reference. https://developer.apple.com/library/IOs/documentation/CoreMotion/Reference/CMAttitude_Class/index.html, 2011.

[16] Apple Inc. ADBannerView Class Reference. https://developer.apple.com/LIBRARY/ios/documentation/UserExperience/Reference/ADBannerView_Ref/index.html, 2013.

[17] Apple Inc. App Programming Guide for iOS. https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/iPhoneAppProgrammingGuide.pdf, 2014.

[18] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan and Claypool Publishers, 2012.

[19] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, October 2010.

[20] Google. Google Mobile Ads. https://developer.android.com/google/play-services/ads.html, 2013.

[21] Object Management Group. Interaction flow modeling language (ifml). http://www.omg.org/spec/IFML/1.0, February 2014.

[22] Herbert Kuchen Henning Heitkotter, Tim A. Majchrzak.
Cross-platform model-driven development of mobile applications with md2, 2013.

[23] International Data Corporation (IDC). Smartphone OS Market Share, Q2 2014.
http://www.idc.com/prodserv/smartphone-os-market-share.jsp, 2014.

[24] Object Management Group (OMG). Unified Modeling Language.
http://www.omg.org/spec/UML/2.4.1/, 2011.

[25] Markus Voelter. A catalog of patterns for program generation, 2003.

[26] Markus Voelter. *DSL Engineering: Designing, Implementing and Using Domain-Specific Languages*. Create Space Independent Publishing Platform, 2013.

Acknowledgments

This project represents the climax of a journey that started two years ago. A journey that I couldn't have completed without the help and support of many people that kept me sane through the toughest moments of this path.

First, and foremost I would like to thank my mum. For her unconditional support, guidance and wise advices. For being there, for not giving up on me, and for all the sacrifices that my life abroad has implied for her.

To my girlfriend and her parents. For teaching me that, like building a house, seeing the end of a project is an endeavour that needs to be faced brick by brick.

To all my cousins but specially to Sebastián, from whom I learned how to live with courage and to never give up in spite of the challenges that life puts in our paths.

I would also like to thank my university back home, Pontificia Universidad Javeriana, and my sponsor, Colfuturo, for making possible this marvelous experience, and for supporting me in every step of the journey.

Finally I want to express my gratitude to my supervisor, Marco Brambilla, for being always available for discussion and for helping me drive this project to a good end.