ISCTE ⊗ IUL

University Institute of Lisbon

Department of Information Science and Technology

Model-Driven Generative Programming for BIS Mobile Applications

Luís Miguel Pires Teixeira da Silva

A dissertation in partial fulfillment of the requirements for the degree of
Master in Computer Science and Business Management

Supervisor:
Fernando Brito e Abreu, PhD, Associate Professor
DCTI, ISTA

July, 2014

[This page was intentionally left blank]

Acknowledgments

I would like to thank:

– to my family, for all the given support and for the patience shown during the realization of this master thesis;

– to all my friends, for the encouragement and positive criticism which helped me guide and improve this thesis;

– to all the participants that were part of the experiments performed in this master thesis, for their commitment and dedication;

– to Prof. Fernando Brito e Abreu, which supervised this master thesis, for all the time and dedication, not only to this thesis but also to me, in both good and bad times.

[This page was intentionally left blank]

Abstract

The burst on the availability of smart phones based on the *Android* platform calls for cost-effective techniques to generate mobile apps for general purpose, distributed business information systems (BIS). To mitigate this problem our research aims at applying model-driven techniques to automatically generate usable prototypes with a sound, maintainable, architecture. Following three base principles: model-based generation, separation of concerns, paradigm seamlessness, we try to answer the main guiding question – how to reduce development time and cost by transforming a given domain model into an *Android* application?

To answer this question we propose to develop an application that follows a generative approach for mobile BIS apps that will mitigate the identified problems. Its input is a platform independent model (PIM), with business rules specified in OCL (Object Constraint Language). We adopted the Design Science Research methodology, that helps gaining problem understanding, identifying systemically appropriate solutions, and in effectively evaluating new and innovative solutions. To better evaluate our solution, besides resorting to third party tools to test specific components integration, we demonstrated its usage and evaluated how well it mitigates a subset of the identified problems in an observational study (we presented our generated apps to an outside audience in a controlled environment to study our model-based centered and, general apps understandability) and communicated its effectiveness to researchers and practitioners.

Keywords: Model-driven development, source code generation, Unified Modeling Language (UML), design patterns, software prototyping, object oriented software development

[This page was intentionally left blank]

Resumo

O grande surto de disponibilidade de dispositivos móveis para a plataforma *Android* requer, técnicas generativas de desenvolvimento de aplicações para sistemas comuns e/ou distribuídos de informação empresariais/negócio, que otimizem a relação custo-benefício. Para mitigar este problema, esta investigação visa aplicar técnicas orientadas a modelos para, automaticamente, gerar protótipos funcionais de aplicações com uma arquitetura robusta e fácil de manter. Seguindo para tal três princípios base: geração baseada no modelo, separação de aspetos, desenvolvimento sem soturas (sem mudança de paradigma), tentamos dar resposta à pergunta orientadora – como reduzir o tempo e custo de desenvolvimento de uma aplicação *Android* por transformação de um dado modelo de domínio?

De modo a responder a esta questão nós propomos desenvolver uma aplicação que segue uma abordagem generativa para aplicações de informação empresariais/negócio móveis de modo a mitigar os problemas identificados. Esta recebe modelos independentes de plataforma (PIM), com regras de negócio especificadas em OCL (Object Constraint Language). Seguimos a metodologia Design Science Research que ajuda a identificar e perceber o problema, a identificar sistematicamente soluções apropriadas aos problemas e a avaliar mais eficientemente soluções novas e inovadoras. Para melhor avaliar a nossa solução, apesar de recorrermos a ferramentas de terceiros para testar a integração de componentes específicos, também demonstramos a sua utilização, através de estudos experimentais (em um ambiente controlado, apresentamos as nossas aplicações geradas a uma audiência externa que nos permitiu estudar a compreensibilidade baseada e centrada em modelos e, de um modo geral, das aplicações) avaliamos o quanto esta mitiga um subconjunto de problemas identificados e comunicamos a sua eficácia para investigadores e profissionais.

Palavras-chave: Desenvolvimento orientado por modelos, geração de código fonte, Linguagem de Modelação Unificada (UML), padrões de desenho, prototipagem de software, desenvolvimento de software orientado a objetos.

[This page was intentionally left blank]

Index

List of Figures

Luís Silva – July, 2014

List of Tables

[This page was intentionally left blank]

Glossary

TERM	DEFINITION
ACID	Atomicity, Consistency, Isolation and Durability
ADT	Android Development Tools
API	Application Programming Interface
BIS	Business Information System
CASE	Computer- Aided Software Engineering
CRUD	Create, Read, Update and Delete
DAO	Data Access Object
DB4O	Database for Objects
DBMS	Database Management System
EMF	Eclipse Modeling Framework
GUI	Graphical User Interface
JPA	Java Persistency API
LOC	Lines of Code
MADP	Mobile Application Development Platform
MDA	Model-Driven Architecture
MDD	Model-Driven Development
MDT	Model Development Tools
MVC	Model View Controller
MVVM	Model View View-Model
OCL	Object Constraint Language
OODBMS	Object-Oriented Database Management System
ORDBMS	Object Relational Database Management System
ORM	Object Relational Mapping
PIM	Platform-Independent Model
POJO	Plain Old Java Object
PSM	Platform-Specific Model
SODA	Simple Object Database Access
SQL	Structured Query Language
RDBMS	Relational Database Management System
UI	User Interface
UML	Unified Modeling Language

WPF	Windows Presentation Foundation
XMI	XML Metadata Interchange
XSD	XML Schema Definition

1 – Introduction

1.1 – General introduction

The burst on the availability of smart phones and tablets based on the *Android* platform calls for cost-effective techniques to generate mobile apps for general purpose, distributed business information systems (BIS). What drove us in doing this research was the need to find a better solution for the time consuming app creation problem, as recognized in (Parada and Brisolara 2012): *"Developing applications for mobile platforms demands additional worries such as code efficiency, interaction with device resources, as well as short time-to-market".*

1.2 – Motivation

The aforementioned burst characterizes the growing mobile business ecosystem (Basole and Karla 2011). This concept is based on the ecological metaphor that firms are part of a larger ecosystem, each occupying a contributing role and forming symbiotic relationships with customers, suppliers, and competitors. This ecosystem is fuelled by the emergence of an "App Economy", enabling new products and services, but also influencing strategies and shaping business models (Page et al. 2013). For instance, according to the forecasts for the software market in the EU27 region, the apps share is the fastest growing one and will account for roughly half of that market by 2020 (see Figure 1).
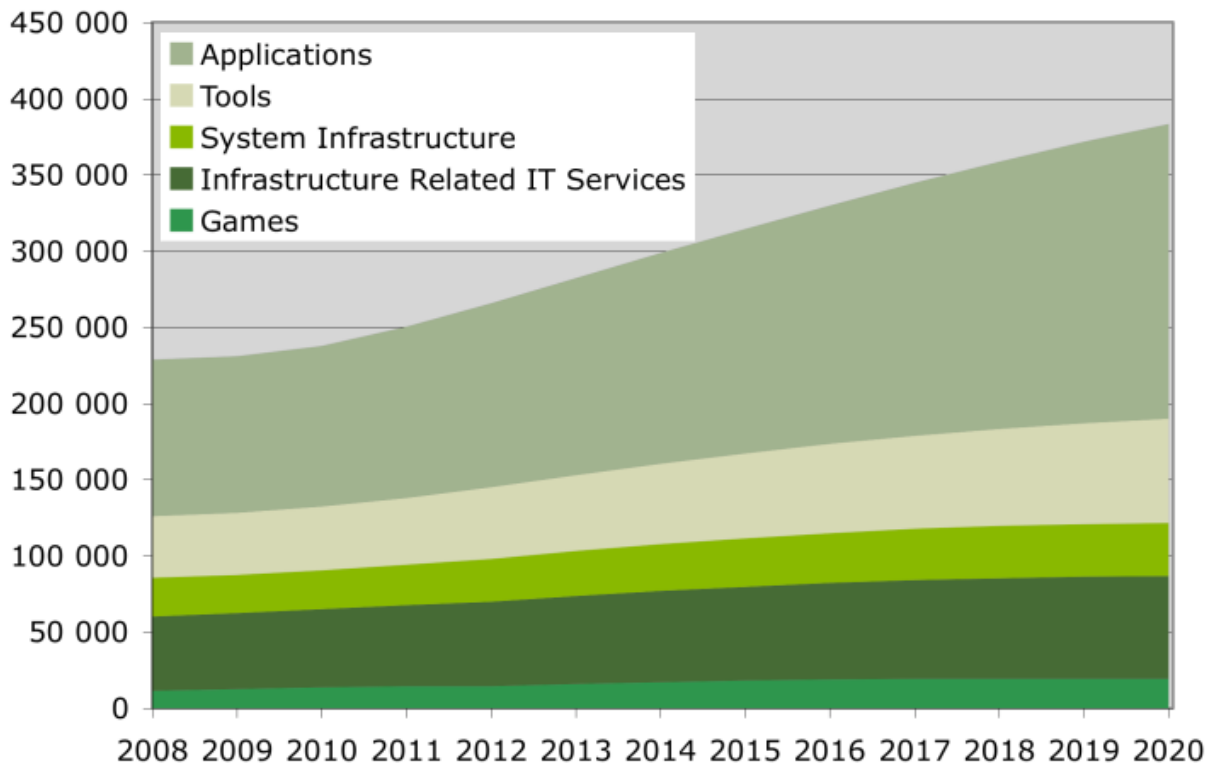
Figure 1 – Software market in EU27 in M€ (adapted from (Aumasson et al. 2010))

The expanding mobile ecosystem will put an increasing pressure in the demand for mobile business information systems (BIS) apps, therefore enforcing the aforementioned short time-to-market requirements (Parada and Brisolara 2012).

BIS apps target specific business-to-business or business-to-consumer problems and therefore they must be built "from scratch" to answer each business requirement. Furthermore, as aforementioned, developing such applications is a challenging task, due to several technical hindrances. Some are generic, such as the need to support localization features, and others are specific to mobile devices, such as the ability to support the diversity of available deployment platforms (e.g. diverse screen sizes, resolutions and orientation). The combination of these factors greatly increases both development time and cost.

The fast release pace of new Android versions also stands as a maintainability concern. As we can see in Table 1 (Wikipedia 2013a), the average period between each "major" release is around one year. Furthermore, notice that even the market is not able to follow the releases, showing still a large 28.5% usage of the 2.3 – 2.3.2 version. The API (Application Provided Interface – Interfaces that ease the implementation and/or provide functions to control specific system behaviors) level is also different for each released version. The greater it is, the more functionalities and easier interfaces are available for developers. Moreover, if we target the latest releases, a big portion of the market will not be able to use our application. The latter is an important fact since, even the application base structure might change depending on the level of the targeted API. For instance, the *Honeycomb* (API level 12) introduced the concept of fragments which drastically changed how applications can be structured. Therefore and especially for BIS apps, since they tend to endure in time within the business

ecosystem, providing a generative approach will facilitate maintainability and therefore will mitigate the problem of BIS apps migration due to platform evolution.

Table 1 – *Android* version evolution and actual (2/10/2013) app versioning distribution

Version	Code name	Release date	API level	Distribution
1.5	Cupcake	April 30, 2009	3	0%
1.6	Donut	September 15, 2009	4	0%
2.0–2.1	Eclair	October 26, 2009	7	0%
2.2	Froyo	May 20, 2010	8	2.2%
2.3–2.3.2	Gingerbread	December 6, 2010	9	0%
2.3.3–2.3.7	Gingerbread	February 9, 2011	10	28.5%
3.1	Honeycomb	May 10, 2011	12	0%
3.2	Honeycomb	July 15, 2011	13	0.1%
4.0.3–4.0.4	Ice Cream Sandwich	December 16, 2011	15	20.6%
4.1.x	Jelly Bean	July 9, 2012	16	36.5%
4.2.x	Jelly Bean	November 13, 2012	17	10.6%
4.3.x	Jelly Bean	July 24, 2013	18	1.5%
4.4	KitKat	October 31, 2013	19	0%

Due to these facts, we believe that model-driven generative approaches will in time become mainstream in BIS application development.

According to Gartner's viewpoint (Riza Babaoğlan 2013) mobile application development platforms (MADP) are generally based on one of three technologies. Each technology requires different investments and skills: native toolkits (e.g. Apple's iOS development toolkit), web toolkits (e.g. jQuery Mobile) and specialized platforms. Our approach fits in this last category. As claimed in (Riza Babaoğlan 2013), "*Specialized platforms take a more proprietary route, but generally provide more out-of-the-box enterprise capability than Web and native toolkits. They also often address more of the full software development life cycle — from application design, development and integration to testing, deployment and management. Some specialized platforms are optimized for high developer productivity, and others are optimized for high application performance and developer control*". Also, as shown in the latter, our approach falls, regarding the MADP market vendors, in the category of *application generators*, the same as *Kony*, that, as it can be seen in Figure 2, is considered one of the leaders.

Figure 2 – Gartner's magic quadrant – Mobile Application Development Platform vendors[1]

1.3 – Goals

Our main goal is to understand **how to reduce development time and cost by transforming a given domain model into an *Android* application**.

In order to fulfil the previous goal we must answer a more technical set of goals, namely the identification of adequate solutions to:

- Integrate different technologies;
- Generate an extendible and maintainable software architecture;
- Persist app data locally (required for offline usage), while being able to synchronize it with other users using the same app.

Our research aims at applying model-driven techniques to automatically generate usable prototypes with a sound, maintainable, architecture. Our generative approach is targeted to *Android* devices and we have adopted three principles to facilitate understandability and extensibility:

- Model centered generation – everything follows the model, from GUI to persistence;

---

[1] source: http://www.alibabaoglan.com/blog/gartner-2013-magic-quadrant-mobile-application-development-platforms/

- Separation of concerns – there is a clear separation in layers or components, each encapsulating a concern of its own;
- Paradigm seamlessness – the object paradigm is used throughout, since both the host language and data storage share the same type system, thus avoiding type conversions (e.g. from object to relational and vice-versa).

To reify the aforementioned principles we propose to: (i) generate GUIs that allow a conceptual navigation based on the type of relationships among domain entities (as described in a UML class diagram); use a GUI architecture that avoids code repetition and supports several screen sizes, resolutions and orientations; (ii) apply an architecture that separates different concerns in layers and apply already proven design patterns; (iii) allow seamless data persistence by using an object-oriented database; and finally (iv) grant distributed access by generating a simple Java server-side application, which allows data synchronization among mobile devices.

By providing a tool capable of generating robust and usable prototypes, we will free developers from repetitive tasks and by applying proven architectures and patterns developers will be able to extend or adapt the generated implementation – in case it does not fulfil the defined requirements or the domain model does not provide the means to represent them.

1.4 – Contributions

The main contribution of this dissertation is the proposal of a model-driven approach for automatic generation of Business Information Systems applications, to run in the *Android* platform. We also propose a model-based representation and navigation scheme for our BIS apps. Given that the generation targets a mobile platform, performance, screen size and resolution are main concerns which make development harder. We will introduce our approach to reach different screen sizes and resolutions with "minimal" effort, and to control screen orientation change in *Android*. We will also describe how we applied good development practices to an *Android* application, and how we implemented the already existing *Android* design patterns.

1.5 – Dissertation Guidelines

Given the fact that we are proposing a generative approach that focus in different areas, as consequence there are terms which affect different actors depending on the targeted area. Therefore during this dissertation the following concepts are going to be applied to better differentiate the influenced actors.

The actors are:

- Tool engineer – – mainly composed by technology/programming experts actors responsible for the generative approach.

- Domain expert – – mainly composed by business dedicated actors, responsible for the domain model specification and/or final app creation.

- Final user – – composed by every actor that may use the outcome final applications.

Along this dissertation, the given symbols will be placed next to a given requirement, in order to indicate the actors that will be affected by it.

1.6 – Dissertation structure

This dissertation is structured as follows: chapter two, the state of art, describes the current situation regarding the model-driven generative programming for the *Android* platform and also a brief exploration over generative approaches and tools for other platforms, but that are closer to our own approach. In chapter three we present our methodology. In chapter four, we present our solution. Chapter five presents a general review of all technologies used to support our generative approach, and a general explanation of the *Android* platform. In chapter six we describe the implementation of the chosen technologies, we explain the decisions, present and explain our approach regarding the generated applications. Chapter seven presents the research outcome, the *JUSE4Android* tool. Here it is explained the generator implementation, we explain the decisions and, present and explain our approach regarding our generative approach. Chapter eight, validates the decisions and approaches. Finally, in chapter nine, we draw our conclusions, review our contributions and forecast the future work.

2 – Related work

2.1 – Introduction

Model-driven generative approaches and tool development has been the subject of research since the eighties, by then under the *CASE (Computer-Aided Software Engineering)* acronym (Wikipedia 2013c). In this chapter we will survey and comment related generative approaches. Ideally they would be targeted for Android apps, use UML for model specification and Java/XML for implementation. In some cases, that related work only covers some of these requirements.

2.2 – *Android* specific related work

There are already some available software tools that generate code for *Android* apps, namely: *Basic4Android* (Uziel 2013) and *App Inventor* (MIT 2013). Despite the usefulness of these tools, our goals are different. *Basic4Android* is a commercial tool to speed up development by providing an IDE that allows a simple visual programming development style, allowing detailed customizations. *App Inventor* is based on MIT's *Open Blocks* (Roque 2007)*,* a graphical programming system especially suited for novice programmers. Although providing a more user friendly development environment, using these tools for developing BIS apps would require a lot of development effort, namely because it does not provide an adequate level of abstraction. For instance, regarding persistency actions, two different entities may behave the same way, In such approaches we must describe every aspect of such actions, for each entity, independently of being the exact same action or not.

In (Parada and Brisolara 2012) the authors also propose a generic round-engineering model-driven development approach for *Android* applications, based on UML class and sequence diagrams. This approach is not BIS-specific, but rather targeted at developing any type of application, since the approach allows to specify specific *Android* components and through sequence diagrams also specify different interactions between them. So it could also serve BIS apps but we would have the same abstraction level problem.

The IBM Rational Rhapsody (D. Holstein 2011; IBM 2013), which supports modeling and code generation for *Android* applications, shows itself as a complete model-driven solution, but in order to

Luís Silva – July, 2014

properly generate an application every detail must be specified, making the code generation almost a one-to-one mapping, again penalizing development effort. Examples of those details include, the need to specify in detail the actions to be executed in case we rotate the screen, to guarantee that we have the same data, but with the views adjusted to the new layout.

Finally, none of the previous tools or approaches show any concerns regarding data synchronization, an important BIS app requirement that must be fulfilled.

2.3 – Other related approaches

Since we propose herein a generative programming approach, we have also surveyed literature and available tools which take a similar approach to our own, although not targeting the *Android* platform.

2.3.1 – Dresden OCL

The Dresden OCL toolkit is a set of Eclipse plugins (Eclipse 2013). It provides model validation and also a model-driven generative approach. The toolkit supports metamodels in different formats (e.g. MDT (Model Development Tools) UML 2.0, Eclipse Modeling Framework (EMF), Java and XSD (XML Schema Definition)). Unfortunately, XMI (XML Metadata Interchange) import/export is not supported. The Dresden toolkit includes an OCL2 parser and editor (with syntax highlighting, code completion and code folding), an OCL2 interpreter, a Java/AspectJ code generator and a SQL code generator *(Demuth 2013)*.

Comprehensive details on the Java/AspectJ code generator and its integration with the Dresden toolkit, can be found in *(Wilke 2009)*. Several implementation techniques, like *StringTemplate (Parr 2006)*, a Java template engine to generate source code for fragments, are described there. The most important handicap on this approach, since it reduces models' expressiveness, is that it does not support associative classes. Due to the usage of the *Eclipse Modeling Framework (EMF)*[2]. In fact the Ecore[3] (EMF core) metamodels does not consider associative classes.

2.3.2 – OCLE (OCL Environment)

Although this project seems to have been discontinued, the Java code generator embedded in the OCLE toolkit (LCI_team 2005), still provides a good insight to this model-driven generative approach. This toolkit has its own GUI and, as the previous one, also provides model validation support. Java code is generated for every UML class and OCL constraint. Unfortunately, only a simple method checker is provided. The latter issues an error or warning, as a simple print, whenever a constraint fails. Other drawbacks deserve our attention, as follows: (i) the generated code has many

---

[2] http://www.eclipse.org/modeling/emf/
[3] http://download.eclipse.org/modeling/emf/emf/javadoc/2.9.0/org/eclipse/emf/ecore/package-summary.html

dependencies on their own libraries; (ii) the UML/OCL collection types are not translated to the corresponding *Java* parametric collection types, again relying on their proprietary collection libraries; and (iii) the full path of *Java* collection types is also used, instead of the simpler imports. While these aspects do not affect the application execution capabilities, they may affect code understandability and portability.

2.3.3 – The Naked Objects approach

The Naked Objects pattern, initially defined in (Pawson 2004), advocates that all business logic should be encapsulated onto the domain objects. It also recommends that the user interface should be a direct representation of the domain objects, with all user actions consisting, explicitly, of creating or retrieving domain objects and/or invoking methods on those objects. The user interface should be created 100% automatically from the definition of the domain objects, for instance combining source code generation and reflection techniques.

There are some tools that do code generation based on the previous approach, such as the *Naked Objects* for *.NET* (Pawson 2011) or the *Apache Isis* for Java (Haywood 2012). Both follow the same principle, which is providing automatically a strong base structure, where the programmer can then specify directly in code the domain model and reach other layers through annotations. Another example is *JMatter* (Suez 2013), a software framework, also based in the *Naked Objects* pattern, for constructing workgroup business applications, where the domain model can be specified in UML trough *Ultraviolet*, a light UML editor (Ramage 2006). Regarding the visualization they are all very similar, being the main difference that the first two are web oriented, and *JMatter* produces a Java GUI environment. All aforementioned examples seem to follow a table oriented view style to represent entities and their relationships, which could became a hindrance on small screen phones, due to space restrictions for showing information. None of these tools seem to support / adjust to multiple screen sizes. Last, but not the least, we have not found any *Naked Object* based tools targeting mobile devices.

2.4 – Conclusions

We surveyed some *Android*-specific generative approaches, but they did not meet BIS apps concerns. We also turned to other approaches and tools that, albeit not targeting the *Android* platform, were closer to our main goals. These tools provided us with a better insight of the BIS apps generative approach, on implementation techniques, and other generative specific concerns that somehow inspired our proposal. In Table 2 we can see the strong and weak points of these tools and/or approaches.

Table 2 – Strengths and weaknesses of generative tools and/or approaches

	Model-driven generation	Model customizability	Developing time (modelling plus coding)	App customizability
IBM Rational Rhapsody	yes	high	high	high
Dresden OCL	yes	medium	small	none
OCLE	yes	medium	small	small
Naked Objects	no	n/a	medium	high
Apache isis	no	n/a	medium	high
JMatter	yes	small	small	none

Except for the *Naked Objects* and *Apache isis*, notice that all approaches show a similar pattern. If we gain in model customizability (higher value) and app customizability (higher value), it seems we must sacrifice developing time (Smaller time is better). Based in Table 2, and in conclusion, we did not find any tool or approach that performed well for all the different BIS application requirements. The story seems to repeat itself in every targeted platform independently of the approach, that is, we either target the persistency and model layers, or we specifically target the UI construction, adaptability and flow control. If we target both, we either lose in developing time or customizability.

Regarding the *Naked Objects* and *Apache isis*, they seem to present the best maintenance approach. Such statement derived from the fact that these approaches are built based on a reflection approach and therefore there is not a domain specific language to describe the model, but instead we describe our domain directly on the target code. And based on simple annotations we reach and set rules to other layers like UI or persistency. The main problem with this approach is the main language restrictions which may directly reflect on understandability 👤 of the domain itself, for instance the simple description of a subclass usually requires a set of preparations (in order to use such API) that normally are not necessary in the normal natural language. Such approaches reveal themselves as great domain specific generative approaches since they mitigate very well the maintenance ⚙ problem and provide a great customizability 👤 , since the domain is directly described on our target code language (i.e. the programmer can create its own calculations code and make use of the API to only set the result). However where they gain in maintenance⚙ and customizability 👤 , they loose on understandability 👤 . For instance, only a high level programmer would be able to use such an approach and he would still have to learn the complex API system in order to properly use it. Finally, besides providing a very strong domain specific app generation, the final generated UI does not seem to be size concern and also does not seem to have screen rotation capability, maybe due to the fact that all these approaches still only target web type apps.

Finally, all the generative domain specific tools that offered a more complete generative approach (i.e. a complete or almost complete app would be generated), followed the same, or very similar, client/server data transfer pattern, i.e. the client would not persist any database data and therefore there is not a need for any synchronization strategies. Their drawback is that they do not allow the app to be used offline (i.e. without network connection).

A taxonomy for MADP generative approaches is outlined in Table 3. While it was used implicitly to assess the different proposals discussed in this chapter, we plan to improve it by objectifying the categorization. We have provisions to use the resulting taxonomy in a systematic review of MDD-based mobile application development platforms.

Table 3 – MADP taxonomy categories

Category	Sub-categories
Composition / extensibility	Getters/setters
	Navigations
Domain semantics enforcement / functionality	-
User interface / usability	-
Platform support / Portability	Independency on external/proprietary libraries
	Ability to handle multiple devices (sizes/resolutions)
Persistency	Local persistence
	Server synchronization granting distributed consistency
Efficiency features	Workload generation and benchmarking
Reliability features	JUnit tests generation

[This page was intentionally left blank]

3 – Methodology

The adopted methodology to lead this research was the Design Research. We followed the guidelines or "criteria" suggested in (Hevner and Chatterjee 2010), as shown in Table 4 to guide our research.

As shown in the 1.3 – Goals sub section we accomplished the first criteria since we provide a tool with generative capabilities. In 1.2 – Motivation, we showed how relevant the problem is (second criteria). In 1.4 – Contributions we presented this research contributions (fourth criteria). Regarding the evaluation and the research rigor (third and fifth criteria's) we followed proven design patterns, tested the tool by means of black box and white box techniques in both real and emulated devices, and we used third party tools to confirm the tests. In a more quantitative manner, in order to test final user experience and model app understandability, a series of tests were also taken over the generated apps. The research was presented and criticized by external experienced researchers, namely in the annual workshop QUASAR research group[4], and also in the MODELSWARD'2014[5] conference, thus fulfilling the seventh criteria.

Table 4 – Design research criteria. Source: (Hevner and Chatterjee 2010)

Criterion	Description
1. Design as an artifact	Design research must produce a viable artifact in the form of a construct, a model, a method, or an instantiation
2. Problem relevance	The object of design research is to develop technology-based solutions to important and relevant business problems
3. Design evaluation	The utility, quality, and efficacy of a design artifact must be rigorously demonstrated via well-executed evaluation plans
4. Research contributions	Effective design research must provide clear and verifiable contributions in the areas of the design artifact, design foundations, and/or design methodologies
5. Research rigor	Design research relies upon the application of rigorous methods in both the construction and evaluation of the design artifact
6. Design as a search process	The search for an effective artifact requires utilizing available means to reach desired ends while satisfying laws in the problem environment
7. Communication of research	Design research must be presented effectively to both technology-oriented and management-oriented audiences

The research was divided into three major phases: Exploratory, Development and Evaluation that were mainly carried out by this order, although some cycling occurred between them, namely because intermediate evaluations were required along the way. These phases, while presenting some differences, also combine with the three cycles (relevance cycle; rigor cycle; and design cycle)

---

[4] https://sites.google.com/site/quasarresearchgroup/
[5] http://www.modelsward.org/?y=2014

Luís Silva – July, 2014

presented by Hevner. Figure 3 shows in a timeline manner the focus on each stage. Notice that the literature review extends to the development stage. We will now describe each of those phases.
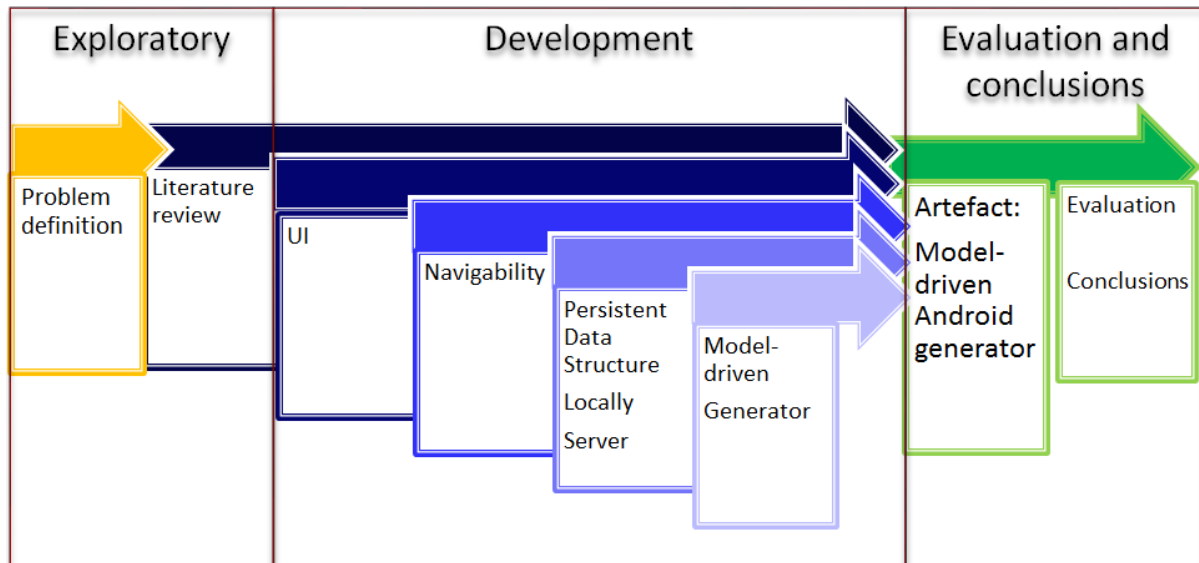
Figure 3 – Followed methodology lifecycle

3.1 – Exploratory phase

During this phase a problem was identified and a bibliographic literature review was conducted to identify possible approaches, available technologies and implementation techniques to deal with it.

3.2 – Development phase

The development of the proposed tool was split in three steps, as follows:

3.2.1 – Development of a prototypical *Android* BIS application

In this step we developed a simple *Android* BIS application to increase our understanding of the *Android* technology and possible problems for the model-driven generative approach since, we added all the types of constructs that we wanted to test. Those issues include the interaction between UI, user actions, navigability and data in *Android*. With this approach a more technical knowledge was gained, making it possible to find patterns in the development process of *Android* applications.

3.2.2 – Development of the proposed artefact

In this step, with a working prototype as a reference app, we started to implement the generator. In this phase we reviewed related approaches to code generation as described in section 2 – Related work. After the development we evaluated the generative capabilities of the tool by comparing the prototype application with the generated application.

3.2.3 – Generator Validation

In this last step we tested the generator, fixed eventual bugs and tested its scalability by providing it with different and, in each step, larger models. In each step we tested the generated apps and fixed or adopted the generative capabilities.

3.3 – Results Evaluation

Lastly we evaluated the final outcome of our research. By analyzing the current development approaches in the market with our generated applications functionalities we could study the feasibility of our approach. As aforementioned we also carried out two quantitative experiments in order to better improve UI interaction and most importantly, study the feasibility of the aforementioned model-based navigational approach.

[This page was intentionally left blank]

4 – Domain and GUI specification

4.1 – Introduction

The proposed architecture for our generative approach follows the Model-Driven Architecture (MDA) principles, since it *"... provides an open, vendor-neutral approach to the challenge of business and technology change. Based on OMG's established standards, the MDA separates business and application logic from underlying platform technology. Platform-independent models of an application or integrated system business functionality and behavior, built using UML and other associated OMG modeling standards, … "* (Object_Management_Group 2013). By adopting MDA principles, we aim at providing an architecture that enforces portability, domain specificity and productivity (Object_Management_Group 2013), therefore reducing the development schedule and cost for new applications.

In our proposal the Platform-Independent Model (PIM) is a UML class diagram embedded with OCL clauses and annotations. OCL is required because UML class diagrams do not allow providing all the relevant business constraints required for model specification. As claimed in (Warmer and Kleppe 2003) model-driven approaches require good, solid, consistent, and coherent models. We can build such models using the combination of UML and OCL.

4.2 – Domain specification (PIM)

To illustrate the usage of the proposed approach, the *Projects World* example (see Figure 4) will be used throughout this dissertation. The semantics of this example is straightforward. Each company hires workers and runs projects. Workers become employed when hired by a company; otherwise are unemployed. Each worker has a set of qualifications and can get more by attending a special kind of projects – the training ones. All projects require a set of qualifications to be run (become active).
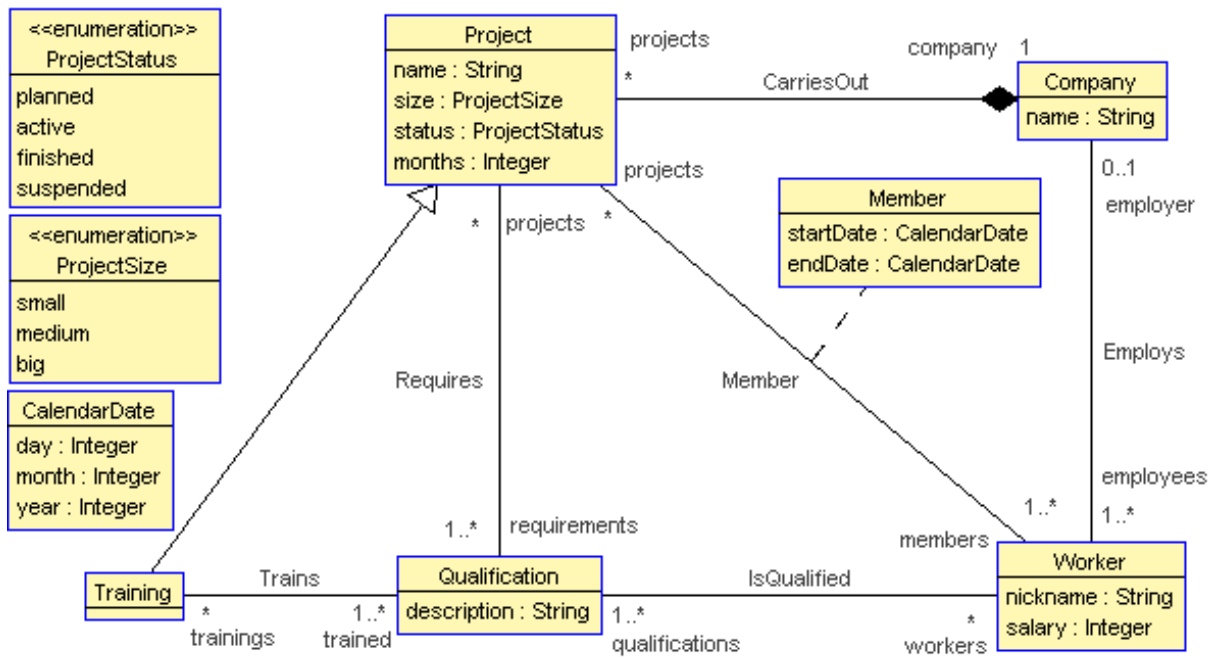
Figure 4 – Projects World UML class diagram

The serialized (textual) version of the *Projects World* model specification, contained in Projects World – USE specification is self-explanatory. To generate a fully-functional BIS app, namely to produce its different layers (e.g. view layer, domain layer, persistence layer), we need additional information in the model, that is added to it as annotations. The latter can also be used to discriminate between domain and utility classes (e.g. *CalendarDate*). Table 5 summarizes the available annotation types, along with their rationale.

Table 5 – Available annotation types

Annotation			Description	
Name	**Values**		**Description**	
	key	**Value** (String)		
StartingPoint	NameToDisplay	The name that will appear on the launcher screen to describe the class	This class will appear on the launcher screen (first screen to appear when application starts)	User Interface Definition
	ImageToDisplay	The path of the image. If absent a default image is used		
list	Any available attribute including inherited ones	Number indicating the order	Indicates which attributes will be shown in a list.	
creation	Any available attribute – already contains inherited ones	Number indicating the order	Indicates which attributes need to be filled by the user in order to create an object.	
display	Any available attribute – already contains inherited ones	Number indicating the order	Indicates which attributes should be shown in the detail screen (the screen that describes the object).	
unique	Any available attribute – already contains inherited ones	Number indicating the order	Indicates which attributes are going to be used in order to create a unique ID.	Model and Persistency
holder	none	none	Used over the associations to specify which side of a relationship will hold the data. Only viable for specific associations like a many-to-many association.	
domain	none		Used to differentiate domain classes from utility classes.	PIM

The following code snippet illustrates the use of annotations for the *Project* class.

```
@StartingPoint(NameToDisplay="Projects", ImageToDisplay="project")
@list(name="1")
@creation(name="1",size="2",status="3",months="4")
@display(name="1",size="2",status="3",months="4")
@unique(name="1",size="2",status="3",months="4")
@domain()
```

These annotations allow generating a full working prototype without requiring other external inputs or PSM, while granting specification simplicity.
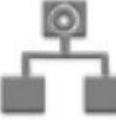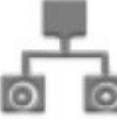
The generation approach follows a specific template but, as expected, for the same model, the generated implementation may vary depending on the provided annotations

4.3 – GUI specification

4.3.1 – Model-driven navigation metaphor

Regarding the generated GUI, we propose a homomorphism between the traversal of the domain space and the app navigation space. We have identified a limited set of domain traversal (navigation) genders and we assigned an icon to each one, as shown in Table 6. These icons are used in the navigation bar to provide semantic advice to the user, when he decides where to move to. Each domain traversal gender corresponds to a single movement from one domain entity to another, towards a UML association end (e.g. with cardinality one or many) or inheritance relation end (towards the parent or the children classes). For instance, in the *Projects World* app, while standing in the *Worker* form, we would get a "to many" icon for navigating to *Qualification* and a "to one" icon for navigating to *Company*.

<div align="center">Table 6 – Navigation Icons</div>

Icon	Navigation Gender
	To One
	To Many
	To Associative
	To Super
	To Sub

Therefore, the given domain model affects directly how the user navigates and perceives the objects in the application. In Figure 5 we can see the launcher screen of the *Projects World* example.

The available options (i.e. the domain types we can choose to explore) are the classes that have been marked with the *@StartingPoint* annotation.
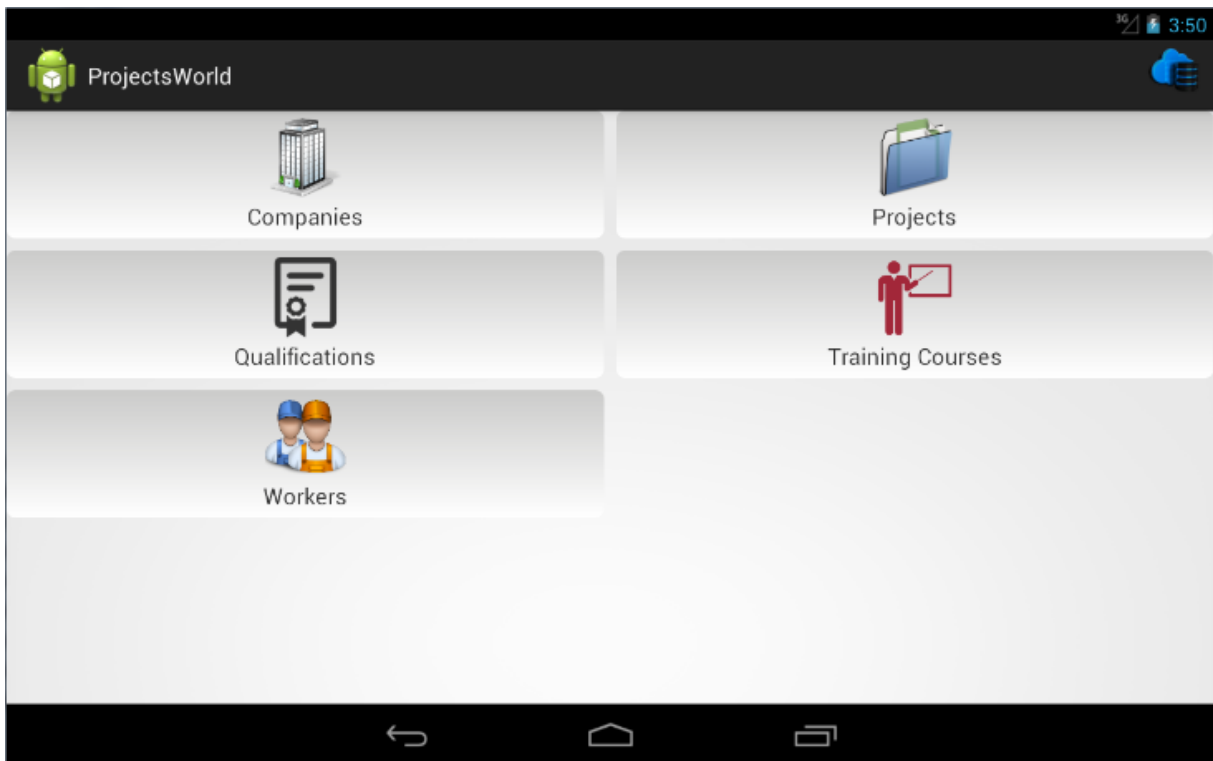
Figure 5 – Projects World launcher screen

4.3.2 – GUI views and widgets

Let us consider that we press the *Workers* button, whose class is associated with *Qualification, Company, Member* and *Project*. The provided GUI functionalities would include the ability to: (i) browse and select the available *Worker* instances (List View), (ii) access the detail of a selected *Worker* instance (Detail View), (iii) navigate to the related domain entities (Navigation Bar) and (iv) apply the basic CRUD operations. As shown in Figure 6, the aforementioned four requirements are met by three distinct views, each with its own purpose, and by adding three buttons (create, update and delete) to the default *Android ActionBar*.
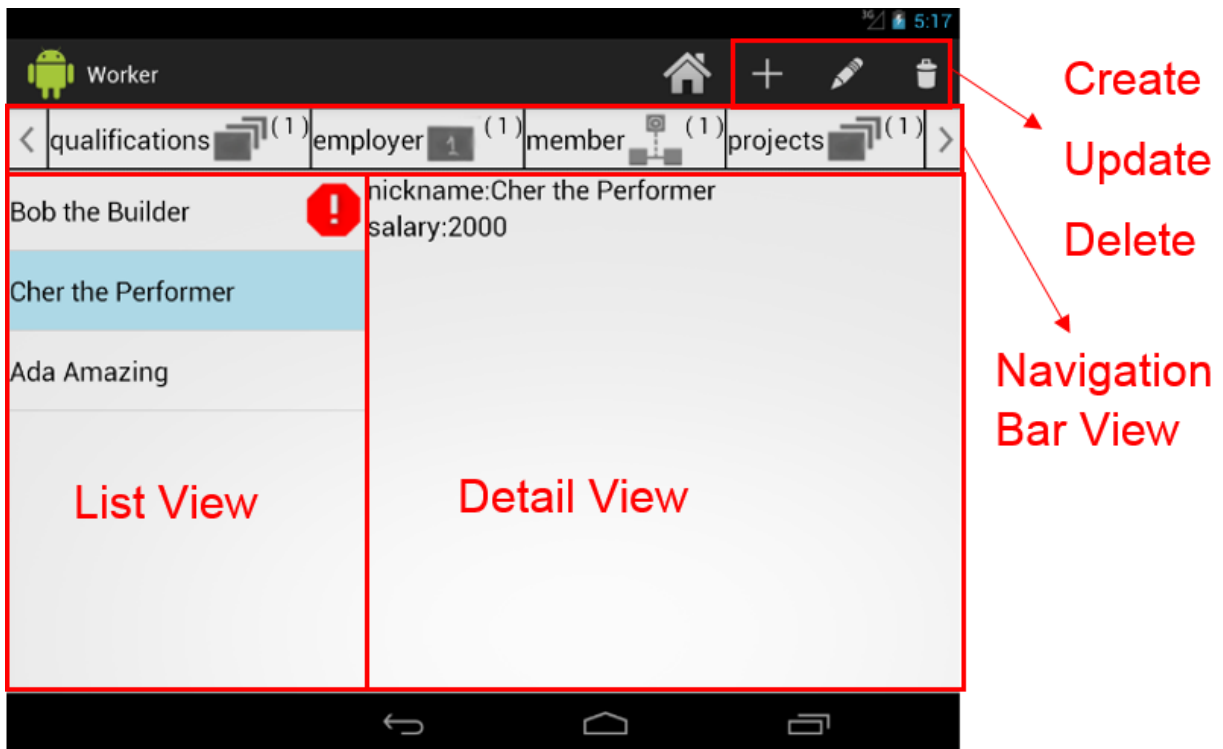
Figure 6 – Screen Division – Worker Class Example

As already shown in Table 5, we can configure what we want to show in each view. For instance, in the *List View* of the *Worker* class we only show the attribute "nickname" since it is enough to distinguish the workers among themselves. This behavior can be specified with the @*list* annotation.

The *Detail View* always follows this template style, where the attributes description is followed by their assigned values, but it can be static (for reading purposes only, as shown in Figure 6) or dynamic (for creating new instances or updating existing ones). Finally, the *Navigation Bar View*, as already explained, has a button for each allowable navigation, through class associations and/or inheritance relationships. In the case of the *Worker* class we have three associations, but since one of them is an associative class we have four possible navigations. The corresponding buttons show the association role, followed by the navigation icon and the number of objects of the associated entity that are linked to the current *Worker* instance. Clicking in one button changes the context to the selected domain entity. The alert icon shown in Figure 7 is used to indicate that an object is not fulfilling all its constraints, if the user clicks on it a message will appear telling which constraints are not being met.

Figure 7 – Alert icon

4.3.3 – Objects and links creation

If the user creates a new instance through the add button, the latter will be persisted, but it will not be linked automatically to any other instance. In order to do so, the user must do a long-click (pressing a button constantly for a few seconds) over the corresponding association, to trigger one of two possible scenarios:

- Normal class scenario – the user navigates to a new screen corresponding to the select target type, in WRITE mode i.e. with the intention of associating something.

- Inheritance class scenario – the exact same behavior as in the normal scenario with the difference that, before the user navigates to the new screen, he will be prompted with a dialog in order to choose the desired sub class. That is, when the user navigates to, and only to, a super class (class that is parent to, at least, one other class), it is expected that the user defines which type he really intends to associate to (the selected super-class or one of its children). This behavior is normal since sub-classes also inherit their parent associations. In Figure 8 we can see such a dialog from our *Projects World* example. In this case the user is present in either Qualification, Worker or Company dedicated screens and did a long click in "projects". Since the Project class is not abstract, both Project and Training types are available thus granting the user with the choice of associating with a Training (user navigates to Training thus only seeing trainings which are also projects) or, a Project (user navigates to Project thus sees both projects that might be super of Training or just projects).
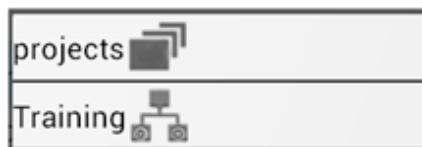
Figure 8 – Dialog when navigate in WRITE mode to Project

In the new screen, in WRITE mode, the user can choose an already existing object or create a new one. In the former case (explicit linking), the user presses the *"Confirm"* button ("check" icon in Figure 9) and the selected object will become linked to the one where the navigation started. In the latter case, the user presses the create button ("plus" icon in Figure 9) and the newly created object will be implicitly linked to the one where the navigation started.

Figure 9 – Available options in WRITE mode

After any of the latter operations are completed, the system closes the screen and it reopens and updates the former screen (screen where the association creation action began).

4.3.4 – Server synchronization support

There are three distinct mobile app types where internet connection is the key differentiator: those that only work locally; those that retrieve all dynamic data from a server and therefore only work if an internet connection is available and finally a mix of the previous two, i.e. where the mobile app can alternate between online or offline periods. Since the latter case characterizes BIS apps, where multiple users will be using it in a distributed fashion, we must provide synchronization features between the local database (required for offline usage) and the server database (required for keeping the overall system state consistent. Regarding the GUI specification, database synchronization is triggered by the provided button on the launcher screen, as shown in Figure 10. This button is only active when an internet connection is available.
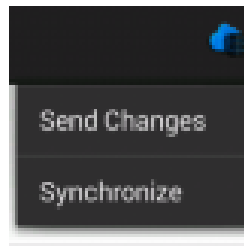
Figure 10 – Synchronization button

5 – Android apps technological requirements

5.1 – Android

Although built for mobile and tablet-based devices, the *Android* operating system exhibits the characteristics of a full-featured desktop framework (Google 2013a). Application developers rarely feel they are writing to a mobile device because they have access to most of the class libraries available on a desktop or a server – including a relational database (Komatineni and MacLean 2012). The language syntax used in *Android* is based on *Java*, but despite this similarity it uses a different virtual machine (Dalvik Virtual Machine). As a result, not all Oracle Java Virtual Machine libraries are available[6]. *Android* uses XML to describe the user interfaces (UI) and/or raw data, and it offers several different ways of persisting data.

5.2 – Android Applications Structure

5.2.1 – Architecture

The *Android* UI framework, like other Java UI frameworks, is organized around the common MVC (Model-View-Controller) pattern (Mednieks et al. 2012). Nevertheless, it does not fulfill all the required requirements, namely the usage of the XML language in order to represent the user interfaces (UI) for dynamic data. For instance, we can define views in XML but we cannot assign instances declared in Java in the XML, only the other way around. As it can be seen in Figure 11, the View layer knows about the Model layer, which in *Android* cannot be done when representing dynamic data, therefore we must create a middle layer with the purpose of inflating (instantiating) the XML files and setting its values. The other solution in this case would be defining the UI in Java code, but if we do so, we would not be following the software development good practices: "*The advantage to declaring your UI in XML is that it enables you to better separate the presentation of your application from the code that controls its behavior. Your UI descriptions are external to your application code, which means that you can modify or adapt it without having to modify your source code and recompile.*" (Google 2013a). The

---

[6] http://developer.android.com/reference/packages.html

approach followed by the *Android* operating system, is to guarantee that its UI looks the best on each device, by taking care of screen size adjustments and/or orientations for different device types. (Google 2013a). To achieve these adjustments, UIs are declared in XML files and pre-compiled.
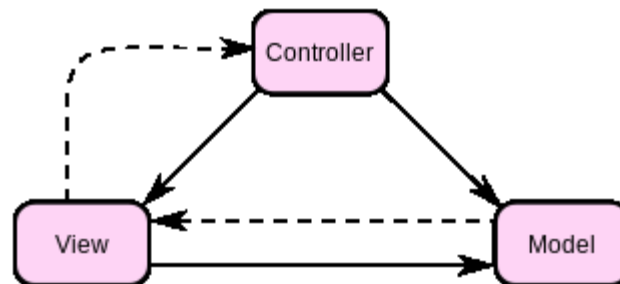
Figure 11 – Model View Controller pattern

5.2.2 – General structure and functionality

Independently of how *Android* applications are going to be used, they are built over five main building blocks (class containers): *Activities*, *Intents*, *Services*, *Broadcast Receivers*, and *Content Providers*. There is not a need to use all five blocks simultaneously, but certainly one or a combination of them is going to be used in every application. All the classes defined on the aforementioned blocks must be declared in the *manifest.xml* file. Not doing so will result in an error when the application tries to use/call the corresponding class.

Android Manifest

The *manifest.xml* file holds application-specific configuration information, is mandatory in any *Android* project and is located in the project folder (the root folder of the project). From all the possible settings this file may contain, we can highlight as essential: the definition of the package name, the definition of all the components used by the application, the required application permissions and the minimum *Android* API level required by the application. By default, all the automatic behavioral settings are on. So, if we want to control any of these settings, we must declare them in the manifest. As an example, consider the configuration change control. By default, the screen rotation is enabled, but we can disable it in the manifest and choose a standard orientation setting for the application (e.g. adding the line `android:screenOrientation="portrait");`

Activities

Activities are the most common *Android* building blocks. An activity can be seen as a "screen". Just like a webpage, we can view all the contents it is displaying and, we can interact with the components it is holding. Each activity is responsible for showing some pre-determined type of information. Users navigate between activities in the app UI. Each activity is implemented as a single class that extends the *Activity* base class and it will display a user interface composed of Views (which may be declared

Luís Silva – July, 2014

in XML) and respond to user or system events. For example, in *Android* phones we often have an activity that shows a list of the available contacts. When we press one contact, we navigate to another screen/activity responsible of showing the contact in a more detailed manner. So, we can say that moving to another screen is accomplished by starting a new activity. An activity is also able to return a value to the previous activity (i.e. its caller). For example, an activity that lets the user pick a photo would return the chosen photo to the caller. In order to start a new activity, i.e. navigate to it, intents musts be used.

Intents and Intent Filters

An intent generically defines an "intention" to do some work (Komatineni and MacLean 2012). Intents can be initialized by our own application or by the operating system, for example for notification purposes. Intents can be implicit or explicit, that is, we can send simple information data and, in an implicit way, let the system decide what application or activity is the best to handle the requested intent, or we can create an intent and explicitly define the source and target. For most cases, as for application navigation purposes, we can simply create a new intent with two arguments, the caller activity and the destination activity. For an implicit call, which is depending on what the system has to offer, the two most important parts of the intent are the action and the data to act upon. Depending of our intentions, these requirements may vary. For example, if our application manages information like emails or contacts, and we want to send an email to one of those contacts, then, instead of creating our own activity to send email messages, we can start an activity with an intent like the following:

```
Intent intent = new Intent(Intent.ACTION_SEND);
intent.putExtra(Intent.EXTRA_EMAIL, recipientArray);
startActivity(intent);
```

This allow us to reuse the system default email sending activity or, if there are many available, it will ask the user to decide which one to use. Activities will be chosen depending on the action. Typical values for the latter are MAIN (the launcher activity of an application), VIEW, PICK, EDIT, etc. For example, if we have an URL, we can call the default browser to see the corresponding page by creating an intent with the VIEW action, and send the URL as Website-URI, like in:

```
new
Intent(android.content.Intent.VIEW_ACTION,ContentURI.create("http://developer.android.com"));
```

If we expect a result from the called activity, then, instead of using the `startActivity(intent)`, we can use the `startActivityForResult(intent, requestCode)`. For example, we can retrieve a contact by using the already defined *Android* default actions and codes like this:

```
Intent intent = new Intent(Intent.ACTION_PICK, Contacts.CONTENT_URI);
startActivityForResult(intent, PICK_CONTACT_REQUEST);
```

In order to catch the result every *Activity* offers the following method that will be triggered every time the called activity posts a result:

```
onActivityResult(int requestCode, int resultCode, Intent data)
```

So far we have shown how to navigate between activities inside an application and how to make use of the activities already available, or expected to be, on the system. While an *Intent* may be seen as a request, an *IntentFilter* describes what intents an *Activity* or *Intent Receiver* are capable of handling. These are described in the *manifest.xml* file.

An *Android* application can also offer its own activities for other applications to deal with data. Navigation is accomplished by resolving intents. If the destination is not defined, we may say that the system checks all the installed intent filters and picks the one that best matches the given intent.

Android also retains history stacks for each application launched from the home screen. When the user navigates to another screen/activity or the current screen/activity loses focus (i.e. another screen/activity gains focus) due to unforeseen system events, the activities enter in a paused state and are put onto a history stack. This mechanism allows to restore the control after any unforeseen system event, or navigate back and forward. Activities are also cleaned from the history stacks, when they are no longer required. For example, we can create an *Intent* and by making use of an already defined flag, like "`Intent.FLAG_ACTIVITY_CLEAR_TOP`", we can navigate to another activity cleaning every other activity opened by our application, making the next activity the first in the history stack.

Intent receivers and broadcast receivers

As shown before with the *Intent Filters*, we can use the *Intent Receiver* when we want our application to react to an external event, as when the phone rings, or when a network connection is available. Intent receivers do not display an UI, although they may display notifications to alert the use. Intent receivers are also registered in the *manifest.xml*, but we can also register them dynamically in code using `Context.registerReceiver()`. Applications do not have to be running for their intent receivers to be called. When an intent receiver is triggered, if necessary, the system will start them. Applications can also send their own intent broadcasts to others with `Context.broadcastIntent()`.

In order to implement a receiver, we must implement a class that extends *Broadcast Receiver*. Then we can listen to events through the method `onReceive()`. As of Android 3.1, the *Android* system will by default exclude all *BroadcastReceiver* from receiving intents if the corresponding application has never been started by the user or if the user explicitly stopped the application via the Android menu (in *Manage Application*) (Vogel 2013).

Content providers

Applications can persist data by means of a *SQLite* database, *SharedPreferences* (a primitive key-value pair storage system provided in *Android* – usually used for simpler data types, for example for persisting user preferences) or any other way. A Content Provider is a class that implements a

standard set of methods, like CRUD, to let other applications store and retrieve the type of data that is handled by that content provider. So, it can be used if we want to share our application data with other applications.

Services

A *Service* is a class that runs without a UI and long-lives beyond the general state of the application, for example locking the screen does not affect its state. We can create services that run independently of our application state, for example to count the number of calls received, for statistical purposes. A good example of a possible service is a media player. A media player found in any *Android* phone usually provides activities that allow the user to choose songs and start playing them. If a song is already playing, the user will be able to "close the application" (i.e. exit the presented activity, go back to the home screen, lock the phone) without stopping the playback. For this scenario the playback cannot be handled by an activity, but instead by a service. The media player activity would start a service using `Context.startService()` to keep the music playing in the background, until the service has finished. We can also connect to a service with the `Context.bindService()` method. When connected to a service, we can communicate with it through an interface exposed by the service, for example regarding the music service (e.g. to allow pause, rewind or stop).

User interfaces

As aforementioned, *Android* UIs can be built either by defining XML structures (best option) or by programming them in plain *Java*. As also said, an activity can be seen as a screen and we inflate XML views in to it. In order to express the UI, we must work with *Views* and *ViewGroups* – the basic units of user interface expression on *Android*.

Views

The *View* class represents the basic building block for user interface components – found in *android.view.View* – It occupies a rectangular area on the screen and is responsible for drawing and event handling. It is also the base class for widgets. *Android* offers a varied set of widgets, (e.g. *TextView*; *EditText*; *RadioButton*; *CheckBox*; *Button*; *ScrollView*;) which are used to create interactive UI components. The widgets handle their own measuring and drawing, so we can use them to build our UI more quickly.

ViewGroups

The *ViewGroup* subclass is the base class for *layouts*, which are invisible containers that hold other Views (or other *ViewGroups*) and define their layout properties as shown in Figure 12. By using *ViewGroups* we can add structure to our UI and build up complex screen elements that can be addressed as a single entity. *Android* offers a varied set of predefined *ViewGroup* subclasses that provide common types of screen layout (e.g. *RelativeLayout*, *LinearLayout*).

In Figure 12 we can see the *Android View* class hierarchy and get a better perspective of the available widgets.
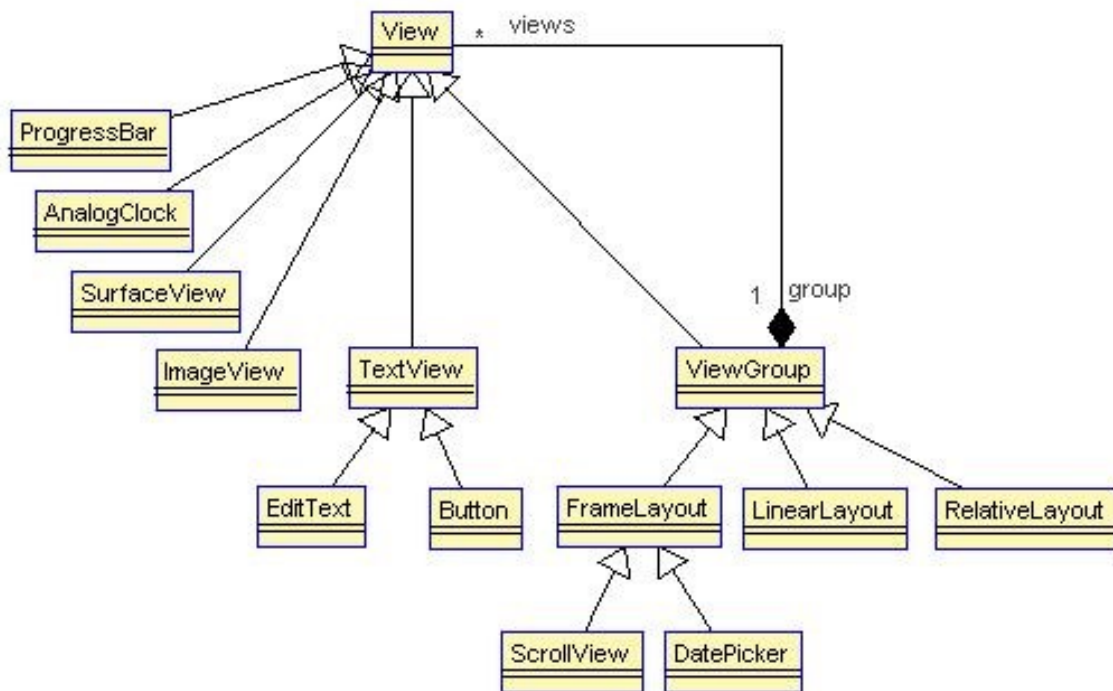
Figure 12 – Layout building blocks

A Tree-Structured UI

Now we can define an *Activity* UI using a tree of *View* and *ViewGroup* nodes, as shown in Figure 13. The tree can be as complex as required and can be defined in one or many XML files.

Figure 14 shows an example that matches the tree represented in Figure 13. As the latter shows the two *ViewGroup*s: *LinearLayout* and *RelativeLayout*, are not seen by the user. We use them to order and properly place other *View*s in the screen.
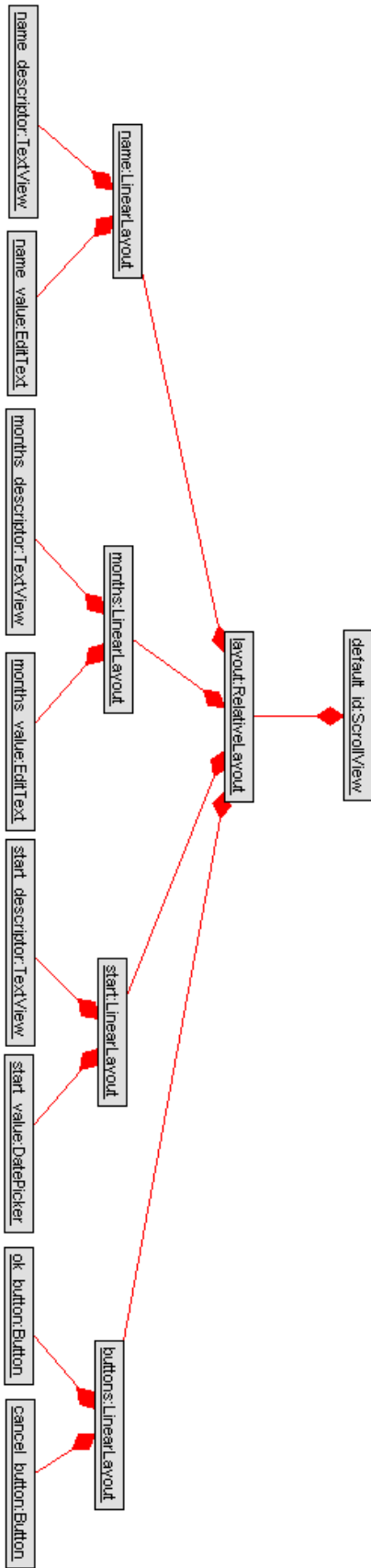
Figure 13 – Illustration of a view hierarchy, which defines a UI layout

To attach the defined tree to one *Activity* we must call, in the *Activity*, the `setContentView(<Resource ID to be inflated>)` method and pass a reference of the intended tree layout. Once the *Android* system has the reference, it can work directly with the root node to invalidate, measure, and draw the tree. When any *Activity* becomes active and receives focus, the system notifies the *Activity* and requests the root node to measure and draw the tree. The root node then requests that its child nodes draw themselves and, in turn, each *ViewGroup* node in the tree is responsible for drawing its direct children. Each *ViewGroup* has the responsibility of measuring its available space, laying out its children, and calling `draw()` on each child to let it render itself. The children may request a size and location of the parent, but the parent object has the final decision on where and how big each child can be.
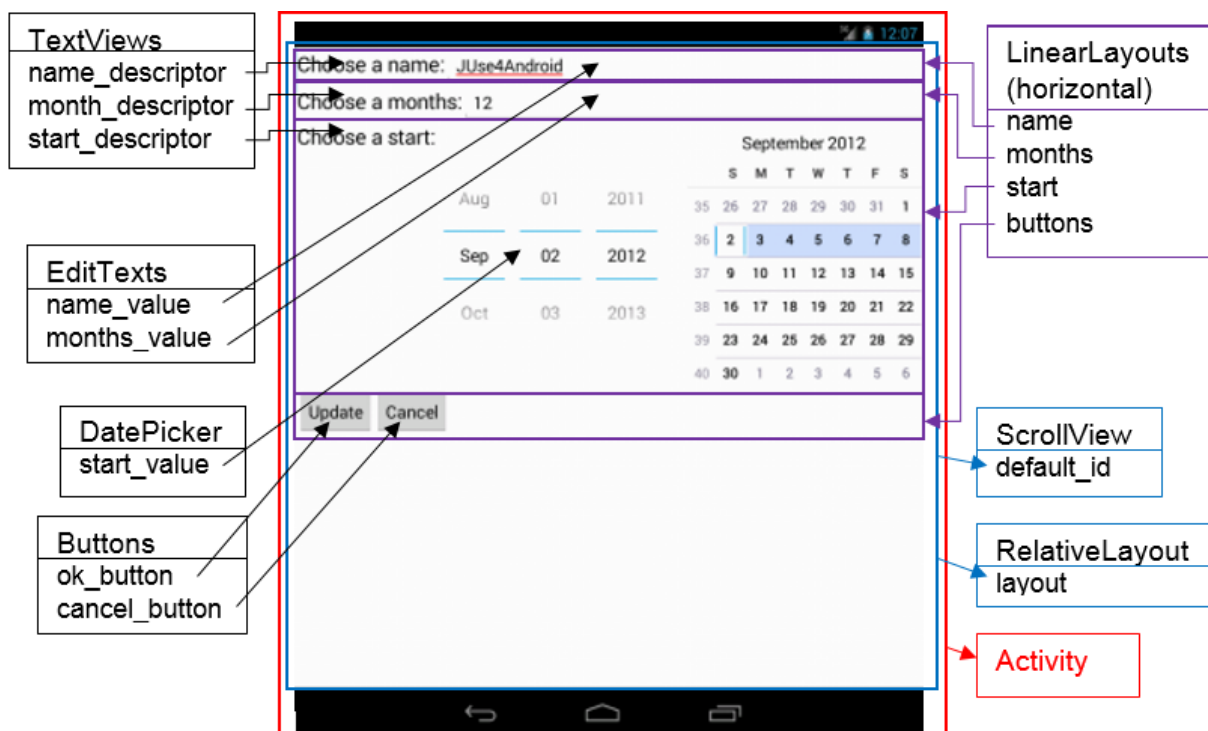
Figure 14 – User Interface Tree – Illustrating example

Resources

In *Android* there are two main folders, the "*src*" (source) that will hold all the *.Java* code files and the "*res*" (Resources) folder that holds any resources. Resources are external files (non-code files) that are used by the code and compiled into the application at build time. Android supports a number of different kinds of resource files, including XML, PNG, and JPEG files. The XML files have very different formats, depending on what they describe. Resources are externalized from source code, and XML files are compiled into a binary, fast loading, format (e.g. by compressing strings) for efficiency reasons.

To differentiate among the different resource types, *Android* offers a set of folders inside the "*res*" folder, each with its own purpose, namely:

- layout-files – "/res/layout/"
- menu-files – "/res/menu"
- images – "/res/drawable/"
- animations – "/res/anim/"
- styles, strings and arrays – "/res/values/"
- raw files like media (music or videos) – "/res/raw/"

Screen size, resolution, orientation and localization

Android offers a static way to handle the required UI adaptation, regarding the different screen sizes, resolutions, orientations and languages. By implementing predefined qualifiers, in the list of resources folders shown before, we can benefit of the *Android* automatic UI behavioral system. The available predefined qualifiers apply to the screen size (*small*, *normal*, *large*, *xlarge*) and to the screen resolution or density (*ldpi*, *mdpi*, *hdpi*, *xhdpi*), as represented in Figure 15. There are also predefined qualifiers for the screen orientation (*land*, *port*) and localization, following the usual two letter ids used for language support such as: *en* (English), *pt* (Portuguese), or *es* (Spanish).
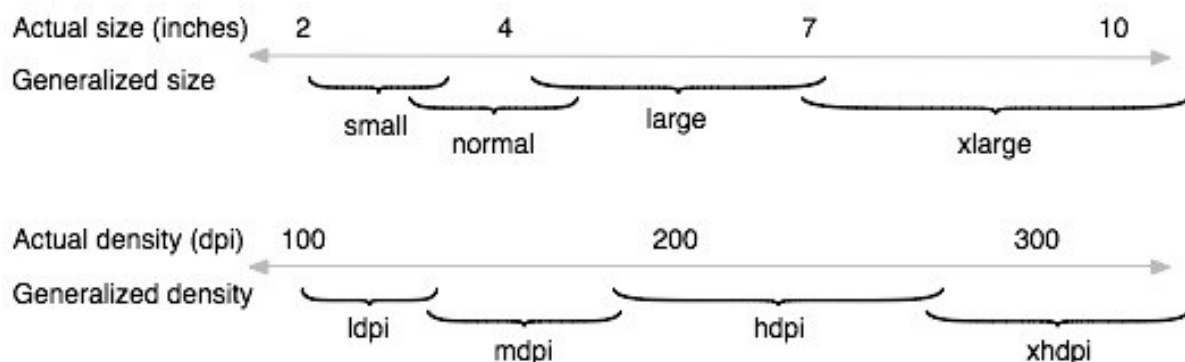
Figure 15 – Approximate map of *Android* devices sizes and densities to generalized sizes and densities [7]

Layout-specific XMLs are stored in a folder whose name is a composition of the qualifier name and a specific range (e.g. *layout-large*). The same logic is applied to the other qualifiers, although in a different context. For instance, namely resolution or density qualifiers are usually used/merged with the *drawable* folder with the intention of separating one image into four, each one with a different resolution. Meanwhile, both orientation qualifiers are used with the *layout* and/or the *values* folders, depending on the goal. Finally, the localization qualifiers are mainly used in the *values* folder, since the latter is the one that holds UI related static data.

With *Android* 3.2, Google launched three new qualifiers to address the increasing different size screens available in the market and improve user support, namely: *sw<N>dp*; *w<N>dp*; and *h<N>dp*. "N" is a user-defined constant (e.g. 600, 720 or 1024), corresponding to the desired customization. These three qualifiers have different meanings. The "sw" represents "smallestWidth" and it can be

---

[7] source: http://developer.android.com

used when we want to make sure that certain UI XMLs are used if a device has at least N dp's (Density-independent Pixels) in the smallest of its width. The other two prefixes ("w" and "h") mean available width and height, respectively. *Android* will chose them if the present device has a minimum of N dp's of default width or height respectively. The difference regarding the previous one is that in this case both width and height can change, depending on the orientation, while the smallest width does not.

5.3 – Android persistency

There are several ways of persisting data locally, which are offered by default in *Android*. If we have a considerable large data set that may be organized according to the relational model, then the *SQLite(Owens and Allen 2010)* database management system may be used. If it is simple data (e.g. application user configurations, temporary data or other small data sets) then the *SharedPreferences* class can be used. If we need to persist more specific and simple data like a *View* state (which within *Android* is almost mandatory, due to orientation changes) or if the data is not shared across the application, instead of using the *Shared Preferences*, we can use the *onSaveInstanceState()* method available in activities and fragments. This method enables the user to persist data in a bundle, tied to the activity or fragment state. This method should be used with caution, since its calls are done automatically by the system, according to a lifecycle that will be explained ahead.

5.3.1 – Persistency

Regardless of the technologies and platforms involved, the database management system should provide transactional semantics. Transactions are bundles of CRUD operations to be execute against the database as a single logical unit of work and, as such, treated in a coherent and reliable way, independent of other transactions. The properties that the transactional semantics must fulfil are the ones commonly called as A.C.I.D. (Grehan 2006):

- **Atomicity** – the components of a transaction must execute in an all-or-nothing fashion. For example, if a transaction involves deleting 4 objects, then those 4 objects must be deleted as they were a single object;
- **Consistency** – operations on the database move it from one well defined state to the next, with no intermediate states visible. For instance, if an object references other objects and vice-versa, in case of its deletion every other object must also be updated to a state where the deleted object does not exists. Failing to do it would result in a, *referential integrity failure*;
- **Isolation** – multiple ongoing transactions are unaware of each other. So, if two users attempt to modify the same object simultaneously, the database must implement some mechanism for serializing their access to the object, so that neither user's work interferes with -- or even 'sees' -- the others;
- **Durability** – once a transaction has been 'committed' to the database, its work is not lost, even in face of a hardware or software failure. So, if a user executes a transaction on the

database to delete 3 objects, and the system crashes in the process of deleting the second object, then, when the system is rebooted, the database will recover itself, including finishing the pending transaction.

5.3.1.1 – Persistence management

Since we need to persist data, several alternatives were assessed to consubstantiate our choice, as follows:

Flat files

The most basic approach of persisting data, is a plain text or binary file. This approach maybe suitable in cases where there is a very small amount of data and there is no structural relationships between the records (Wikipedia 1994). This is obviously not the case for BIS apps, where we will have to deal with highly structured data and its size may be relatively large, depending on the domain.

Network databases

Network databases appeared as an attempt to improve the already existing hierarchical databases, by adding the possibility to model many-to-many relations between entities. Each record may have multiple parent and child records, forming a generalized graph structure. This property applies at two levels: the schema is a generalized graph of record types connected by relationship types and the database itself is a generalized graph of record occurrences connected by relationships *(Wikipedia 2012a)*.

Network databases, now dubbed "*graph databases*", are said to be very efficient and became a mainstream research topic[8] mainly due to the need to process social network data, as well as other web-based networked info. However, we could not find yet such type of databases for *Android.*

XML databases

An XML database allows data to be stored in XML format. These data can then be queried, exported and serialized into the desired format. XML databases are usually associated with document-oriented databases. Two major classes of XML databases exist: (i) **XML-enabled** – these may either map XML to traditional database structures (such as a relational database), accepting XML as input and rendering XML as output, or support native XML types within the traditional database; and (ii) **Native XML** (NXD) – the internal model of such databases depends on XML and uses XML documents as the fundamental unit of storage, which are, however, not necessarily stored in the form of text files *(Wikipedia 2012b).*

Regarding the second type (NXD), no database management system could be found for *Android*. Meanwhile, the first database type (XML-enabled) is much used in Android, but using the cloud for persistency (i.e. it requires internet to access the data), instead of local storage. In the server side we must have a system able to interpret and create the XML files, but the actual storage system (server side) may not be XML based. Two very popular libraries that are used in *Android* as support for this

---

[8] http://mashable.com/2012/09/26/graph-databases/

type of system, are: JSON (Crockford 2013) and the GSON (Singh et al. 2013). Instead of XML format, the files are in the JSON format. GSON is a project that enables to convert, in a very easy way, any Java object to the JSON format. In 2013 Google released the JSON API in Google Cloud SQL(Google 2013b) which is a service that offers a full relational database management system. This would make very easy the transference of relational objects between the client side (Android) and the server side (relational database in the cloud), and implement the CRUD operations, since the service is already prepared to receive these operations within the JSON format, with all the required key features, referenced in section 5.3.1 – Persistency, implemented automatically (abstracted).

Relational databases

Relational database management systems (RDBMS), which are based on the relational model as introduced by (Codd F. 1970), are the most widespread ones (Wikipedia 2012a). The default persistency engine offered by *Android* and most mobile solutions is *SQLite* (Owens and Allen 2010; Pocatilu 2012). *SQLite* supports a subset of SLQ (ISO/IEC 9075-1 to 9075-14), the most widely used query language in RDBMS. Since our target platform is based on *Java,* an object-oriented language, there would be an object-relational impedance mismatch if we choose a RDBMS. That mismatch arises from the different type systems, as follows:

- essential OO programming concepts, such as objects, inheritance and polymorphism are not supported in RDBMS;
- the notion of information hiding in an OO language (i.e. visibility classifiers (public, private, etc.), is disregarded in RDBMS;
- the available data types are different (e.g. a single type like the String in object-oriented languages may have several primitive types in SQL to correspond to);
- OO enumerations have no equivalent in RDBMS and must be mimicked through an auxiliary table and a foreign key.
- class instances (objects) have an implicit unique identifier (object id), while table instances (tuples) require a primary key to grant unicity;
- relationships among classes are implemented through embedded attributes, while auxiliary tables are required in OCL, as well as foreign keys.

As an example, for a many-to-many relationship, while in an object-oriented approach (Figure 16) each type (class) holds its relationships itself as references, in a SQL based relational model (Figure 17) a third table is needed to hold the relationships. Notice that any type of data must be also converted to the adequate SQL type.

Luís Silva – July, 2014
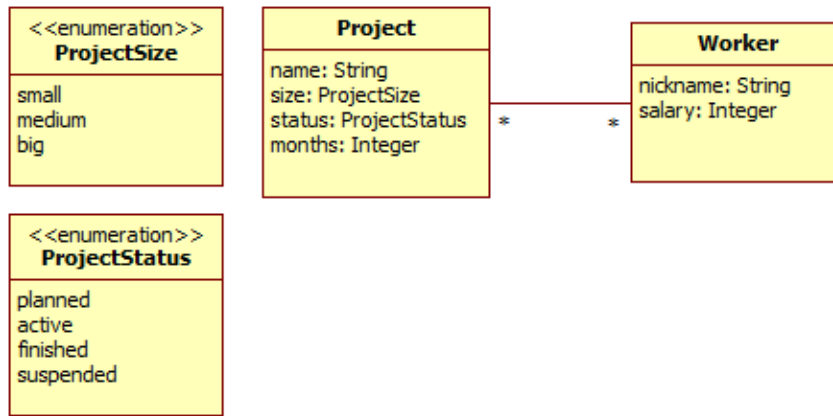
Figure 16 – Object-Oriented Model many-to-many relationship with UML example
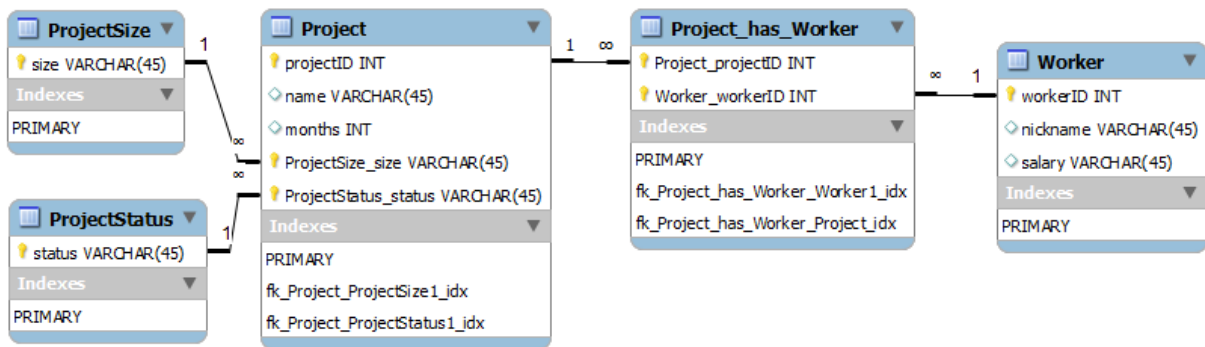
Figure 17 – Relational model many-to-many relationship with SQL example

The aforementioned facts make this type of persistency solution hard to implement. To mitigate them, most relational databases currently offer ORM (object-relational mapping) capabilities. However this type of solution also comes with a cost, like (Grehan 2006) said "(…) *an RDBMS adds space and time overhead to the application* (…)".

Object-oriented databases

In OO database management systems (OODBMS) transient objects used in OO programming are stored directly, therefore discarding complex mappings and transformations. Usually OODBMS use the same model employed by the application programming language. This means that references among objects are stored along with the objects themselves.

Objects-relational databases

OO databases (ORDBMS) are a hybrid of the previous two approaches (Wikipedia 1995). We could not find any ORDBMS for *Android*, but there are other alternatives. In the ORM libraries for *Android* section we present several solutions which, in combination with the provided *SQLite*, allow obtaining an ORDBMS.

5.3.2 – ORM libraries for Android

To avoid repeated work and also to facilitate code generation, we have surveyed existing *ORM* solutions for *Android*, as represented in Table 7. Although there are more solutions available, like *AndrORM* (Giese 2012) and *activejdbc* (Polevoy 2012), these ones seem to be the most popular and more mature. After analyzing each one, we concluded that none fitted well our goals, since they hold some important limitations. Some of the latter were: no inheritance support, no many-to-many relationship support, or transactions support. Those limitations (shown in Table 7), along with the aforementioned costs of using ORMs, made us consider an alternative solution, in our case the *DB4O* OODBMS (Versant 2013b).

Table 7 – Available ORMs solutions for Android

Project		ORMLite	ORMAN	greenDAO	jpa-android
Reference		(Watson 2013)	(Alp Balkan et al. 2012)	(Junginger and Dollinger 2013)	(Junior 2011)
Annotations		JPA or custom	JPA-like	N/A Codegen	JPA
Model specification		Direct in code (through annotations)	Direct in code (through annotations)	Specified in outside project (Schema objects) that will generate classes to be used in project	Direct in code (through annotations)
Database Operations		DAO or Entity	Entity	DAO or Entity	Entity
License		Open	Apache2	Apache2	Apache2
Age		07-01-2010	14-02-2011	4-08-2011	4-08-2011
Mapping Capabilities	To one	Yes	Yes	Yes	No
	To many	Yes	Yes	Yes	No
	Inheritance	Only for non-entity classes	No	Only for non-entity classes	No

5.3.3 – The DB4O OODBMS

As we can see in Figure 18, NoSQL databases management systems only occupy 5% (others) of the market share versus the large 95% of market share occupied by RDBMS. However, the expected growth rate by 2015 for NoSQL databases is the largest one (Figure 19). A good evidence of this interest is that Google cloud datastore[9] already offers a NoSQL database to its users, affirming that it

---

[9] https://cloud.google.com/products/cloud-datastore/

is easier to work with. So we decided to look for an OODBMS that would work on *Android* and we found the DB4O.
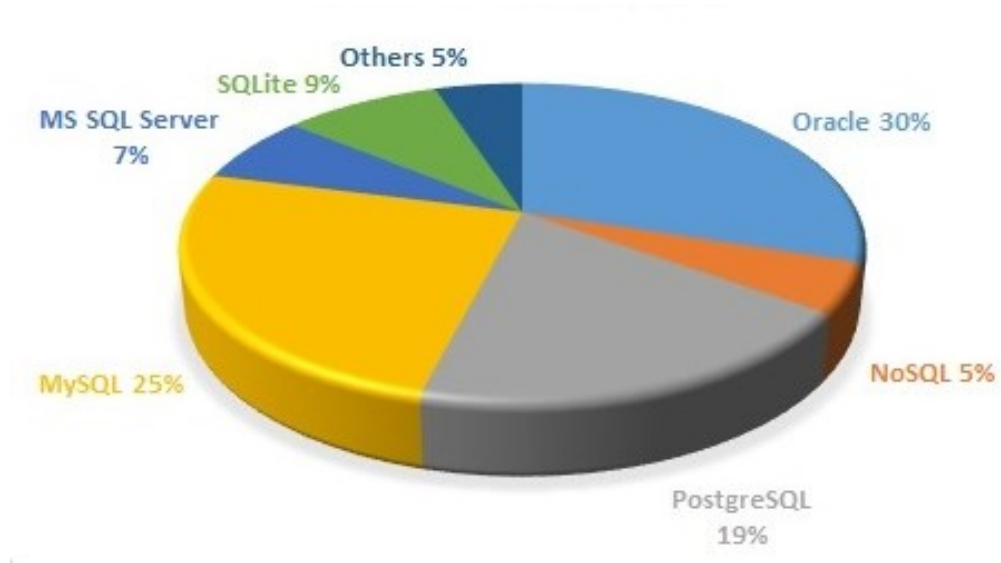
Figure 18 – Database engines – market share [10]
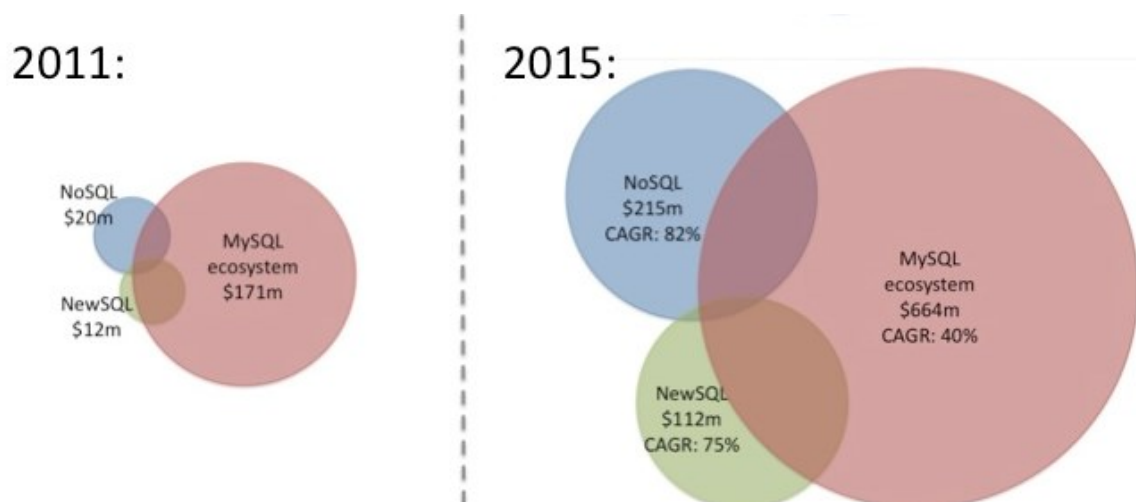
Figure 19 – MySQL, NoSQL and NewSQL compound annual growth rate [11]

The main features offered by *DB4O* are the following: (i) is an embeddable (through an API library) open source OODBMS for Java and .NET languages, and therefore is supported in different platforms and systems; (ii) has client-server mode, which allows message and data exchange between the client and server sides much more easier; (iii) is a true OODBMS, which means that the class model is the same as the database schema; (iv) supports transactions with the ACID properties and commit-recovery on system failures ; (v) and finally is a non-intrusive system which makes the implementations seamless persistent (i.e. it does not require to specify which classes are going to be

---

[10] source: http://www.vertabelo.com/blog/jdd-2013-what-we-found-out-about-databases
[11] source: http://blogs.the451group.com/information_management/2012/05/22/mysql-nosql-newsql/

Luís Silva – July, 2014

used and how they are used, as other implementation do, for instance by using annotations in the business classes). *DB4O* is supported by *Android*, by means of the *DB4O* Java library.

Table 8 – Db4o versus SQLite – main feature comparison

	Features				
	Seamlessness	Transactions	A.C.I.D.	Maximum Size	Client-Server support
SQLite	No	Yes	Yes	2GB	No
DB4O	Yes	Yes	Yes	256GB	Yes

As Table 8 shows, DB4O not only has all the features that *SQLite* has, but still offers more, namely: (i) a bigger maximum limit size for the database, (ii) client-server support and (iii) seamlessness. Database size is very important for BIS apps, since they usually work with large data sets. DB4O supports *BLOB* (binary large object) storage, therefore allowing to store media type data which consumes a lot of storage space without affecting querying performance. Thus, since we are targeting mobile devices which usually offer by default media content capturing hardware capabilities, such a limit in size may be proven as a very important feature when BIS apps require multimedia contents. Regarding Client-Server support DB4O offers an API with the ability to directly connect to the database on the server and also to exchange messages between the client and the server. Such features are essential when creating a BIS app since, as it was aforementioned, synchronization support is a requirement for allowing both offline and online modes in alternation.

Regarding seamlessness, the best way to figure out its improvement on understandability is by looking at an example as the one in section DB4O vs SQLite – seamlessnessDB4O vs SQLite of the Appendix.

For querying purposes there are several alternatives: the *Native Query*, *Query by Example* and *Simple Object Database Access* (SODA) *Query*. *SODA Query* is faster, and is our alternative although being the less "object oriented" one; *Native Queries* are the closer approach to the OO approach but are much slower, since "*Under the hood DB4O tries to analyze native queries to convert them to SODA*" (Versant 2013a). Finally, we do not use *Query by Example*, due to the dangerous nature of losing references. *Android* has garbage collection capabilities, so references to objects may be accidently removed, for instance, in an orientation change event. Due to this nature, we must temporarily persist instances values and since every object will have an id, performance wise, it is better to persist just the object id (using the provided temporary *Android* system) and query by it on restoration, instead of the whole object and query it by example. The latter is also the reason, regarding our choice for SODA, since we are able to retrieve objects by the id, using a static singleton class with SODA built-in methods that search only by a given class type and id, we avoid moving any database regarded code to above layers. We still do not disregard the other querying capabilities, since for more specific filtering approaches they still may prove to be the best option, both for the generative approach and for code understandability ⚙🧑 sake. Table 9 shows an example of these three query types.

Table 9 – DB4O different querying capabilities example

	Example – query of every pilot, named "Michael Schumacher"
Native Queries	```List <Pilot> result = db.query(new Predicate<Pilot>()
{
public boolean match(Pilot pilot) {
return pilot.getName().equals("Michael
Schumacher");
}
});```
Query by example	```Pilot proto = new Pilot("Michael Schumacher", 0);
ObjectSet result = db.queryByExample(proto);
listResult(result);```
SODA query	```Query query=db.query();
query.constrain(Pilot.class);
query.descend("name").constrain("Michael
Schumacher");
ObjectSet result=query.execute();
listResult(result);```

5.4 – Android Patterns

Navigational patterns, UI design patterns and code structure patterns are enforced by Google while developing *Android* applications. Of all the researched patterns, none made so much impact as the *ListViewHolder* pattern, due to the performance increase of list views (Google 2013a; Guy and Powell 2010). Another set of reference patterns was also studied to provide better extensibility, namely the *Observer* (Wikipedia 2013d) and the *Command* (Wikipedia 2013b) patterns. In 6 – Generated apps structure we explain how we implemented those patterns.

5.5 – Android Fragments

Fragments were introduced in Android version 3.0. Their primary goal is to provide a more dynamic and flexible way for UI design, namely in larger screens, as advocated in the "*Building a dynamic GUI with fragments*" section of (Google 2013a): *"To create a dynamic and multi-pane user interface (…) You can create these modules with the Fragment class, which behaves somewhat like a nested activity that can define its own layout and manage its own lifecycle"*. In the more recent versions (version 4.x) it still meets that goal and is also used for proper code separation, either for UI design or not. Besides code separation, using fragments may also have a great impact in performance. Fragments have a different lifecycle from the activities, which create and hold the fragments. Consider, for instance, a configuration change like screen rotation, which forces activities to restart (destroyed and created again), therefore also triggering the corresponding fragments to restart. If we set the fragments to automatically retain their instance (setRetainInstance(true)) we can reuse the fragment, for instance, on orientation changes, therefore saving all the deletion and recreation (which may include database querying) process. In other words, the fragment still restarts with the

activity (this action is essential for example if a UI change is needed) but we skip the `onCreate()` method in the fragment lifecycle, since variables are maintained. This is why when developing for *Android* it is essential to understand the components lifecycles, discussed ahead.

5.5.1 – Static versus dynamic approach

One of the most important decisions that a developer must make, before developing an *Android* application, is choosing a static or a dynamic approach, although the latter may be mandatory in specific scenarios. An easy way to explain both approaches regards the usage of resources. If it is a static approach, there will be much more data in the resources, including fragment definition in the UI XMLs, but it will be much less code in the controllers (*Activities*). In contrast, the dynamic approach is much harder to implement, and has more code, but it gives the developer greater control and much more variance possibilities, for example, when there is a need for a change in an already working implementation (maintenance). While in the dynamic implementation we do not have to change the implementation structure, in a static one we might have to, depending on the required changes. For instance, consider a UI which has two fragments dividing the screen (let us call them fragment 1 and fragment 2), and we want to add to this project a new behavior, namely the ability to show just the fragment 1 or 2, depending on the user action. In the presence of a dynamic approach we just need to catch the user action and hide the fragment 1 in order to fill the screen with the fragment 2 or vice-versa. On a static approach, the screen would be left with a blank space (the one occupied by the "old" fragment). The solution for this scenario would be creating 3 different static scenarios, one where we have the two fragments, another with just the fragment 1 and another with fragment 2. As it can be seen, the same problem may be very easy to solve or very hard (in the sense of writing much more code) depending on the followed approach. Nevertheless, the static approach tends to be faster, since the static UIs are compiled, while the dynamic approach implies run-time rendering. This may affect application performance in situations such as screen rotation.

5.6 – Android Lifecycles

In Figure 20 and Figure 21 we can see the lifecycle of activities and fragments, respectively. There, the arrows represent the flow based on events, the rounded nodes represent the component state and, the rectangles represent the methods called upon the events. As briefly shown before, it is very important to understand how these lifecycles work, in order to avoid bugs, time consuming mistakes and to achieve good coding structure and performance. For instance, if we are working with one activity, as we can see in Figure 20, we could not determine exactly where methods would be going to be called and when, since the system overpowers our application. For example, if an activity is in a paused state, we do not know when it will gain focus again, if it is going to start again on the `onResume()` or on the `onCreate()` methods. This leaves us with one question: what code can we put in each method to guarantee that the activity is always going to work and do it in the best possible way. In other words, we could code everything on the `onResume()` method to guarantee that our

code would always be executed in every possible scenario, but that would make the application more confusing and probably with worse performance, since it would repeat steps even when not required. As we can see in Figure 21, the same concern logic is applied for fragments, despite some minor differences in the corresponding lifecycle. Yet another concern, that a programmer must be aware when working with fragments, is the activity and fragment lifecycles intersection, as shown in Figure 22. Finally, as we can see in Figure 23, the programmer must also have into consideration, in case the implementation requires it, activity state recreation or recuperation. As aforementioned it is possible to instruct *Android* to retain fragments instances in order to avoid the repetition of the creation process. This is one more concern of how to properly arrange the implementation. That illustrates the possible advantages of a static approach, since it masks the programmer from all these problematics.
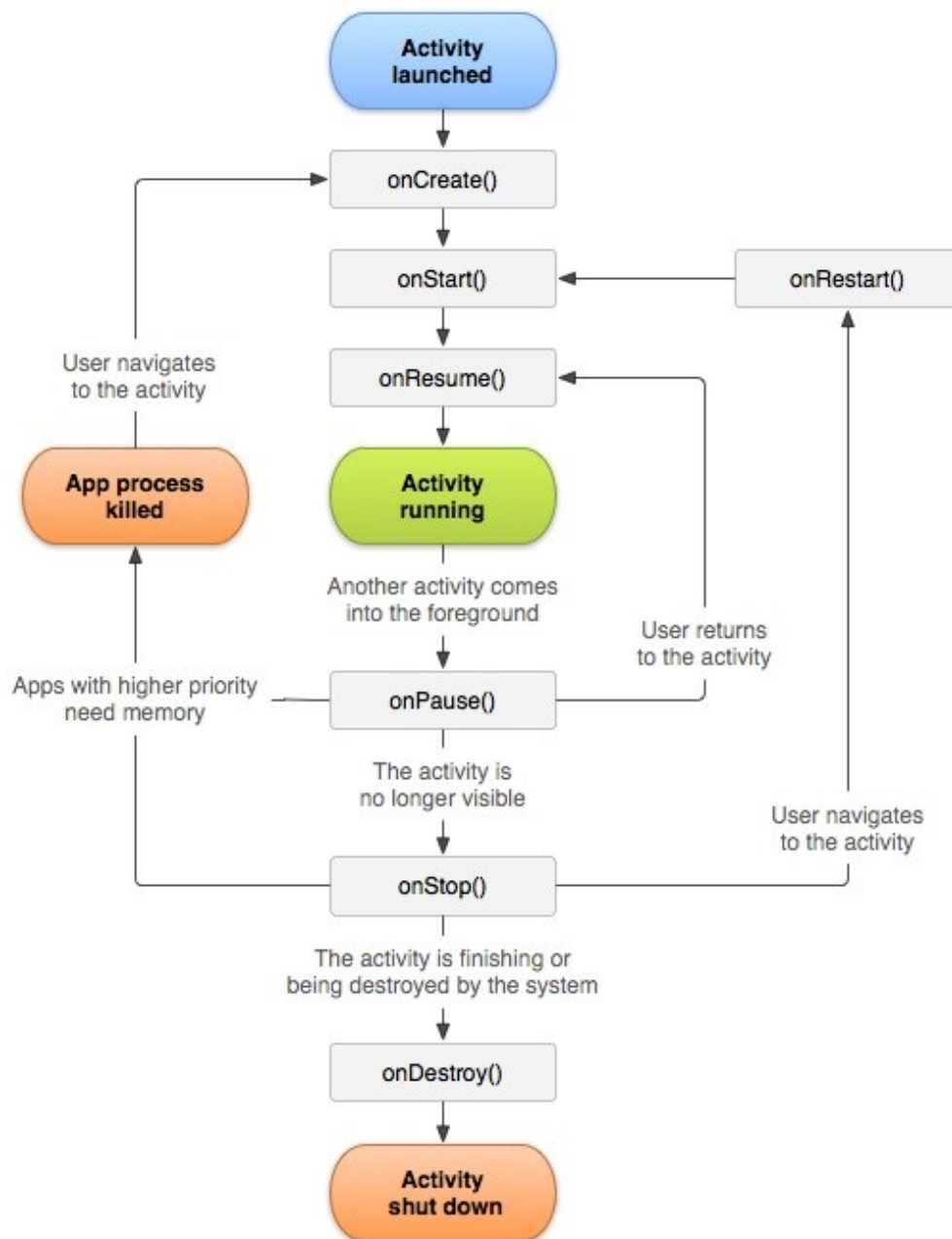
Figure 20 shows many possible scenarios (e.g. if the device receives a call, another activity comes into the foreground). Notice that there is not a need to identify what disrupted our activity, but we must be aware of such a lifecycle, in order to properly make use of the available methods.

---

[12] source: http://developer.android.com/reference/android/app/Activity.html
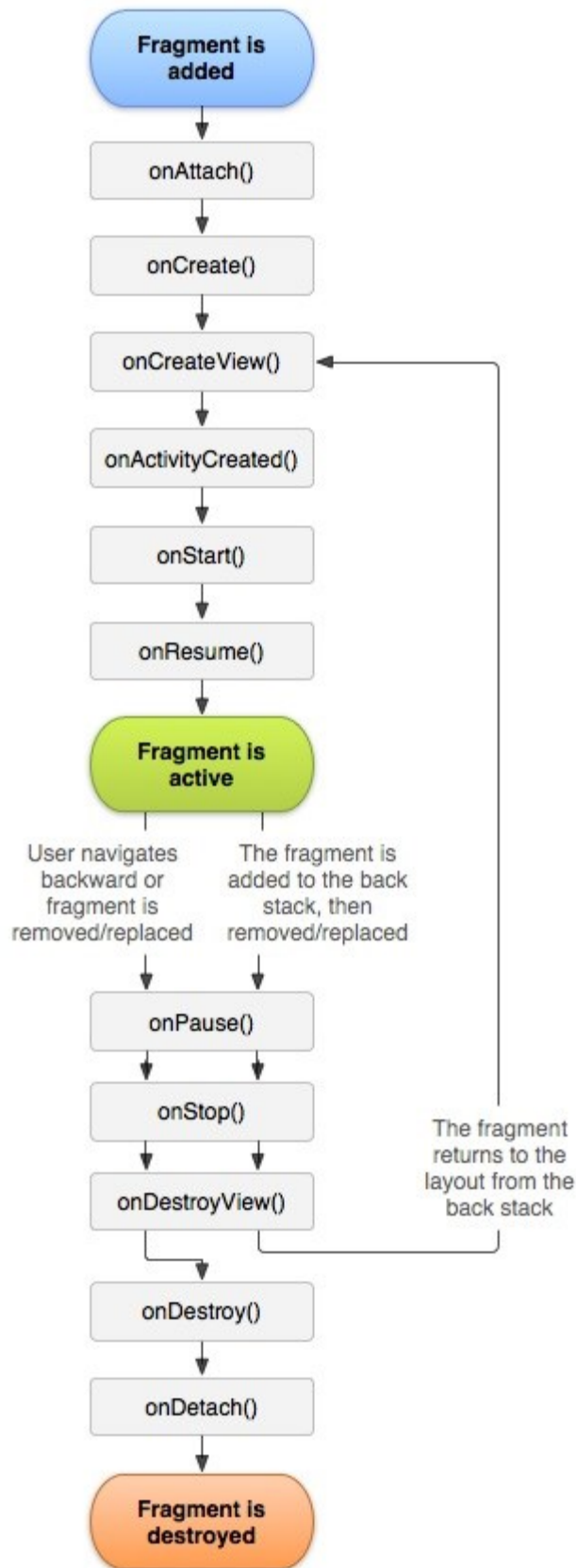
Figure 21 – Fragments lifecycle[13].

---

[13] source: http://developer.android.com/guide/components/fragments.html

Luís Silva – July, 2014

Fragments lifecycle (Figure 21) is similar to the one of activities. Notice that in this scenario we have two possible ends: (i) the left one, where the fragment gets destroyed; (ii) and the one on the right, where the fragment is reused. On (i), the fragment follows the typical lifecycle, i.e. is created, used, and removed from memory. On (ii), the fragment is reused, skipping the attachment to the parent activity and creation process, thus saving in possible reads or processing data on the latter steps. An example scenario of the latter is a screen rotation event, where the view may change (i.e. the layout must be updated and therefore the views must be recreated) but there is no need to change, for example, the object that is being used to fill this view.
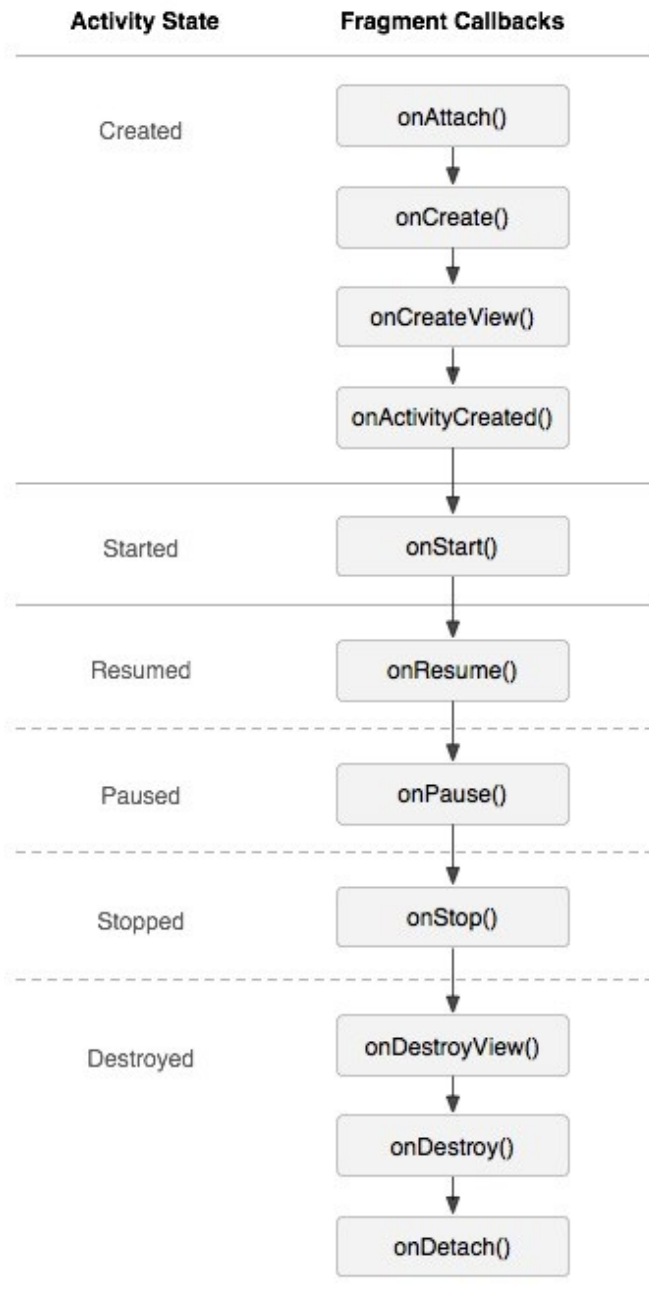
Figure 22 – Fragment lifecycle intersection with activity state[14].

---

[14] source: http://developer.android.com/guide/components/fragments.html

Luís Silva – July, 2014

Figure 22 shows the intersection between the lifecycles of activities and fragments, that must be understood to avoid bugs and reach good code structure and application performance. For instance, if a fragment needs to know the identity of his holder activity to do an action, it can only do it, for example, after its view is created. This can be seen as a potential place where a developer can make mistakes when trying to control the normal flow of an application, namely when deciding if a certain view should be visible upon a rotation. Since we would be dealing with views, it is normal to execute such functions inside the `onCreateView()` method but, in this case, we could not do it because the activity was restarted and it would still not be created. The same logic is also applied to the activities. As it could be seen in both Figure 20 and Figure 21, the methods required to execute certain tasks must be carefully chosen, since we must follow a certain call order, to avoid errors.

Figure 23 –Activity recreation lifecycle[15].

In the recreation lifecycle in Figure 23, two more persistence dedicated methods are supplied to persist states temporarily. The robot in this scenario represents the app icon. The user would have pressed the home button and therefore, the app would be "minimized", which in *Android* is the equivalent to close the app in its current state. The user can go back to the paused app by pressing its icon (in this case the robot icon). The `onSaveInstanceState()` is where we persist the data and the `onRestoreInstanceState()` where we restore the data. Again, we should notice that the restore method is only called after the `onCreate()` method. The latter is very important when creating an *Android* application, since we cannot expect to have access to persisted data before the activity creation.

Such interactions are essential to achieve good performance by avoiding repeated tasks, for instance, avoiding repetitive querying the database due to screen rotation, which is very common for an everyday *Android* user.

---

[15] source: http://developer.android.com/training/basics/activity-lifecycle/recreating.html

[This page was intentionally left blank]

6 – Generated apps structure

6.1 – Generic architecture

We follow a template like generation, based on defined navigation principles. Since it is our goal to decrease development time by means of a generative approach, patterns were studied so that the generated implementation offers a low, or more loosed, coupling and thus offer better maintainability and reusability.

To achieve that goal, the MVVM (Model View View-Model) was the base architecture chosen. As aforementioned, *Android* is based on the MVC pattern, but does not force it. The MVVM is a more recent pattern proposed by John Gossman (Gossman 2005) and based on Martin Fowler's Presentation Model (Fowler 2004). Both feature an abstraction of a View, which contains state and behavior. The difference is that Gossman presented the MVVM as a standardized way to leverage core features of the Windows Presentation Foundation (WPF) and Silverlight, in order to simplify the creation of user interfaces. Besides not being an *Android* specific pattern, both are based on the MVC pattern: *"The Model is defined as in MVC; it is the data or business logic, completely UI independent, that stores the state and does the processing of the problem domain. The Model is written in code or is represented by pure data encoded in relational tables or XML."* (Gossman 2005). Some adaptions had to be made in order to fully implement our solution in the *Android* platform. In Figure 24 we can see the basic representation of the MVVM and how and which data is passed between layers, notice that by comparison with Figure 11 (MVC) the main difference is that the view layer does not communicate with the model layer. In the MSDN[16], a more detailed explanation of Figure 24 and its implementation is available.

---

[16] http://msdn.microsoft.com/en-us/library/ff798384.aspx

Luís Silva – July, 2014
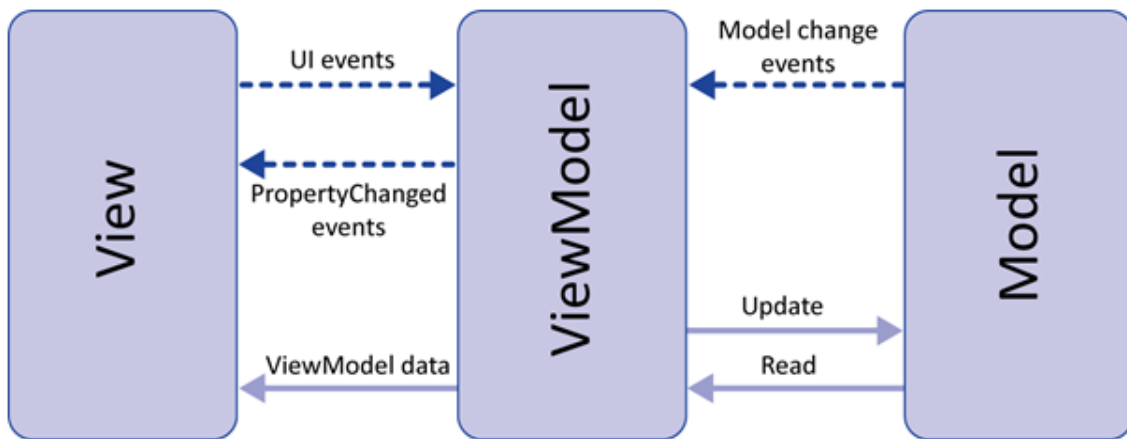
Figure 24 – Model View View-Model[17]

The UML component diagram in Figure 25 presents, a generic architecture of the generated applications: the client-side (*Android* platform) and the server-side (*Java* platform). As in the MVVM, there is no communication between the GUI specification component (*View* layer) and *Domain logic* component (*Model* layer), as opposed to the typical Model-View-Controller. A persistency helper (*Persistence manager* component) is also provided which contains generic methods that return persisted data, based on the data type. If more specific data is required, a set of methods are also available which can filter by a given data type, attribute name and value/constraint. The generated *Java* server application holds the shared database and a Server manager component, whose only function is to start a DB4O server instance, available through its API, with the given settings, in order to trigger server-side listening and acceptance or refusal of incoming connections. The *Synchronization manager* component is responsible for all the synchronization process and therefore is the only one that uses the client-server API available in the DB4O to establish connections to the server. In order to synchronize the data, the latter accesses type dedicated methods offered by the *Domain logic* component (*Model layer*) by means of an interface thus, it is not required to know or instantiate any domain logic. To know the data that is supposed to send/synchronize it accesses the *Persistence manager* component and reads its own data specific type (*Transactions*) from the database which in turn holds, in an ordered by creation manner, domain specific data type.

---

[17] source: http://msdn.microsoft.com/en-us/library/ff798384.aspx
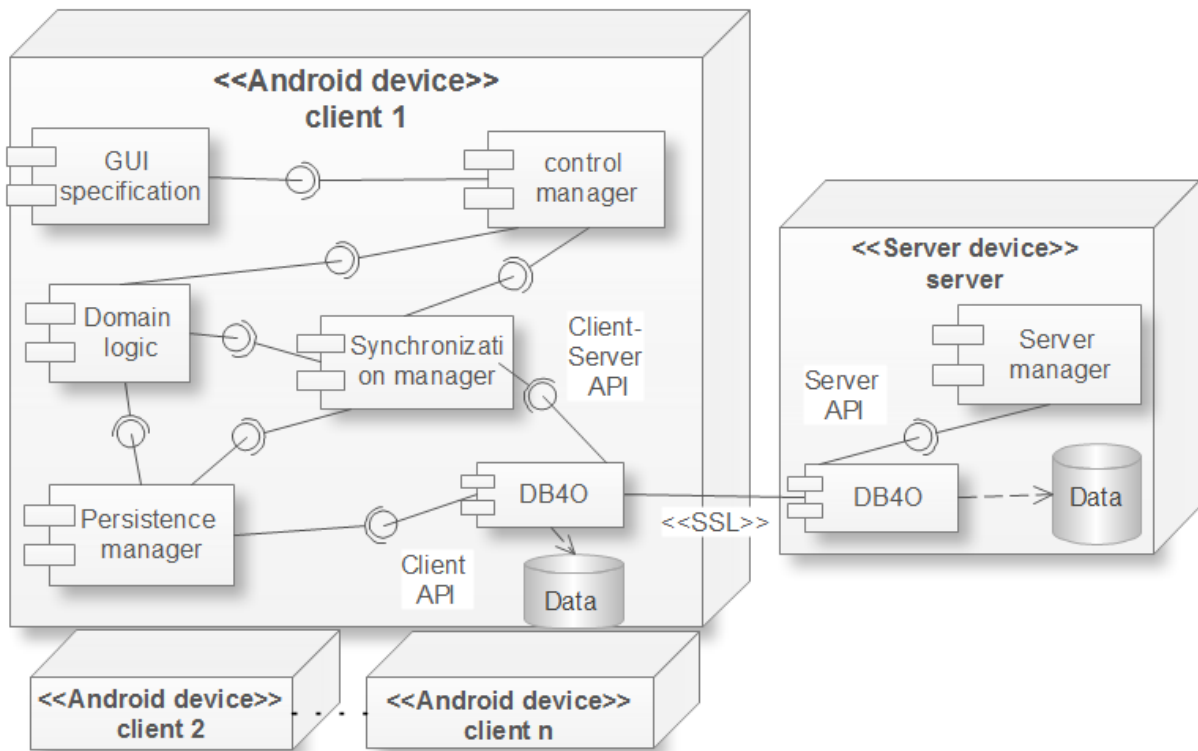
Figure 25 – Generated applications architecture (UML component diagram)

Figure 26 shows our implementation based on the MVVM in a more detailed fashion, where the persistency layer is shown, as well. Next we will demonstrate, in a more detailed manner, how each layer is implemented and how it fulfils its concerns/responsibilities, namely regarding: (i) model requirements; (ii) mobile requirements (i.e. screen size, density and rotation adjustments); (iii) persistency requirements; and (iv) our implemented synchronization system.
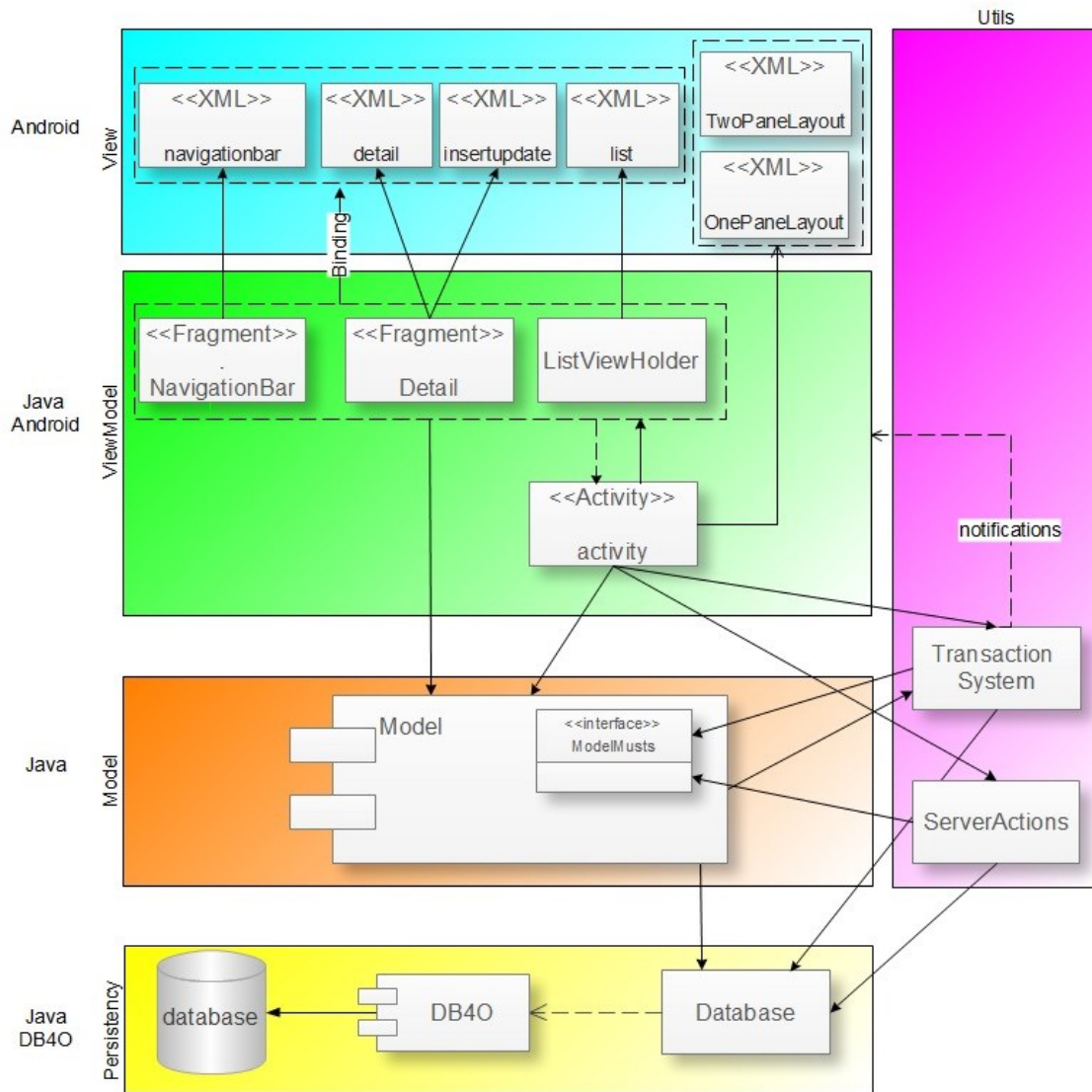
Figure 26 – Generated client-side apps main architecture

6.2 – View Layer

6.2.1 – Concerns

The View layer is responsible for guaranteeing screen size and resolution adjustments, defining the UI components for each object type and for any configuration (portrait and landscape orientations), and finally for defining basic UI action states (i.e. simple state changes like button pressed). To ensure proper display on different screens types, Google provides a set of proven best practices (Google 2013a), such as using `wrap_content`, `fill_parent`, or `dp` units when specifying dimensions in an XML layout file and supplying alternative bitmap drawables for different screen densities. Not using hard coded pixel values and *AbsoluteLayout* (deprecated). Some additional organizational techniques were applied to ensure better maintenance or change support, as described in the following subsections.

52

6.2.2 – View Layer structure
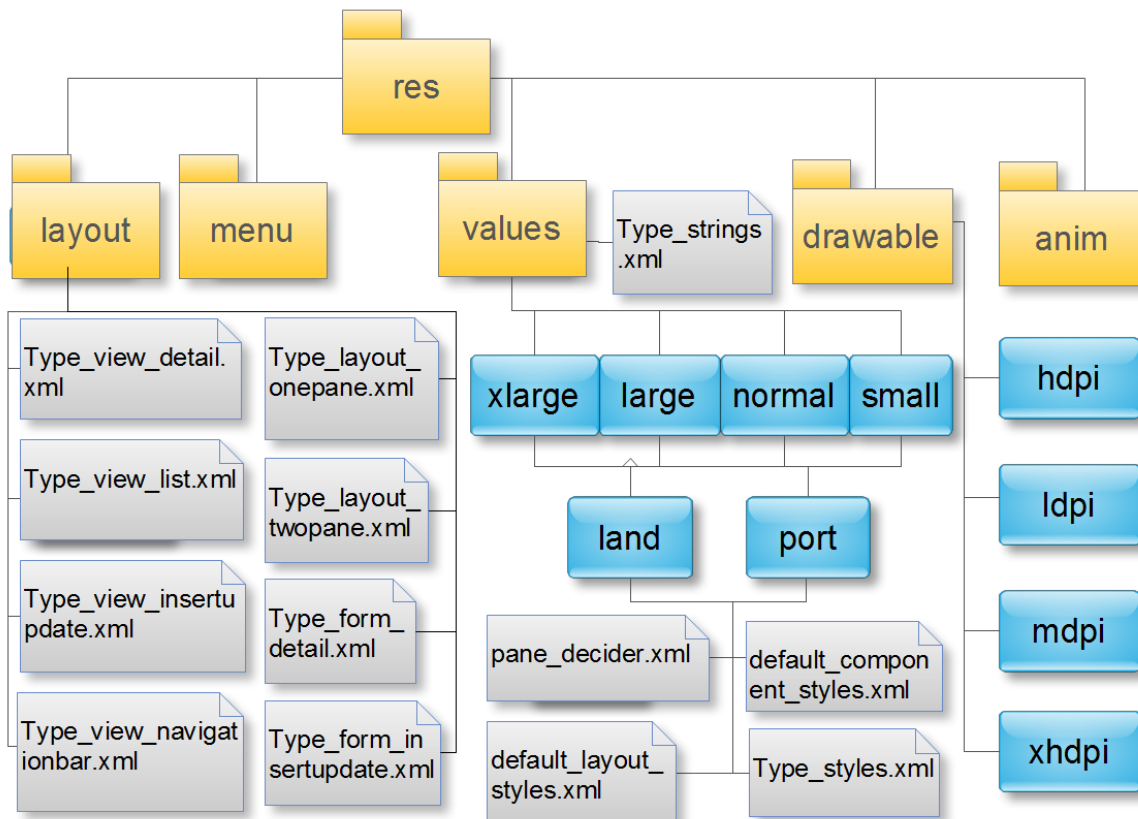
Figure 27 – Proposed *Android* View Layer Structure

The view layer is composed by general layout XMLs (one pane and two pane layouts), view XMLs (specific to object types) that describe the layout used in a certain space of the screen and finally the forms, the most specific XML type, which describe the layout of/and the components used for the specific data type of the objects. Regarding this layered structure, the general layout XMLs are independent, i.e. they merely represent the general layout. Any binding is done dynamically in the View-Model layer. On the other hand, the view XMLs are set statically. For instance, they will hold the form XMLs files and we is define statically which forms they hold. The form XMLs are defined as mergeable XMLs, which means that they can be included in other XMLs, namely the view XMLs, thereby promoting reuse. All XML files are placed in the *layout* folder, as shown in Figure 27, and required by *Android*. The *menu* folder will hold XML files that describe the *ActionBar*, one for each XML for each different *ActionBar* setting. The *anim* folder is used to store animations, but note used so far. The *drawable* folder is used to store basic action styling based XMLs files. For example, it contains an XML that will change the background colour of a view if the view is pressed. It is also used to store media related files for density purposes as it will be explained next. Lastly, the *values* folder is used to store raw data and in our approach as the base size and style controller like it is also explained followingly.

6.2.3 – Reaching several screens sizes and densities

All the screen size and density control support is handled in this layer. It could also could be controlled in a dynamic way in the *Java* code but, as shown in chapter 4 – Domain and GUI specification, *Android* offers a set of qualifiers to improve the construction and maintenance of this control structure. As also shown in Figure 27, we may apply these qualifiers in several ways. Instead of using the qualifiers in the *layout* folder, we use them in the *values* folder, and in the file "`pane_decider.xml`" we reference the proper layout file depending on the size qualifier. The same approach is used for specific component sizes, but this time we make every decision inside the files "`default_layout_styles.xml`" and "`default_component_styles.xml`", for example if we want to define the text size. For density adjustments the same will happen, i.e. we will have in each qualifier folder the exact same image or icon but with different densities. The difference here is that we do not need an XML to reference different images, since they are always the same. Therefore we have different images with the same content but with different densities but they must have the same name. The same also applies for both previous XML files and the used IDs inside them.

This approach avoids a lot of code repetition. For example, we avoid the repetition of the file "`<type>_form_detail.xml`" at least seven more times (the four size qualifiers in each orientation setting), since more sizes can be defined beyond those standard settings. As Figure 28 shows, when defining styles for specific UI components, like a *TextView*, we point to another XML file. This allow using first one XML file for describing the objects (in this case the form XML files). Any information that can be volatile depending on the screen size is set in the "`<type>_styles.xml`", which in turn inherits its styles from the "`default_component_styles.xml`". This last XML inherits the *Android* standard style chosen by us. For instance, for the *normal* qualifier we set the *TextAppearance.Medium Android* setting. Currently we only use this structure for text specific components since, as far as we know, these are the only ones that have changeable settings, besides width and height.
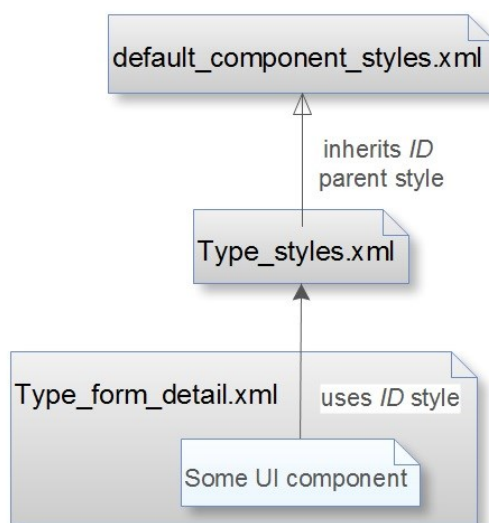
Figure 28 – UI components styles reference structure

However, as it will be shown, despite applying these good practices in our generation approach and therefore enforcing good coding, the amount of code generated for the *View layer* is, even with considerable small models, very extensive.

6.3 – View-Model

6.3.1 – Concerns

The *View-Model* layer is responsible for binding the data to the views, controlling in a dynamic way every configuration change that may occur, handling and validating the user inputs and acting as a "middle man" between the model and view layers (i.e. handles information exchange). This layer was sub-divided into a *binding layer* and a *control layer*.

6.3.2 – Binding Layer

The *binding layer* is composed by every class responsible for doing the binding between the abstract views (XML defined view) and the data types, i.e. filling each view component with the value of the given object type. Therefore, in our template approach there will be three main binding classes, each responsible for a different view: (i) the "`<type>DetailFragment`" responsible for the *Detail View*; (ii) "`<type>ListViewHolder`" responsible for representing the view of each item in the list (i.e. this class does not represent or control in any way the list itself only the view of each item); and finally the "`<type>NavigationBarFragment`" responsible for the navigation bar view. These classes also carry other responsibilities. For example, they also handle user input, can control visual effects and navigate to other activities. However, these actions are always restricted to the own view "action space" (i.e. the action must not involve other views outside the "action space"). For example, the "`<type>DetailFragment`" handles input validation and responds accordingly and the "`<type>NavigationBarFragment`" can start other activities (user navigates to another screen) which may seem like it is overpowering other views or activities, but in fact when we navigate to another activity all the views and activity states are saved, and restored when returned to focus. Therefore, it does not interfere with other views or activities, since each view and activity must be prepared for handling any state change on its own. Another responsibility that these classes hold is the set of listeners, namely upon buttons or the views themselves. While this behavior can, and should, be set in the view layer thus following more closely the MVVM, we have chosen to do it here (in the classes themselves). The rationale is that the *Android* framework only allows the activities to receive such action triggers. In fact, we could had set the listener statically in the *view layer*, which in turn could trigger a pre-defined method in this layer. However, since only activities could receive the calls, and not the fragment classes, or the list view holder, in order to achieve such a structure we would have to pass the call from the activities to the proper receiving classes. That would increase the coupling effect and would go against our goal of separation of concerns with loose coupling.

In conclusion, these classes set and remove their own listeners from both the view layer (i.e. views or buttons listeners) and the model layer (in order to receive object state modification (Observer pattern)), set and fill with data the static views defined in the view layer and/or remove them. They also provide methods to control themselves more easily. For instance, simple hide and show methods are generated in these classes, granting the activities easier control over their screen space. Finally they also can show any type of *Android dialog* and handle the response independently (e.g. error input messages).

6.2.4.3 – Control Layer

The main controllers are the activities. In our template approach each data type will have a dedicated screen, which is an activity, named "`<type>Activity`". These activities: create or destroy every class/fragment of the binding layer and set their parameters; serve as base communication between them, and if there is a cascade effect other layers[18]; control the *ActionBar* actions; and lastly and most important since the activities are the first to start, they are the ones that receive the arguments and decide which layout is best to display[19]. As shown in chapter 4 – Domain and GUI specification in order to control every behavior, while maintaining good performance and avoiding as much repetition as possible, the proper lifecycle methods must be chosen to each behavioral control, namely:

- `onCreate(Bundle savedInstanceState)` – this method is the first to get called in the lifecycle of an activity and it is called only once. It is called again when the activity is destroyed and restarted, for example due to orientation change (screen rotation). So it is here where we create and set the views layout instances to the proper fragments or, if the received Bundle is not null (activity restarted), we just set them and also set the activity variables state. Also in this method we check the activity received arguments and therefore it is here that we control our navigation principle, i.e. decide what type of screen we should present based on the navigation genders, if the activity should behave in a READ mode or WRITE mode.

- `onSavedInstanceSate(Bundle outstate)` – this method is used to store any data for later use (restore screen state). Is only called when the activity is destroyed without intention. For example, on screen rotation and also if in a paused state the activity may be destroyed by the system due to low memory.

- `onStart()` – this is one of the most important methods, since it must be here where the view settings must be implemented. For example, in our scenario the list may support long clicks or

---

[18] The creation of a new object is done in the "`<type>DetailFragment`" fragment which in turn is passed to the "`<type>Activity`" which is the one that triggers the action (insert, update or delete) by communication with the Model layer.

[19] In our approach when navigating to another screen/data type the layout chosen to display may vary depending on the target multiplicity namely, if the user navigates in a ToONE (multiplicity of 1) direction since it has only one possible object it does not make sense to show the list view therefore the one pane layout is automatically chosen.

not, depending on the device size, i.e. if we are in a two pane screen it will not, on the other hand if on a one pane screen it will support in order to show the detail view.

- `onResume()` – used only to place the restarted instance (used to know if the activity suffered from a restart effect, like a screen rotation) to false. This method was chosen since it is the last method to be called, thus granting us assurance that the restarted instance will be true (in case of a restarted) until "everything" is done (recreation and state restore where is possible), that is the activity will be ready to start receiving events.

- `onPause()` – not used.

- `onStop()` – not used.

- `onBackPressed()` – in our approach, as shown, instead of navigating to details screens/activities, we replace views. We use this method to gain control over the back button (present in every *Android* device) thus deciding its behavior.

- `onActivityResult()` – in this method we prepare the activity for each possible answer from other activities. In this case we prepare the activities for answers of both: association creation or cancelations types.

- `onDestroy()` – called when the activities are destroyed. We use this method to remove any previously set listeners.

By using the aforementioned methods accordingly, we can prepare every activity for every possible scenario. Since we are dealing with dynamic data and we want to reach every possible combination regarding screen rotation, we need to dynamically control these behaviors. For instance, in the worst case scenario, we could have for the same device the one pane layout in a portrait mode and when in landscape the two pane layout. In the `onStart()` method we know if the screen was restarted (for example due to screen rotation) therefore here we decide if we are going to show the detail view or not. This last type of control is performed by the activities in their screen controller.
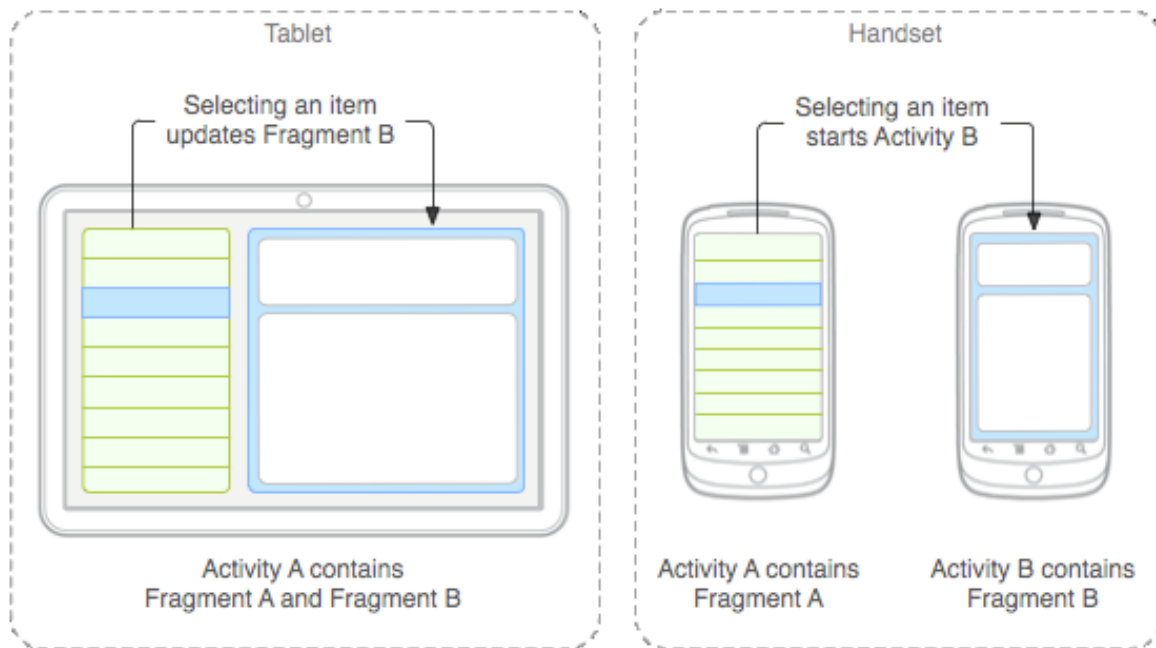
We decided to follow this approach, even if not following Google's recommendations for this case scenario, as shown in Figure 29, since it showed a greater changeability and behavioral control and a much easier implementation and therefore comprehension. We must also point out that we are dealing with dynamic and relational data and a lot of information exchange between different modules with CRUD operations as a requirement. For instance, in the last example, where we would pass from a two pane layout to a one pane layout, we do every check and change with very few lines of code. Since we are working in the same activity, everything is already set. If we followed Google's recommendations, we would have an entire new activity that we would need to prepare and possibly destroy in case we passed from one pane to the two pane layout. We would also have to pass a lot of information from one activity to another, since we would need to show the navigation bar on the detail screen (it shows data of the objects), or at least be aware of the data transference. Furthermore, but in this case for our navigational approach, as previously mentioned if we navigate to a ToONE gender, the list view is not required, only the detail view. Therefore, in this case scenario, we would have an activity A which only purpose would be to call activity B. If the activity B would be called directly, then the activity B must have a lot of repeated code, since it can be called in both READ or WRITE modes. This behavior would also hamper code understandability and generation, since it would fragment into different places what it should be done in the same method or place. For example, if an instance type is target of a ToONE gender type, in READ mode it only shows the detail view, but in WRITE mode it must show the list in order to give the possibility of choice. Therefore, activity B would repeat the code that analyses the received arguments for READ mode.

6.4 – Model Layer

6.4.1 – Concerns

This layer includes the POJO classes corresponding to the model classes, plus the ones that will hold the persistency dedicated methods. One of our main goals was to make use of OCL to describe a more "volatile" constraints checking. Since in an application lifecycle the base model structure has a minor chance to suffer changes we can consider the OCL constraints a more volatile type of constraints. Besides this fact and due to the large nature of the research we still do not check such constraints, but we have considered the UML associations, i.e. how an instance type relates to another, as simple model constraints. For each model class this layer will contain two *Java* classes: (i) the POJO class named with the same name as the class in the model, therefore `Type`; and the `Access` class, named as `<Type>Access`. The word *Access* was added since it is this class that mainly communicates with the persistency layer and implements the *ModelMusts* interface which contains the necessary methods to realize all the actions, i.e. local persistency actions, server synchronization actions and notification actions.

6.4.2 – POJO classes

These classes are a "pure" representation of those described in the model. The above layer (*View-Model*) will only use these classes to obtain information, set listeners or to command new actions, namely persistency actions, and they never call directly the persistency layer or the access classes, which are the ones that fulfil these actions. Therefore the POJO's will have: (i) every parameter and association instance described in the model; (ii) extra parameters to hold each association state (i.e. if the current association is considered valid), a general object state (i.e. valid or not valid) and an *ID* attribute (Integer) used as an identifier (for persistency purposes); (iii) getter methods (i.e. methods that retrieve information) for every parameter, association and states (previous point). Besides these it is also provided an *allInstances* method which returns all the objects present in the database, a type getter and lastly a specific by ID object getter (takes an ID as argument); (iv) to complement the previous getters methods the same set of setters methods (i.e. methods that set/change information) are also available, except for the *allInstances* and the *type* method. (v) lastly, it is also provided a *getAccess* method in case there is a need to access the *Access* class, and five persistency action methods namely, *insert, update, delete, insertAssociation and deleteAssociation*.

6.4.3 – Access class

The *Access* classes act as a bridge to the persistency layer. They hold the listeners and the respective `add` and `remove` methods. Regarding the persistency methods, the aforementioned five methods are provided and it is provided the, previously mentioned, five methods and it is in these

classes that every local action is done. Also four server dedicated methods namely `serverInsert`, `serverUpdate`, `serverInsertAssociation` and `serverDeleteAssociation`. These server methods are needed for synchronization purposes. For instance, we need to call all the setters' methods for an update action since the object is not in memory, therefore is not seamlessly update by the DB4O engine. Furthermore for insert purposes we need to verify and set the object state. Finally and most important, since is not in memory the references used in the server to represent the associations are different from the ones used locally, therefore we must replace them "manually" (i.e. simple object replacement would not suffice). The delete method is not necessary since it does not require any special treatment. Other methods are also provided for coding purposes.

6.5 – Persistency Layer

6.5.1 – Concerns

As aforementioned, the goal of this layer is to fulfill the persistency requirements required by the other layers.

6.5.2 – Database

For persistency purposes, instead of the default *Android* persistency engine (*SQLite*) the DB4O was chosen as discussed in section 5.3.3 – The DB4O OODBMS. As shown in Figure 26 (MVVM diagram), only one class is used for this layer, which is the reason of using DB4O in our approach, since this engine allow us to seamlessly persist data. Depending on the model some configurations must be implemented. Such configurations will affect the engine behavior, regarding the results upon usage of the normal CRUD operations. So, in order to make the DB4O act accordingly to the model, one essential configuration must be set, namely: (i) the *update depth*. For instance, inheritance trees and many-to-many relationships must have a minimum *update depth* of two, since a change of any object participant of such relationships will mandatorily affect another type object, namely its neighbour or child. The later configuration setting not only is important for performance, but also for regular operations that may involve any relationship. (ii) Lastly, and by default, we set all the domain types present in the model and therefore in the database to be indexed by the "*ID*" attribute, added by default. Since we will do all the queries based only on this attribute, we index all the classes by this attribute. By doing this the queries will perform faster.

6.6 – *Utils* – Support layer/package

6.6.1 – Concerns

In this layer/package is placed every class that will serve as support. For instance, to avoid code repetition in the caller class, to serve specific purpose algorithms like the classes used for synchronization purposes and other all-round purposes. Every class in this package should provide such service to the callers and still maintain itself fully independent (i.e. it should receive enough arguments to completely perform its task and return, if it is the case, the result without compromising the process flow of the caller).

6.6.2 – Generated support classes

As aforementioned there are classes that will be independent of the model namely, the classes present in the *Utils* layer. In Table 10 we can see all the classes present in this layer and their purpose.

Table 10 – Support classes

Class	Purpose
AndroidTransaction	Used to represent one *Android* transaction, i.e. it contains a list of commands, an `id` and if it is the case an error tittle and message.
Command	Class that represents a command.
CommandTargetLayer	Enumerated target layers.
CommandType	Enumerated type of commands.
DetailFragment	Interface to be used by the detail fragments.
FragmentMethods	Super interface with generic all-round fragment methods.
InheritanceListFragment	Used to show the possible navigational possibilities for inheritance scenarios (as shown in Figure 8 example).
ListAdapter	Generic list adapter. Can be used to create list adapters for any type of objects.
ListFragmentController	Generic list fragment. Can be used to create list fragments for any type of objects, with a set of defined settable rules.
ListViewHolder	Interface, with the required methods, used by the specific type list view holders.
ModelContracts	Class that contains methods that check the UML constraints given an object (single object or collection) and the cardinalities.
NavigationBarFragment	Interface to be used by the navigation bar fragments.
PropertyChangeEvent	Class that represents a persistency action event.
PropertyChangeListener	Interface used to communicate events.
ServerActions	Synchronization dedicated class. It holds two functions, one to send and the other to update the local database.
ServerInfo	Contains all the required information to connect to the right database on the right server.
StartServer	It starts the server (server-side).
StopServer	It stops the server (server-side).
Transactions	It sets and controls the transactions. Class used in a higher level to better encapsulate commands and notifications.
UtilNavigate	Class used as a facilitator; it holds methods; which enable code reuse thus granting greater understandability and maintainability and facilitating the generative approach.
Utils	All-round generic useful methods.
WarningDialogFragment	Class used to show messages in *Android*. It should be used by

	means of the method *showwarning* present in the *UtilNavigate* class, which can be called anywhere, that is any activity or fragment.

6.7 – Synchronization

6.7.1 – Concerns

As aforementioned, every BIS app requires consistent data in both server and client sides. Therefore, we either provide an application that would always require a server connection in order to work which in turn, would greatly affect our final mobility factor or, we provide an application with synchronization capabilities, i.e. with the ability to work offline and still offer the same working capabilities/features. When a connection becomes available the application should be able to automatically synchronize the new data in a consistent way.

6.7.2 – Synchronization class

As shown in the previous sub section the only class fully responsible for the synchronization process is the *ServerAction* class. The latter provides two methods, one to send the new changes, and the other will synchronize the local database with the database present in the server. The code in these two methods is executed in a separate asynchronous thread.

For this specific task, not only the user actions (deletion or creation of an object) are persisted in a command like form, but also every automatic command derived from that action (e.g. the deletion of an aggregated object would result in possible several other deletions). Therefore every action is going to be replicated in the server side.

The process to send the changes to the server requires querying for all the transactions persisted in the database (persisted as *AndroidTransaction* type) using the provided persistence manager component. Every *AndroidTransaction* is composed by one or many *Commands* which in turn hold the attributes shown in Table 11.

Table 11 – Command class attributes

Type	Attribute
CommandType	type
CommandTargetLayer	targetLayer
Integer	oldObjectID
Integer	oldNeighbourID
Class<?>	oldNeighbourType
Object	oldObject
Object	newObject
Object	oldNeighbour
Object	newNeighbour
Object	source

The existence of both ID attributes may be useful to avoid or even accelerate object comparison or querying (these commands are also used locally for notification purposes). To access any needed domain logic and still maintain a low coupling effect, this class uses the provided interface (*ModelMusts*) of the model layer, which as aforementioned provides the four needed, synchronization with server, dedicated methods which are then filled with the values stored in these commands. Getters and setter methods are also available for each attribute, to access these values. A detailed diagram showing the relationship of the *Command* class is shown in the next sub section since it is a fundamental piece of the aforementioned command pattern.

6.8 – Implemented patterns

To guarantee maintainability, the generated apps implementation is built upon reference patterns. Among other used patterns like the singleton pattern, we present in this section three well known patterns, since they are important for our generated architecture. The observer and the command patterns, whom are known to be implemented in several different languages and platforms, and the list view holder pattern, which, as far as we could find, is an *Android* specific pattern. Using modelGoon[20], a plug-in for the Eclipse IDE that enables the creation of UML class, interaction, package and sequence diagrams, from an existing source code, we created, from a generated application, the following presented diagrams illustrating these patterns.

6.8.1 – List view holder

The list view holder pattern is introduced in the generative process of the lists. By using this pattern we guarantee looser coupling, and greater maintainability. As shown in (Guy and Powell 2010), by following this pattern we also increase the performance of the list views. We also created a generic

---

[20] http://www.modelgoon.org/

Luís Silva – July, 2014

ListFragmentController class (adapts to every object type), which in turn removes complexity to both generation and generated implementation. In Figure 30, we can see our implementation of the given pattern. Notice that we only have the *<type>ListViewHolder* class as the concrete object type class. The latter class is responsible for inflating its concrete XML view, and react accordingly to state changes. The, also shown, concrete activity is not part of this pattern, it is present in the diagram for understanding purposes. In conclusion by applying this pattern we provide better performance, and both greater maintainability and easier generative process, since within our implementation only one class must represent its own domain type.
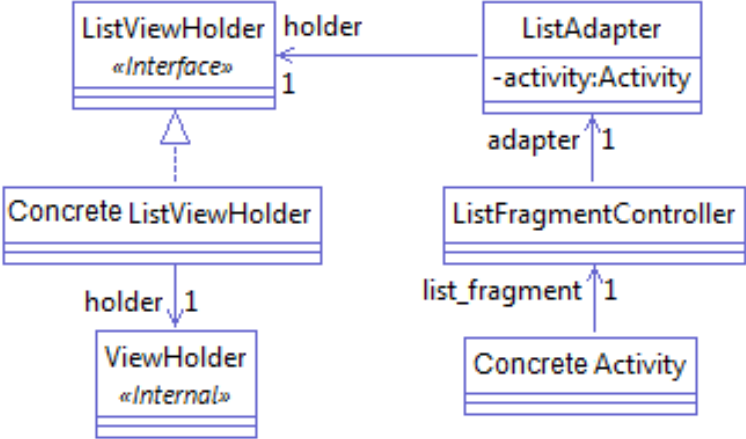
Figure 30 – List view holder pattern

6.8.2 – Observer pattern

Another very important followed pattern is the observer pattern. With this pattern we guarantee that every view its properly updated, but most importantly, as the latter, we also significantly reduce coupling. Since each fragment has its own responsibilities and can also be reused in other ways, that is, by other activities besides its own purpose type activity, it is very important that each fragment has the ability to, by following its own lifecycle, set itself as a listener, remove itself from listening, and finally update itself or its contents in a completely independent way. If any notification received affects other components, it is passed to the holder activity. In Figure 31, we can see the diagram representing our implementation of this pattern. The concrete observer must implement the *propertyChangesListener,* and to start listening it only has to set itself as a listener by means of a static method present in each domain class type, for example, `<type>.getAccess().setChangeListener(this);`.
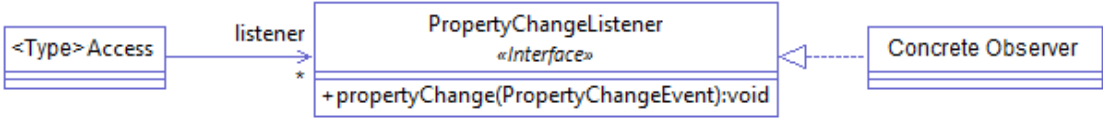
Figure 31 – Observer pattern

6.8.3 – Command pattern

Lastly, the command pattern is also implemented. This pattern plays a heavy role over the implementation. As Figure 32 shows, besides the usage of the *DB4O,* since we allow the data to be synchronized in a later state, we had to create our own transaction system to store action commands. Since we allow to associate objects by means of different screens usage, that is, in order to associate objects a user can navigate to other activities that do not hold the listening fragment. The presented *Transactions* class is also responsible for triggering the state change notifications (implemented by means of the latter observer pattern), which are only triggered after we guarantee database consistency, i.e. even if DB4O or other management systems provide a function to guarantee consistency, namely the *commit* function, our views do not get updated by these management systems, therefore we had to create our own system. Thus the *Transaction* class besides triggering the notifications of the views, it also confirms the end of a transaction to the DB4O by means of the *commit* function. We provide five commands types, which every listener can then use to separate action based events, and we specify which layer is the target of the command, i.e. if it targets the database or the listeners. For instance, let us consider a many to one association and a creation of an association action (action that represents a creation on a link between two objects). Regarding the views, we may have to update every possible view available therefore there might be more than one view command available. However regarding the database or synchronization process there can be only one database command and the latter is created by the holder class (i.e. the class that holds the neighbor). By separating the intent of the commands, this specific model logic is abstracted to the other layers, but it is still available for use in case it is needed, since we know the source of the command. Thus granting greater maintainability, understandability and easier code generation.
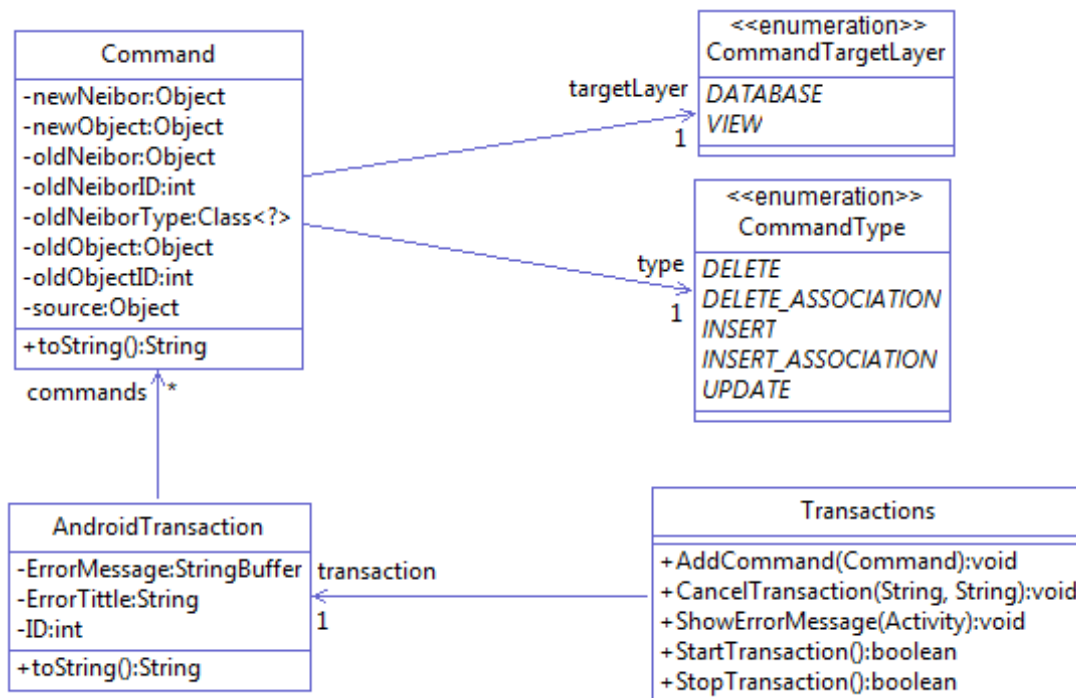
Figure 32 – Command pattern

7 – JUSE4ANDROID

7.1 – *JUSE4Android* – GUI and Requirements

In Figure 33 we can see the *JUSE4Android* GUI screen, created in order to provide a more friendly and easier input environment. Each filling zone is a required setting for the generation process.
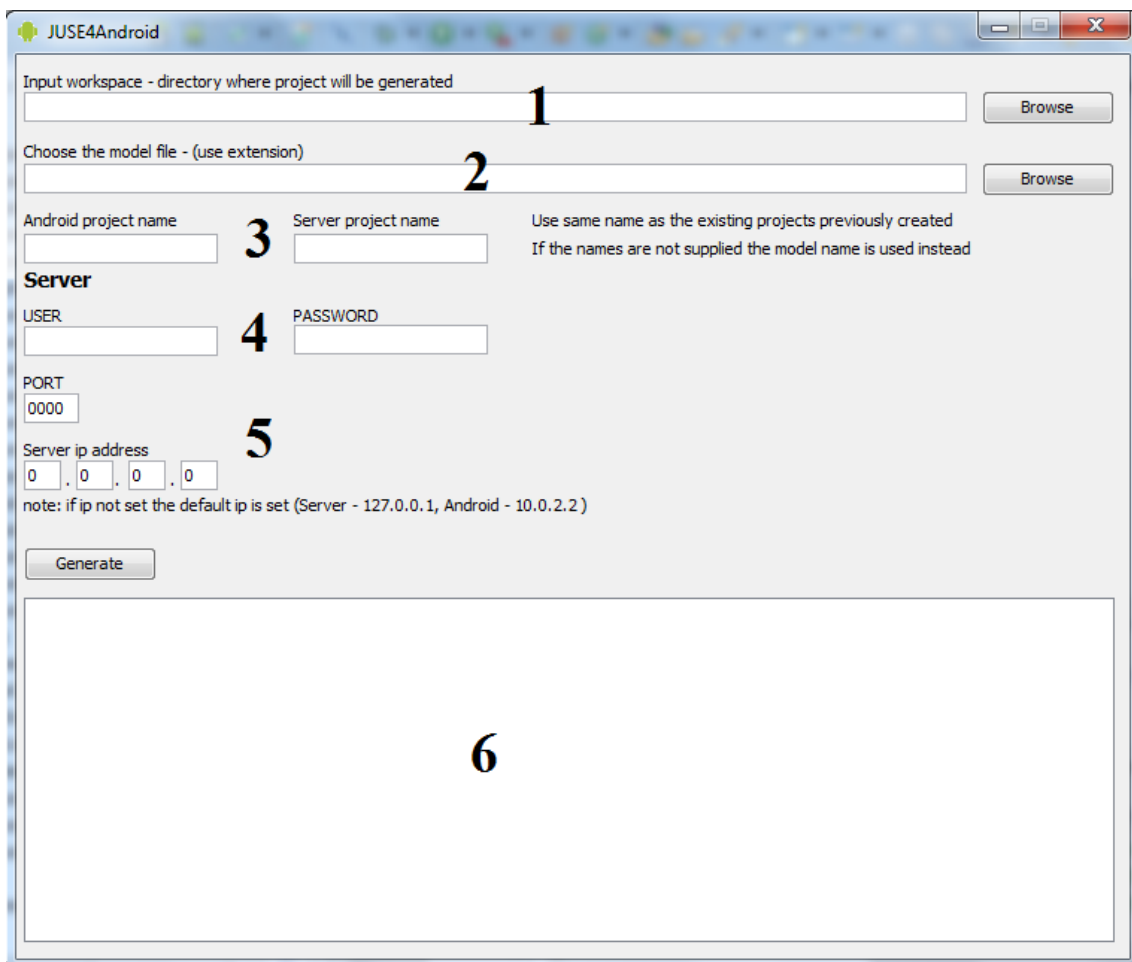
Figure 33 – JUSE4Android GUI screen

1. The directory where the project is going to be generated to.
2. The path of the file containing the model.

Luís Silva – July, 2014

3. Since two different projects will be generated, a name for each one can be set. If these fields are not set the model name is used instead followed by *Android* and Server for, respectively, the client side and the server side.

4. The user name and password used to access the database on the server.

5. A port and ip address used by the server. If not set the local host ip address is used (can be used for local tests).

6. The box or console used to show the outputs of the generator (the outputs regarding errors thrown by USE or the generation done, and not what is generated).

Besides these simple input requirements, *JUSE4Android* also has other *Android* specific requirements in order to successfully generate an error free application project. When creating a new *Android* project in Eclipse many files are created which may overwrite the generated ones, in case the project is created after the generation, an example of such a file is the "*manifest.xml* Since this tool was tested mainly on an Eclipse environment, which in turn uses the ADT for the *Android* development, it is important to notice that a project should be already created before running the generator.*"*. On the other hand, the server side project does not suffer from this type of problems, since it is a *Java* based application.

Furthermore, and considering other alternatives since Eclipse solves this problem seamlessly, the generator does not considers the R.Java file in the generation process or any available *Android* compiler, so in order to build the final .apk's a R.Java must be created/generated. Lastly, to run *JUSE4Android*, *Java* must be installed.

7.2 – JUSE4Android – Structure and generation process

7.2.1 – Open-source tool integration

To reach our proposed goals we researched available open-source projects, upon which we could integrate our project. To make the project feasible, these projects would have to present a minimal set of requirements. They should present the ability to specify models in the UML syntax and also support the OCL. As shown in the 2 – Related work section we found some open-source projects, like the Dresden toolkit, that already provided an advanced generative approach upon we could work on, but unfortunately this project, as aforementioned, does not allow the specification of associative classes. Besides the projects presented in the related work section we also found the UML-based Specification Environment, also known as USE (Gogolla et al. 2007), which is a Java open-source tool developed in the University of Bremen. USE is a system for the specification of information systems. It is based on a subset of the Unified Modeling Language (UML). A USE specification contains a textual description of a model using features found in UML class diagrams, *expressions written in the Object Constraint Language (OCL) may be used to specify additional integrity constraints on the model. The *USE* tool also gives the possibility animate a model *"(…) to validate the specification against non-formal requirements. System states (snapshots of a running system) can be created and manipulated during an animation. For each snapshot the OCL constraints are automatically checked. Information about a*

system state is given by graphical views. OCL expressions can be entered and evaluated to query detailed information about a system state …"(Fabian et al. 2013). While this tool provides a graphical interface that can interpret models, the same ones must be declared textually in a file with a ".use" file extension. Besides the advantage that this fact give us, because the models become this way independent of the tool UI, a possible down factor is the fact that this tool does not offer any XMI to USE model conversion capabilities, while not critical this fact requires the users to learn the needed syntax to describe the UML models.

In conclusion, we decided to use the *USE* tool as base model interpreter and validator, since from the described tools it was the solution that fulfilled all our requirements, as shown in Table 12, *USE* allows the models to have associative classes, and contrarily to *OCLE* it is open-source. In order to better access the *USE* tool we used the *J-USE (Brito e Abreu 2011)*.

Table 12 – OCL tools differences

	Dresden	OCLE	USE
UML class diagram completeness	No	Yes	Yes
Open-Source	Yes	No	Yes
OCL support	Yes	Yes	Yes

By offering a facade to work with the *USE* services, combined with the fact that *USE* uses input models declared outside its UI environment, we were able to create model-driven generation capable projects in a *USE* seamless way.

7.2.2 – GUI and generator – project and standalone

As previously mentioned this tool was created on top of other tools, i.e. makes use of separate open-source projects like *J-USE* and *USE*. In order to take advantage of such open-source projects and at the same time create a stable working environment, without the need for direct adaptation in the other projects, two mains classes or launchers were created: (i) the main tool launcher built over the J-USE; (ii) and a separate main for the GUI. This allows the *JUSE4Android* tool to be launched as a standalone tool without third party (*J-USE* and *USE*) crash consequences, for instance due to thrown exceptions.

In order to achieve such characteristic the GUI main class becomes the standard choice for the standalone tool. This main starts the GUI as in any normal Java application and waits for the start command as shown in Figure 34. When the generated button is pressed this application launches the other *JUSE4Android* main (generator purpose main) on a different process or JVM. This will allow the generator to throw any exceptions or crash, without crashing the GUI application. In order to inform the user of any validation errors, for example, caught by the *USE* toolkit, the output stream channel of the generated is redirected to the GUI process. Therefore, any output or thrown exception is then printed there, as shown Figure 34.
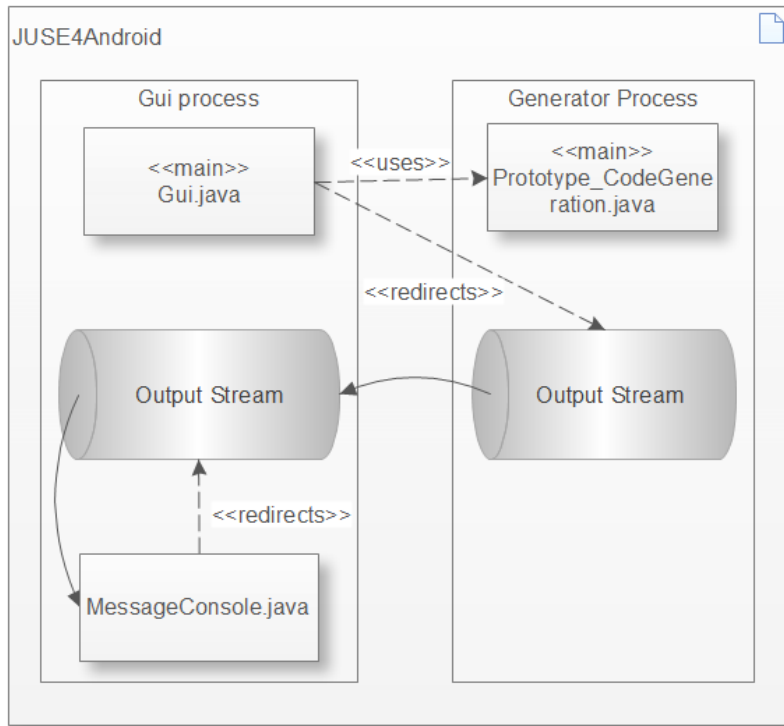
Figure 34 – *JUSE4Android* Standalone mains structure

7.2.3 – Internal generator structure

JUSE4Android, as shown in Figure 35, also uses some APIs of *J-USE*. From the several APIs available we can highlight the usage of the classes: *FileUtilities*; *AssociationInfo*; and *AssociationKind*. The *FileUtilities*, class provided functions that eased our file creation. The *AssociationInfo* class provided a representation for an association and it provided the required getters methods, e.g. it provide a role name which would already follow a common convention, it provides functions that would return the source and target classes of a given association and it also applied a name convention for the classes' names. The *AssociationKind* class is an enumeration class with an already defined set of association types following the UML class diagram standards.

During the development of this tool other *Android* specific interfaces were added, such as *AndroidTypes*, and several methods were also added to some of these interface to fulfill the project needs.
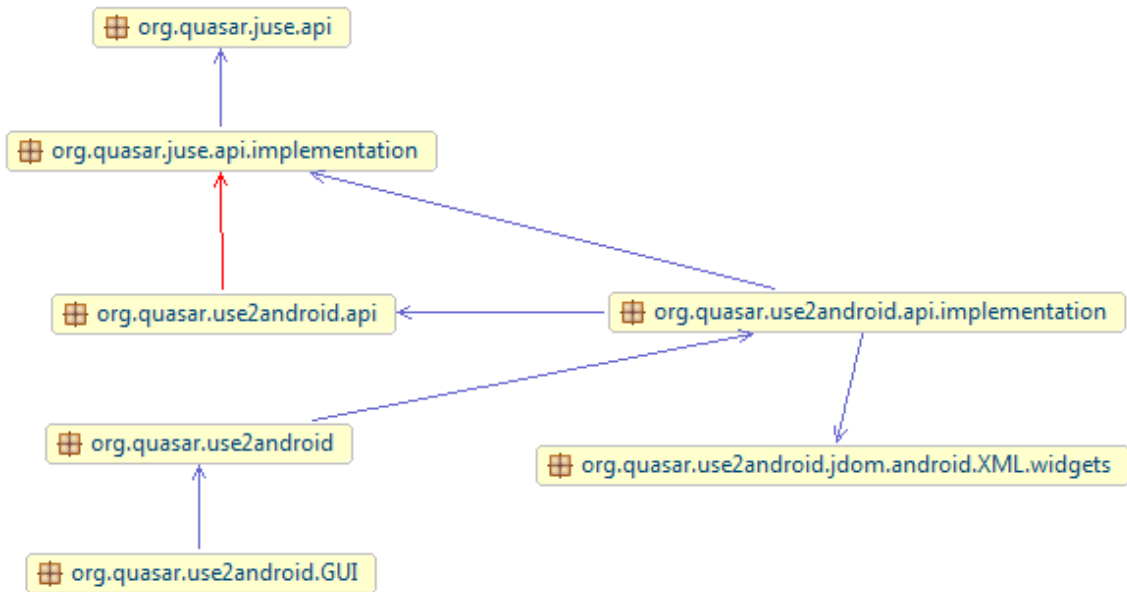
Figure 35 – *JUSE4Android* package diagram

The internal generation process also follows a separate layer structure, i.e. for each layer there will be a class that will handle the generation regardless of any inter layers dependencies. As shown in Figure 36, the main controller (`PrototypeGeneratorFacade.java`) extends the *J-USE* `BasicFacade` class. Also shown in Figure 36, the standard pattern followed is the visitor pattern with the generated methods as being who is visited and the model classes as being the common argument to work with. This pattern was chosen to allow a sustainable growth, i.e. the ability to reuse later these methods. In Figure 37 we can see that only the main controller accesses the *openOutputFile* or *closeOutputFile.* Therefore, the visited classes' only action is to print to an already opened file or stream. Regarding the view layer, since we need to produce XML files, we adopted the *JDOM* toolkit (Hunter et al. 2013) that provide a simple API to create, and manipulate XML files.
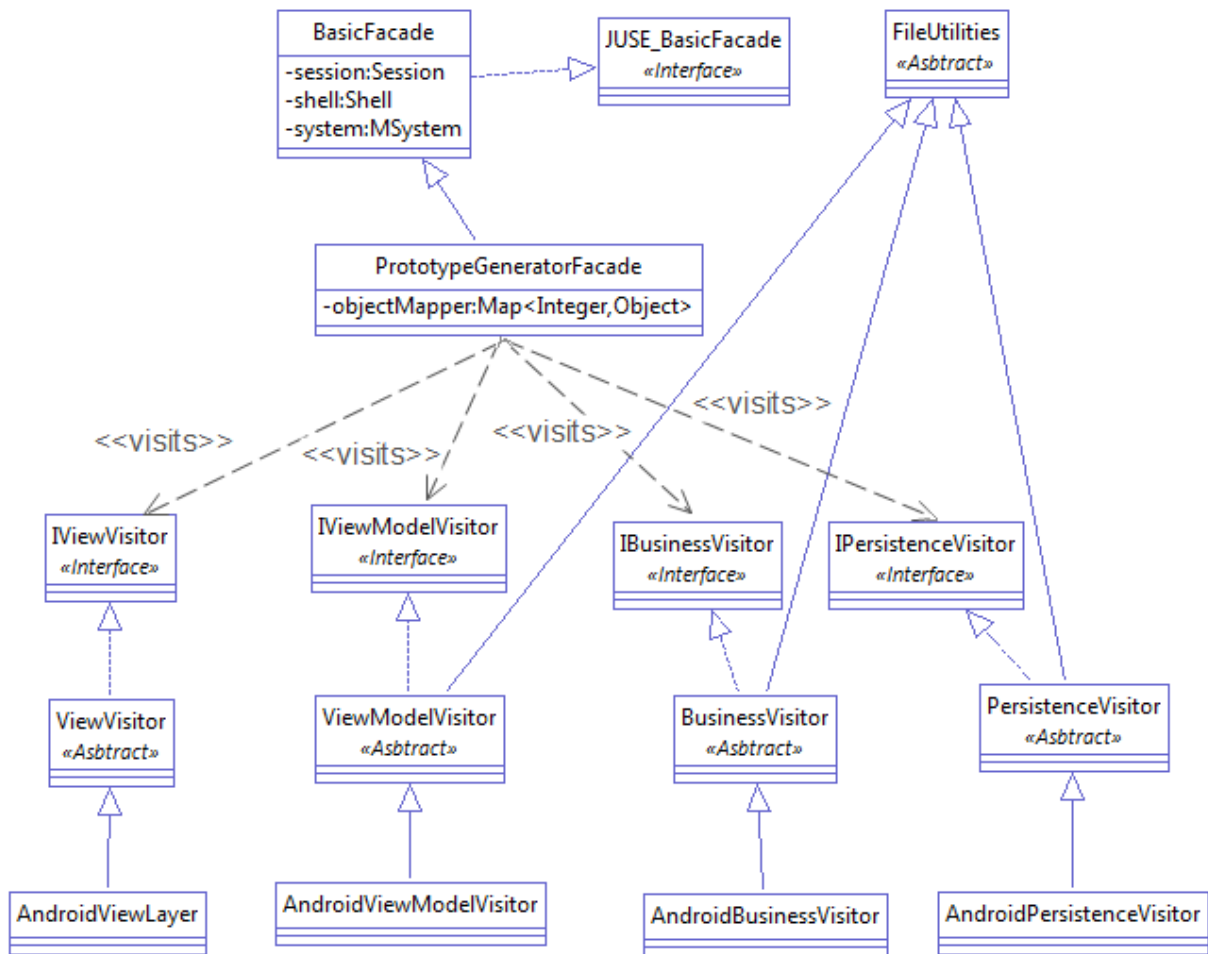
Figure 36 – JUSE4Android relation to J-USE and Visit pattern class diagram
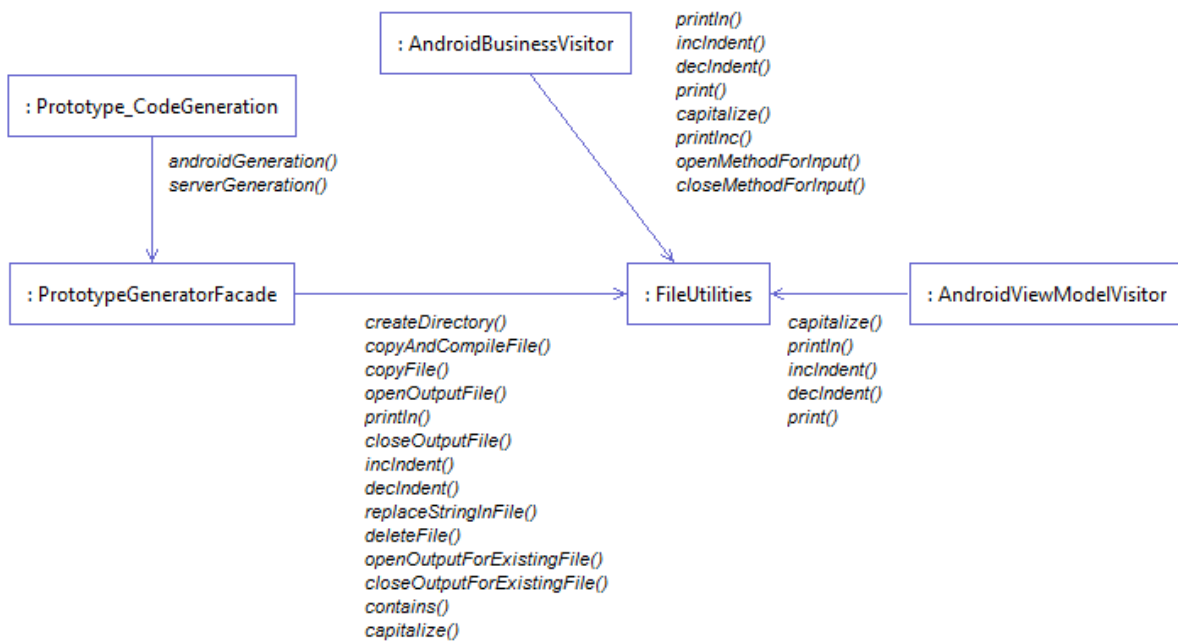
Figure 37 – JUSE4Android interaction diagram

7.2.4 – Static generation process

As shown in Figure 36, the generator structure is divided in layers, corresponding to the target implementation layers, and is controlled by the `PrototypeGeneratorFacade` class (main controller), but only for the above main four layers. Aside this layers there is also the *"utils"* layer, besides being essential to the generation outcome (i.e. generation of a fully working prototype), it was not considered in the previous section due to the fact of the generation process being completely model independent. Therefore its generation did not require "visiting the model".

The generation process applied here was simpler since all the code was independent, except for some classes who need to import other classes, but since these imports are inside this set of classes (i.e. target other independent/static classes) the process could be done differently. For instance, these classes are all defined in text files (.txt) and to solve import problems simple identifiers were used in the right place to be replaced later, namely for the package and in some cases also the *Application* class name, since its name derives from the main model name or final project name. In the sub-section Utils layer of the section Static Generation Process – identifiers in the Appendix we can see the classes that are in this set, notice that most of these are classes that define the implemented patterns. Besides these classes there are also other files that also follow a simple copy approach in the view layer, namely all the files placed in the folders: `drawable`; `menu`; some files in the `layout` folder; and `values`; as can be seen in the sub-section View layer of the section Static Generation Process – identifiers in the Appendix. These files are even simpler and do not even have any identifiers since there is not a need, so they simply are copied to the right folder. Except for four files which will contain the project or model name in the beginning of their file name, but only for better understanding purposes, namely the files present in the latter sub-section whose name file starts with *<model_name>*.

7.3 – Model transformation

In this sub-section we will present the most important part of our generative process, for each of the presented layers, i.e. the applied rules for each given model parameter.

7.3.1 – View Layer

7.3.1.1 – Generation Approach – Static

Since, in our generative approach, we follow a template model there is always a static data side that does not need any information from the model to be generated, so as said before we enforce some decision making, namely by generating:

- The folders, with the shown qualifiers, as shown in Figure 27.
- The sizes applied for each qualifier (`default_layout_styles.xml`).

- The layout applied to each qualifier (`pane_decider.xml`) – off course this is done for all domain classes, and therefore is model dependent, and we already have pre-defined which layout is set for each qualifier therefore is considered static.

But since there is the possibility that the user may want different settings for each type of data, we replicate some of the files like the two layout files.

7.3.1.2 – Generation Approach – Dynamic

For all files shown in Figure 27 are considered dynamic, since they are completely model dependent, in both existence and content generation. Next is shown the goal and the approach applied to each of these files as well as the adopted naming convention.

Naming convention

As shown in Figure 27, every XML file name will start with the name of the corresponding domain class (represented surrogate "type"), followed by other qualifiers separated with an underscore, whose semantics is described in Table 13.

<p align="center">Table 13 – Naming convention qualifiers</p>

Qualifier	XML file content
view	Type specific view
form	Type specific mergeable representation
layout	General layout
strings	Type specific raw string data
detail	A none editable representation of information data
insertupdate	A editable representation of information data
list	Information data used in a list
navigationbar	Information data used in a navigation bar
onepane	One pane layout screen view
twopane	Two pane layout screen view
component_styles	Type specific view style settings

For example, considering again the *"Worker"* class, we would have at least these two files: the *"*`worker_view_detail.xml`*"*; and the *"*`worker_form_insertupdate.xml`*"*; the first would hold the view (i.e. holds all the viewgroups and views) that will be shown, in a static way (not editable), in the detail part of the worker assigned screen (see Figure 6). The second example file is a mergeable XML and holds all the components that allow a creation or modification of a worker object type (i.e. depending on the chosen attributes in the annotation creation).

Forms

Both `<type>_form_detail.xml` and `<type>_form_insertupdate.xml` follow the same generation approach with only three differences: (i) the available attributes are based on their annotations respectively *display* and *creation*; (ii) regarding the ids, the detail has the *detail* qualifier and the *insertupdate* has the *insertupdate* qualifier; (iii) and finally the *Android* UI components applied for each attribute type. Both files are mergeable, so both start with the merge tag. Both will have a `RelativeLayout` view group to define the layout to apply and order all the views inside it. For both the latter id is composed by the class name, followed by the qualifier detail or insertupdate depending on the file, and finally by the qualifier layout. The width is set to `match_parent` (this means that will be as big as its parents width, by default the view groups width will always be `match_parent` which means the available screen width) and the height to `wrap_content` (this means that it will have the height equal to the sum of the of the height of its children views). Immediately inside the latter there are many `LinearLayout` view groups. As the given attributes their width and height are `wrap_content`, their id follow the same principle as their parent, but instead of the qualifier layout it will end with the name of the attribute. Regarding the orientation it will always be horizontal (this setting means that the children of this view group will be disposed horizontally). Lastly, the latter will have another setting namely, and since it is inside a `RelativeLayout`, all attributes will be set below the previous set attribute, unless they are the first to be set. Inside each `LinearLayout` there will be two components: (i) a static `TextView` that acts as a data descriptor; (ii) and a dynamic UI component (for generation purposes this UI component varies depending on the data type).

In Figure 38 we can see the given effect. On the left side is an abstract representation of the template used to both form XML files. On the right side is the concrete representation of the same template, for the *Worker* class example shown previously (*"worker_form_detail"*). The square in the left side with an asterisk (`LinearLayout`) represents a repetition/loop, in the right side we can see the resulting effect.
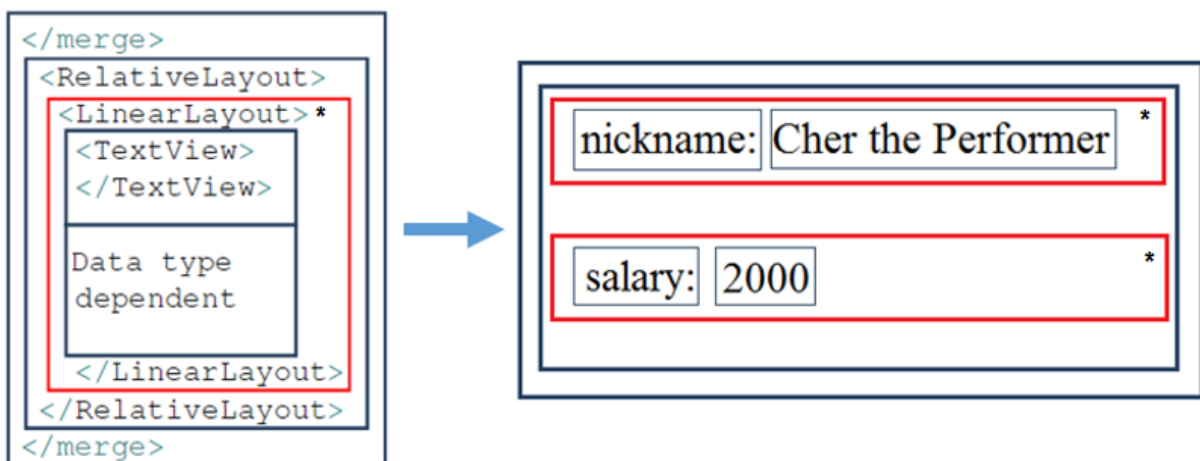
Figure 38– Form XML template and worker detail form example

These last UI components will have the same ID as its parent, with one more qualifier in the end, (i) will end with *text* and (ii) will end with the attribute name.

Regarding (i): it will have `wrap_content` as width and height and since its text is static, its text setting will point to a string which as the same id. This string value can be found in the values folder, in the XML file named `<type>_strings.xml`.

Regarding (ii): the next two tables show for each attribute type the UI component chosen and its settings. In Table 14 we can see the detail, and in Table 15 the *InsertUpdate* case.

Table 14 – OCL type to UI components (widget) transformation – detail XML

Detail				
OCL Type	UI component	width	height	special
Integer	TextView	*wrap_content*	*wrap_content*	None
Number	TextView	*wrap_content*	*wrap_content*	None
String	TextView	*wrap_content*	*wrap_content*	None
Real	TextView	*wrap_content*	*wrap_content*	None
Date	DatePicker	*wrap_content*	*wrap_content*	Clickable = false
Boolean	CheckBox	*wrap_content*	*wrap_content*	Clickable = false
Enum	TextView	*wrap_content*	*wrap_content*	None
Object Type	Include (uses the dedicated type form)	-	-	None

OCL Type	UI component	width	height	special
			InsertUpdate	
Integer	`EditText`	`match_parent`	`wrap_content`	inputType=number
Number	`EditText`	`match_parent`	`wrap_content`	inputType=numberDecimal
String	`EditText`	`match_parent`	`wrap_content`	None
Real	`EditText`	`match_parent`	`wrap_content`	None
Date	`DatePicker`	`wrap_content`	`wrap_content`	Clickable = true
Boolean	`CheckBox`	`wrap_content`	`wrap_content`	Clickable = true
Enum	`Spinner`	`match_parent`	`wrap_content`	prompt=<type>descriptor , entries=<type_enum>
Object Type	Include (uses the dedicated type form)	-	-	None

The special settings shown are used to constrain the input to special settings (e.g. integers will only show a numerical keyboard and that keyboard only allows simple numerical input), or completely like for the `DatePicker` (it only allows the user to change the date in case is clickable).

Finally, we generate the form for the navigation bar. This last form follows a more static generation approach since the only change is in the settings of the UI components, which are always the same. It also starts with the merge tag, and for each class association found, it will generate a `LinearLayout` with the settings presented in Table 16.

Table 16 – Navigation Bar association view group settings

ID	<SourceRole>_navigationbar_association_<TargetRole> or if no role is found/used <ClassName>_navigationbar_association_<TargetClass>
Width	`match_parent`
Height	`wrap_content`
Orientation	`horizontal`
clickable	`false`
Long_clickable	`True`

Inside this view group a `TextView` will be always generated, followed by an `ImageView` and lastly another `TextView`. All of these will have the `wrap_content` setting for both width and height and as

ID, will have the same id as its parent with one more qualifier in the end namely and respectively: *name, image* and *numberobjects_text*. The only association dependent setting in the above is the image source name (in the `ImageView` UI component). As shown in Table 6, for each association, we select a pre-determined image source, depending on the association target multiplicity value. If is one is a ToONE otherwise it is considered as a toMANY. Besides these, we must also check if the class in question is a sub class, to allow navigation to its super. Finally, if the class is a super class, we must check its direct children and properly set the source image name.

Views

The View type XMLs are responsible for choosing the right forms and putting everything properly together. If we need to add any other component to a specific view, this is where it should be done. This avoid polluting the form type XMLs with any content outside the model specification, making the code much more easy to maintain and construct. As Table 17 shows, both views are very similar. The only differences are: (i) the form that is included – the *"*`include`*"* tag in *Android* is used when we want to use, the already defined, mergeable XML files – in each view; (ii) and for the *insertupdate* view another inclusion at the bottom of the view, namely two buttons (ok and cancel buttons) present in the mergeable XML file *"*`default_okcancel_buttons`*"*.

Table 17 – Detail and InsertUpdate Views XML templates

Detail View	InsertUpdate View
``<ScrollView>
  <RelativeLayout>
    <include layout="
    @layout/
    TYPE_form_detail">
    </include>

  </RelativeLayout>
</ScrollView>``` | ```<ScrollView>
  <RelativeLayout>
    <RelativeLayout>
      <include layout="@layout/
        TYPE_form_insertupdate">
      </include>
    </RelativeLayout>

    <RelativeLayout>
      <include layout="@layout/
        default_okcancel_buttons">
      </include>
    </RelativeLayout>
  </RelativeLayout>
</ScrollView>``` |

Besides the templates not showing, both views may also have a repetition/loop, namely the sub-classes present in the model (inheritance scenario). In case we are dealing with sub-classes, where by their own nature they inherit their parents characteristics, the views are responsible for merging the different forms by the order of the inheritance tree, that is, from the most super class until reaching the class which the view is responsible for. For instance, the *"*`training_view_insertupdate.xml`*"* will firstly include the *"*`project_form_insertupdate.xml`*"* and below it will include its own form

(besides the fact that for this case its own form does not present any new information). The latter is one more example of the importance of separation of concerns. Notice that by separating the data into form type xml we can represent each class in its own XML without any modeling effects like the inheritance case, and thereafter in the view type XML we merge these contents as needed and add any other needed contents and functions like the buttons and scrollable capabilities. The other, not shown, views follow the exact same approach with the difference that the content on the list depends on the annotation "`list`" and the navigation bar only depends on the associations of the class.

Special case scenarios

Some special cases will now be described. One is the conjunction of inheritance with the associations when creation is needed. When the user intends to associate one instance to an instance of a super class, the user is presented with a list of the possible instances types (a super instance if not abstract and all its children). To provide these lists, special XML files are generated for instance types that are not expected. Let us consider the *Worker*, *Qualification* and *Company* types from our example. While they seem normal classes with normal binary associations to other classes, they all fall in this category simply because they are associated with an inheritance three (Project class) and therefore must have these XML view files. On the other hand, every class in an inheritance tree must provide knowledge of its children. So in order to solve this problem two new XML files are introduced in the generation process: (i) "`<type>_generalizationoptions_offsprings.xml`" generated for every super class present in the model. These are mergeable XMLs files (like the form type files but instead of serving their own class type views they will server another class type), which have their direct children representation followed by a simple divider bar, and which also include their direct children off springs XML file. The children will do the same. So a top-down approach, as shown in Figure 39, will be generated guaranteeing a full inheritance tree coverage; (ii) "`<sourceType>_generalizationoptions_<targetType>_view.xml`" are the XMLs views files used by the classes that are associated to an super class, i.e. that are going to show the, explained previously, dialog. So, as shown in Figure 39, in the latter file we represented the association to the super class, therefore the super class is represented, and finally it is included after a divider the super class off springs XML file. In Figure 39 we can also see the dialog box that appears in the *Worker* class dedicated screen when the user does a long click over the projects (*Project* class) association. The reason for the existence of the `offsprings` files in the last child (i.e. it should not be necessary since it is the last child), is to maintain the coherency, that is, we threat the inheritance relationships like associations. A super class should only be aware of its direct children, therefore it should not know if its direct children also have children. Consequently, the children must supply their parents with an `offspring` file even if it is empty.
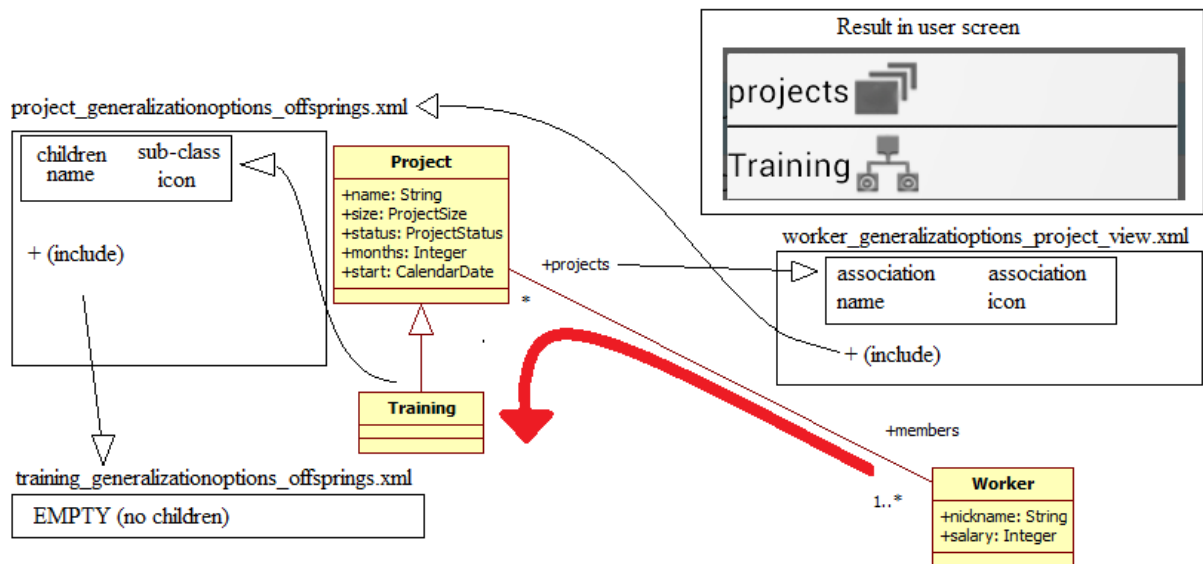
Figure 39 – Association Creation to Inheritance Tree example (worker class screen)

7.3.2 – Type mapping

7.3.2.1 – To Java

So far, we presented the transformation mapping for the view layer that uses XML. From this point forward, that is, the other layers, the target language is Java and also we target *Android* specific types. Therefore, in this sub-section, we present the transformation of the OCL types to the specific target language types.

Primitive types

Table 18 – OCL to Java types mapping

OCL types	Java types
Integer	int
Real	double
Boolean	boolean
String	String

Collections

For these special cases, there is a need for both super *Java* type and implementation type to be able to initialize the collection.

Table 19 – Collection OCL to Java type mapping

OCL type	Java type	Java implementation type
`Bag<T>`	`List<T>`	`ArrayList<T>`
`OrderedSet<T>`	`SortedSet<T>`	`TreeSet<T>`
`Sequence<T>`	`Queue<T>`	`ArrayDeque<T>`
`Set<T>`	`Set<T>`	`HashSet<T>`

7.3.2.2 – To Android

We can also create views dynamically using the Java syntax, for instance to set click listeners or change content in runtime, we reused the same properties. For instance, an OCL Integer is also transformed to a `TextView` or `EditText`, and this is all we need to know, since for instantiation purposes we only need to either set as a *View* type (super type of any widget), or the specific type which is the same. For implementation purposes there is no need for a transformation process, since we retrieve the already initialized view from the system. Again, we just need to do the casting in case the *View* super type is used. In the next piece of code we can see an example of the instantiation code for an `EditText` component.

```
(EditText)rootView.findViewById(R.id.<type>_insertupdate_<atribute>_value);
```

7.3.3 – View-Model layer

There are mainly two file types in this layer that are model dependent, the activities and the fragments. For each domain class one activity, and three fragments (navigation bar, list view and detail) are created. The model independent files, also generated for this layer, are the `Launcher`, `MasterActivity` (super activity that holds generic actions to all activities) and the Application class. The latter three do not have any model specific parsing requirements. Therefore are generated as we specified them, with the exception of the *Launcher* class which requires to know whose classes have the `Startingpoint` annotation so it can set the navigational action upon the buttons shown in the initial screen. Since the initial navigation does not require to send any arguments, it only needs to add "*Activity*" after the class name to indicate where it is navigating to.

In our generative approach the classes are our main argument, i.e. we cycle every class and we generate an entire class content upon its characteristics which include: own features (attributes, methods); and indirect features which include the neighbors class (relate to other class through an association) relation features however, there some specific that deserve special attention: (i) having associative classes as a member of another associative class; (ii) and having a class being associated to a super class. In the (i) case scenario, represented in Figure 40 where the class that we are

processing is the *Associative1*, we are obliged to generate all the path from the first associative class to the farthest members in order to show the intended screen, for instance, in the list view.
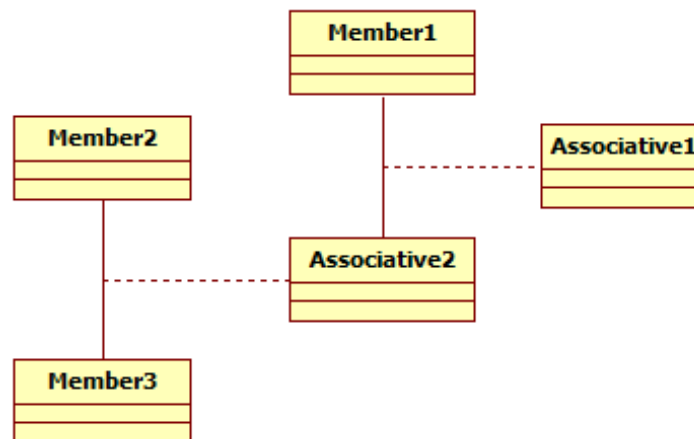
Figure 40 – Aggregated associative classes' example

In the (ii) case scenario, which it was already aforementioned in the previous sub section, we must have knowledge of all the inheritance three namely, when generating the `NavigationBarFragment,` since the navigational action is defined and reified by this fragment and it must know every class it must navigate to and the arguments it has to send. The activities also experience from the same characteristic, i.e. upon a navigation the activities are the ones who received the arguments and therefore must analyse and react upon them. Therefore, an activity representing a subclass must react accordingly to arguments from a neighbour of its super class (not direct neighbour) and, most importantly, an activity must react to arguments from a subclass of its neighbour (not direct neighbour).

7.3.3.1 – Model parsing/analysis

To analyse and decide upon the characteristics of each class, some decisions had to be made, namely which characteristics to use to serve as arguments in the generated implementation (these arguments have to be unique) and also how to perceive those characteristics when analysing the model. In Figure 41 we can see a portion of the model used as example in this dissertation which represents a good example of how we must be careful when making such a decision. Let us consider the Training activity which receives arguments in order to prepare its initial state. When navigating to the Training activity we must send an argument that would identify the source/caller. If we use any of the roles name, the *Training* activity would not know the source, since the roles names are not unique in the entire model. For instance, as it can be seen in Figure 4, there are many "*projects*" role names for the Project class. If we use the class name to distinguish the source/caller, we will be using a unique qualifier, since there cannot be two classes with the same name in the same UML class diagram. However, there is a problem, if we navigate from the *Qualification* activity to the *Training* activity while there is a way to identify the source/caller, there is not a way to identify its origin path, i.e. its intent. Consider that we navigated from the Qualification to the *Training* activity in creation mode. If

this was the scenario, the *Training* activity had to know if the user had navigated through the *Project* or directly from the *Qualification*, in order to decide which association was meant to be created.
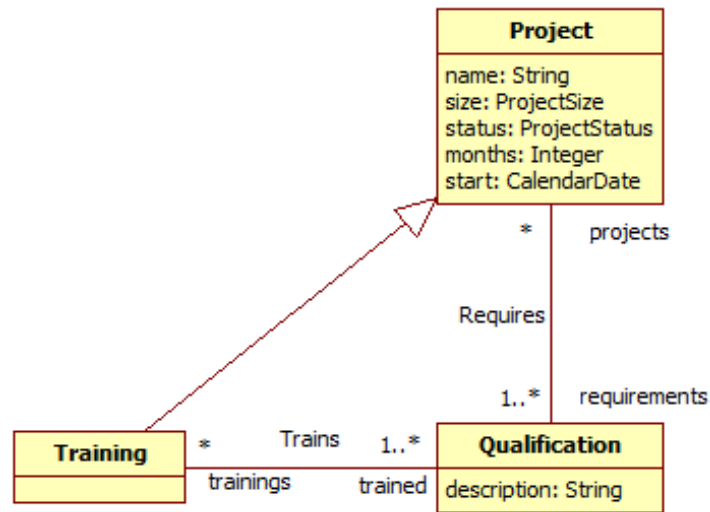
Figure 41 – ProjectWorlds analysis decisions – example.

In conclusion, we used the names of the associations to distinguish them, since they also are unique. For the inheritance scenario we created a naming convention namely the name of the class followed by the word "*Association*". For instance, if navigating from the *Project,* a "*ProjectAssociation*" would be received by the *Training* activity. Lastly, to distinguish the relationship between the associative class and its members, we used the names of both classes like in: "`<name of member class>_<name of associative class>Association`". The relationship between both members has its own name specified in the model.

7.3.4 – Model layer

For each domain class in the PIM (identified with that domain annotation) two classes are generated: a POJO and an Access classes. The @holder annotation, used over the associations' specifications, also directly affects the outcome of the code in this layer.

7.3.4.1 – POJO

The POJO class will hold all the attributes as defined in the model, plus the getters and setters methods, following the common naming convention. The getters have the same name in lowercase, the setters naming will start by "set" followed by the capitalized name. The associations follow the exact same principle but with the target class name, with the exception that, if there is the identifier *role* present in the model its value will be used instead. Besides the defined attributes it is also going to be generated an *ID* attribute and respective getter and setter methods. Regarding the defined

operations a *//TO DO* comment is added in the generated code as common in IDEs such as Eclipse[21] to identify pending issues. The relationships are defined as a normal relationship would, that is, as a reference or a collection of references and also have the respective getter and setter methods.

Regarding our generative approach for the latter instances and methods the type of association and the *@holder* annotation will have impact. For instance, in case we are facing a many to many or a one to one association, we automatically decide, based on the complexity of the classes, which class will hold the other (i.e. which class will have defined the instance of the other class), or if the *@holder* annotation is present in the model, the class with the given annotation will be the holder, regardless of its complexity. In many-to-one scenarios the holder will always be the class that hold only one instance of its neighbour. For each association found in a class, the generation approach will depend on the latter being a holder class or not. If it is, the reference or collection of references, will be created, and the getters and setters methods body will reflect that knowledge by working directly with the reference or collection of references. If it is not the holder class, there is not going to be any reference to the neighbour and the latter methods bodies will make use of the neighbour methods in order to retrieve the data. For collections we also generate an add method and a remove method in both sides to allow adding and removing single instances, without the need of preparing an entire collection for an update action. The latter methods also use the same naming convention as setters, but instead of "set" it is used "add" and "remove", and, regarding the method body content, also follow the same holder logic. To better understand the generative approach an example between the *Project* and the *Worker* classes is shown in the POJO – Relational getters and setters example section of the Appendi.

Regarding the Aggregation and Composition, the POJO class would look the same as the previous ones depending of course on the cardinality between this two classes. For Inheritance, the same principle is applied, since each class is responsible for its own relationships.

Besides the latter an `allInstances` method is also generated, which returns all the instances of the given type and is the same in every scenario, even for the inheritance case, since the responsible method in the persistency layer is prepared for this case scenario. It is also generated a Boolean for each association to control validation state, a Boolean to store the general validation state (e.g. used for deciding if the alert icon should be displayed as shown in section 4.3.2 – GUI views and widgets), and methods that control and verify such states. Five CRUD methods that access the respective methods in the own `<type>Access` class with the appropriate parameters, and lastly other methods which purpose is to facilitate both generation and understandability. An example is the `getType` reflexive method which returns the type of the class and can be useful when using interfaces, like in our synchronization manager component, since there might be a need for more specific verification criteria.

---

[21] http://www.eclipse.org/

7.3.4.2 – Access class

We have already described in section 6.4.3 – Access class which methods these classes hold. They are called `<type>Access` and are generated along with the POJO classes. A sequence diagram of the insert method for the *Worker* class, shown in Figure 42, is used to illustrate the required steps and process to fulfill this action.
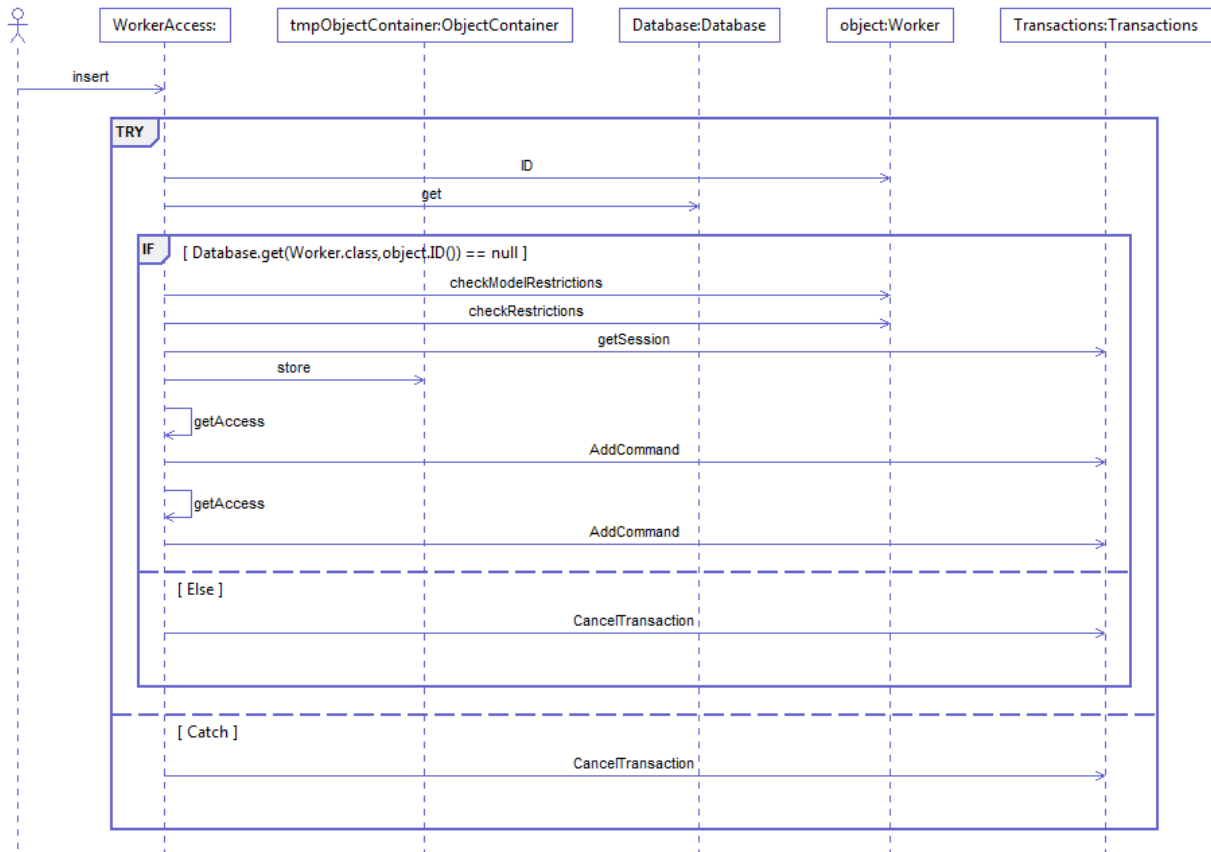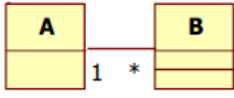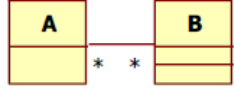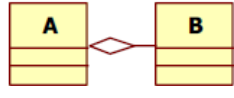
Figure 42 – Insert method – *Worker* class example

Through a try/catch block we ensure that if any error occurs during the insert process the transaction is cancelled. Next, in the `if` condition, we verify if the given object does not already exists, if it does we cancel the transaction. Finally we call the methods that will verify the associations' constraints and in the process update them, then we access the current session and store the object. In the end we had two commands, one to target the database (for server synchronization) and another that targets the views for notification purposes.

The other methods follow the same process with minor changes. The update method verifies that the received object still exists and the new one does not. Both insert and delete associations also receive as argument the neighbour, and also verify if the received object exists, and then if the class is the holder, they do the respective process and create a database command. If not, they call the respective neighbours method. In the end the associations both verify the associations' constraints and store the object.

Lastly, the delete method also follows the same process as the previous methods, but before the deletion it calls the `notifyDeletion` to guarantee consistency. In Table 20 we can see the different scenarios, depending on the existing associations and if it is a holder class or not.

Table 20 – Deletion notification – Mapping solutions

Association Type	notifyDeletion Code	
	Class A	Class B

A ─ B 1 *	`for`(B x : a.Bs())
	        x.deleteAssociation(a
	);
		`if`(b.A() != **null**)
		        b.A().deleteAssociation(
		b);

A ─ B * *
A is the holder
	`for`(B x : a.Bs())
	        x.deleteAssociation(a
	);
		`for`(A x : b.As())
		        x.deleteAssociation(b);

A ◇ B	`for`(B x : a.Bs())
	        x.delete();
		`if`(b.A() != **null**)
		        b.A().deleteAssociation(
		b);

7.3.5 – Persistency layer

As aforementioned, this layer is composed only by the `Database` class and, except for `dbServerConfig` and `dbConfig` methods, every content is pre-defined with the exception of the required imports needed depending on the generated contents in the latter methods. As already explained, we need to set the `updatedepth` for each domain class. Therefore, for each domain class we must analyze its needed update depth. Following the same logic as in the previous sub-section we set the rule for this calculation. If the class is the holder of all its associations, then its update depth is one, since it holds every reference. If the class contains any association which it does not hold, then its neighbor is the holder class. Therefore, the update depth must be two. In addition to the update depth we also configure the database, for each domain class, to be indexed by the attribute *ID*.

As it can be seen in the following code, only two lines of code are necessary for each domain, model class to allow us to set these rules, namely:

```
configuration.common().objectClass(<type>.class).objectField("ID").indexed(
true);
configuration.common().objectClass(<type>.class).updateDepth(N);
```

Where *Type* represents the class name, and *N* is always one, except in the two previously mentioned cases. The rest of this class generative process follows a static approach, since all the querying methods provide simple filtering arguments. For instance, to retrieve an object as previously shown, we only require the type of the object (class type) and an Integer, since every object has its own unique ID.

```
get(Class<T> c, int constraint)
```

The same is also applied to retrieve sets of objects, for both simple and inheritance participants' classes as shown in the next code.

```
public synchronized static <T, Y> Set<T> allInstances(Class<Y> prototype) {
return new HashSet<T>((Collection<? extends T>)OpenDB().query(prototype));
}

public synchronized static <T, Y> ObjectSet<T> allInstancesOrdered(Class<Y>
prototype) {
return (ObjectSet<T>) OpenDB().query(prototype);
}
```

7.3.6 – Static implementations

As already aforementioned, there are classes that will be independent of the model, namely the classes present in the *Utils* layer. Although some of these classes require domain logic algorithmic usage and therefore domain specific type knowledge, as in for our synchronization manager component. By providing various specific methods, which in turn fulfill all the requirements of this layer, through an interface (in our case the *ModelMust* interface as shown in Figure 26) we avoid passing any domain logic to this layer. Thus avoiding the need for any model-driven generative approaches besides the obviously need to know the domain type abstraction provided interface and its methods. Since there is not a need for any domain specific type knowledge, for the generation of this layer we followed another approach. We specified the code in normal text files, since it is sufficient, and for any needed import or cast we specified a specific qualifier, as shown in the sub section Utils layer of the section Static Generation Process – identifiers in the Appendix.

[This page was intentionally left blank]

8 – Validation

8.1 – Generated implementation

In order to test the generated implementation, we followed a bottom-up validation approach. Starting from the persistency layer and going upwards until the view layer, each layer and its purposes were tested. The latter approach was chosen in order to guarantee safer results, since sound third parties technologies were used in order to test the persistency capabilities.

8.1.1 – Seamlessness validation – Persistency and Model layers

To validate the persistency layer, besides following the *DB4O* guides (Versant, 2013a), we used the *Object Manager Enterprise* (*OME*) (Versant 2013a) to query and maintain data in a *DB4O* database. The validation technique was based on matching the black-box perspective granted by the generated app UI, against the white-box perspective granted by the *OME*. This way, by means of a proven third party tool, we could assure a functional persistency and model layers implementations. In the example on Figure 43 we can see a worker object with data content and already associated to other objects. In Figure 44 we can see the corresponding *OME* view in Eclipse, showing all the instances of the *Worker* class and all its attributes and associations. In its bottom we can see the values of the selected object, which is the same as the one selected in Figure 43. After each test, involving the execution of the CRUD operations, we tried to make sure that the expected result was in an expected state in the *OME*, to guarantee that we did not get wrong results due to improper querying in the generated code, or improper code in the presentation layer. After providing a stable data layer we were able to test the upper layers. For example, let us consider an update action, which means that we are going to change an old object state to a new state. With DB4O, since we always use the store function to either update or create objects, we may get false results if we test such an action in our own UI views, If the DB4O engine loses the reference to the object under consideration the same store function will create a new object instance instead of updating the old one, and both objects would still show the same information in the UI views.
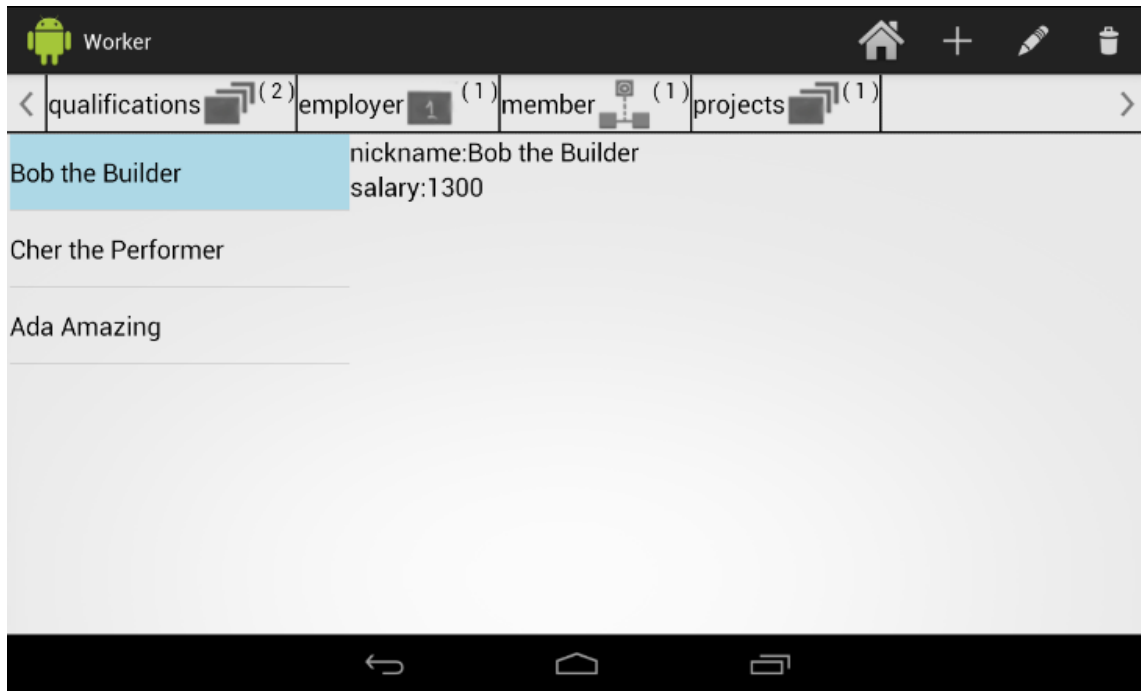
Figure 43 – ProjectWorld Worker class – OME and generated Android Application validation example – Application view

Build Query | Query Results

Row Id	projects	nickname	employer	hasEmployer	hasProjects	hasQualificatio....	ID	AssociationRes....	salary
1	1 items	Bob the Builder	(G) org.quasar.Projects...	true	true	true	-7786321	true	1300
2	1 items	Cher the Perfo...	(G) org.quasar.Projects...	true	true	true	-608969008	true	2000
3	1 items	Ada Amazing	(G) org.quasar.Projects...	true	true	true	-23323873	true	2500

Object 1

Field	Value	Type
org.quasar.ProjectsWorld.bus	(G) org.quasar.ProjectsWorld.busi...	org.quasar.ProjectsWorld.business...
projects	1 items	java.lang.Object
nickname	Bob the Builder	java.lang.String
employer	(G) org.quasar.ProjectsWorld.busi...	org.quasar.ProjectsWorld.business...
name	Incredibly Blue Marrow	java.lang.String
ID	-44106230	int
hasEmployees	true	boolean
hasProjects	true	boolean
AssociationRestriction	true	boolean
hasEmployer	true	boolean
hasProjects	true	boolean

Figure 44 – *ProjectsWorld Worker* class – OME and generated Android Application validation example

8.1.2 – Validating – View-Model and View layers

The implementation for these two layers, besides already following the proposed proven patterns, was tested using a simple observation methodology over the logs present in the *LogCat* (*Android* logging system), a mechanism for collecting and viewing system debug output. We observed that output in both available real devices and emulated ones by means of the *Android* mobile device emulator, and lastly using the Eclipse IDE environment with the *ADT* installed. By generating to an already existing dummy project in the Eclipse IDE, we were able to test, more easily, our generated code against compilation errors. By means of real and emulated devices, we carried out black box tests and, by means of the *LogCat,* we were able to run simple white box tests. Within the first scenario, we were able to check the screen size and resolution adaptions for different devices. At the same time, by means of the *LogCat,* we were able to search for run-time errors when testing basic mobile functionalities, like rotating the screen and locking and unlocking the phone, test the views state consistency, the flow control and the views notifications process.

8.1.3 – Validating – Understandability – UI layer and flow control

Two tests were taken in order to study the model-based UI generation perception and also to study if the control flow was perceived as it was intended. In this case we were interested in perceiving if the – navigate in order to associate – actions were indeed understood.

The subjects of the two experiments were not randomly selected. However, the available convenience sample covers the set of desired characteristics for the expect subject profiles, as described in Table 21.

The *Mobile tech knowledge*, *BIS app knowledge* and *Modelling knowledge* categories were identified to characterize subjects' knowledge. The first relates to an all-round knowledge in technology, more specifically of how to use mobile technologies. The second relates to usage of BIS apps and finally the third to knowledge about generic modelling. These categories are graded with an ordinal scale (*Basic*, *Advanced* and *Expert*) as described in Table 21.

Table 21 – Subject knowledge categories

Characteristics	Values	Description
Mobile tech knowledge	Basic	The subject owns a mobile device, uses it to basic functions like making calls and others, knows its way around the device but is not acquainted to using apps.
	Advanced	The subject owns a mobile device, uses it for everything it is offered to him. Already has enough apps experience to criticize an app in terms of both presentation and performance regarding user experience.
	Expert	The same as "*Advanced*" but also has development experience and so is also capable to criticize from a developer's point of view.
BIS app knowledge	Basic	The subject might have used or uses one or two small BIS apps on a daily basis, besides the fact that would not consider such apps as BIS apps.
	Advanced	The subject knows what BIS apps are and understands the concept. Already has seen different types of BIS apps, might use one or two small BIS apps on a daily basis.
	Expert	The subject knows what BIS apps are and understands very well its concept. Already has seen different types of BIS apps, either small BIS apps or complex corporative BIS apps. Might use one or two small BIS apps on a daily basis in a personal manner and has experience in using complex corporation BIS apps.
Modelling knowledge	Basic	The subject has basic knowledge in modelling, might already use it to describe business processes.
	Advanced	The subject knows more than one modelling language, already has used more than one modelling language to describe both software and business processes.
	Expert	The subject knows many modelling languages, already has used them to describe both software and business processes. Has experience in modelling transformations and generative approaches are nothing new to him.

Table 22 – Participants universe and sample description

Characteristics	Participant	
	Available Sample	**Desired Universe**
Age	18 - 25	18 - 60
Sex	Both	Both
Field knowledge	Advanced and Expert	Basic, Advanced and Expert
BIS app knowledge	Basic and Advanced	Basic, Advanced and Expert
Modelling knowledge	Basic and Advanced	Basic, Advanced and Expert

We carried out two experiments to validate our approach. None of the subjects in our sample had ever seen our generated apps, and therefore had no idea of what they were going to do until the start of the experiment. Before starting a five minutes demonstration was given to everyone. This demonstration introduced the interaction paradigm, showing how to create, update, delete and associate objects. The app used in the demonstration was generated from a model that the subjects already knew, so that the complexity of the problem domain was not confounding factor. After the five minutes demo, an app (generated with a different model which the participants had never seen) a task was given to each participant, and they had thirty minutes to complete it.

- Experiment one – The subject was given a script and an app with no objects. The script told a story which ultimately described a database state. The subject was then asked to create and associate objects so that the final state of his database would describe the script.

- Experiment two – The subject was given an app with no objects. Then the subject was asked to reverse engineer the app. The expected output was a UML class diagram in the USE syntax format that would describe the app that he was seeing or thought that should be.

These two experiments were carried out in two different days. In the first day thirteen subjects performed the first experiment. In the background (i.e. not intrusively) every action they did in their apps was recorded and stored in their own database by the app itself. Also in the first day we did a pre-test of the second experiment. The reason why the second test had an empty database, was to expose the issues of cardinality constraints required for a correct understanding of the underlying model. All subjects used the same platform, an *Android* emulator, therefore blocking a possible influence of adopted device on the results. Running efficiency was not an issue when using the empty database, there was only a small delay when handling apps in the emulator environment.

We noticed, both in the first and second experiments, that the subjects did a lot of random clicks, apparently not associated with the task in hand, namely often performing useless navigations to other neighbor concepts (types). We believe that this behavior was due to the need to explore the provided app, by understanding how the display and navigation features look like.

Experiment one – app usage

The goal of this experiment was to assess our generated apps understandability 👪 regarding our proposed navigational paradigm and overall functionality 👪. A script was given to the subjects which ultimately represented a database state. The goal of the experiment was to reach that state using the given generated apps, i.e. filling the empty database with objects, so that it would represent the state described in the given script. The apps were generated from the Football Leagues model shown in Figure 47 in the sub section Models in the Appendix. This model was already a familiar model to the subjects, since they already had used it to do other recent academic assignments. Once the goal was explained to the subjects, a five minutes presentation was done to show the creation and association of objects in the app process. After that, the subjects had 30 minutes to complete the script.

To evaluate the results of this test, we followed a simple observation method, that is, during the realization of the test we observed and recorded the difficulties shown by the subjects. Since there were many doing the test at the same time, it was not feasible to follow the actions of every subject. Therefore we instrumented the generated app to record, non-intrusively (hidden from the subjects), every action executed in the app. Since our app uses the *Command* pattern to persist actions for later synchronization, the apps architecture already had scaffolding features to support such tracking purposes. For instance, every time the back button was pressed, we persisted a command with the, *CommandType* BACK. For other purposed actions we added other specific *CommandType*'s. In the end of the test, we asked the subjects to press the added *finish* button, which sent the used database to a server, so that we would be able to analyze it later.

After this experiment and analyzing the data we concluded that our generated apps lacked enough informative properties. For instance, we noticed several clicks over the navigation bar without having any object selected, we also noticed that the subjects were constantly trying to navigate to empty sets (i.e. trying to see some associated objects type screen when there were any associated objects of that type). The subjects also showed a lot of confusion when associating two types by means of an associative class. Lastly since we never prevented the navigation in WRITE mode, the subjects already confused, especially with the associative classes, would leave undone, occasionally, navigational actions. The latter faults were easily fixed after the first day tests, by simply adding alert messages for the later actions and preventing any kind of navigational action while any other action is undergoing (e.g. if the user enters in WRITE mode in any screen he will not be able to navigate forward until he either finishes the action or cancels it). In Figure 48 of the section Experiment one – result example in the Appendix, we can see how we could trough an UML sequence diagram better identify problematic user actions. The latter can be generated based upon the commands that we retrieved. In this scenario the *<Type>GUI* represents the activities or screens. The latter figure is already filtered of some actions and only shows a very small step, namely the creation of three objects (championship, participation and country) and their association actions based on the navigation. We did not provided more due to, as it can be seen, space constraints.

Experiment two – reverse engineering

In this experiment the subjects' goal was to reverse engineer the class diagram describing the underlying model, by just navigating in the app. The syntax of the output was one used by the USE tool. Subjects were proficient in it and used it to check model integrity. In Figure 45 (Models section of the Appendix) we can see the model used for this test.

In Table 23 we can see the results of this experiment. On the right column it shows the group score, that is the produced model percentage comparatively to the model used to generate the apps.

In order to not bias the results, since some model characteristics were impossible to retrieve from the app (e.g. association names), we carefully took out such variables from the model comparator program and only used variables which were possible to detect from the app.

To check if the team size (factor or independent variable) influenced the reversed model percentage (outcome or dependent variable), we will apply a between groups test. We have three groups, corresponding to team sizes of 1, 2 or 3 subjects. To determine the appropriate test (parametric or non-parametric), we have to check if the outcome variable is normally distributed. To test distribution adherence we will use the Shapiro-Wilks' W test. This test is the preferred test of normality because of its good power properties as compared to a wide range of alternative tests. The null hypothesis (H0) here is that there is no distance between the theoretical distribution and the sample distribution. The test interpretation is as follows:

- If the W statistic is significant (i.e. $p \leq \alpha$), then the hypothesis that the respective distribution is normal should be rejected.

In our case we obtained $p = 0.798$; therefore, for a test significance $\alpha = 0.05$ (95% confidence interval), we cannot reject H0, which means that we have no statistical evidence that the variable does not follow a Normal population. As a consequence, we can use a parametric test.

Since we have one factor with more than two treatments, we should use the One-factorial ANalysis Of Variance (One-Way ANOVA). This procedure is used to test the hypothesis that the means among several groups (determined by a factor variable) are equal. Therefore, it allows testing if there is a variance on the outcome variable (reversed model percentage) that is due to the factor (the team size).

The ANOVA compares the sum of the squares of the deviations between groups (difference between groups, SSB), with the sum of the squares within groups (SSW). The null hypothesis is tested using the following test statistic:

$$T = \frac{SSB/(k-1)}{SSW/(n-k)}$$

where n (number of cases) = 29 and k (number of groups) = 3.

Luís Silva – July, 2014

Table 23 – Experiment two results

Subjects per group	Reversed model (%)
2	23,6%
2	15,7%
2	23%
2	64,2%
2	42,4%
2	61,8%
2	41,8%
2	44,2%
2	40,3%
2	57%
2	26%
2	30%
1	57,2%
2	10%
2	46,3%
2	50,8%
2	6,6%
3	19,8%
3	32,5%
3	43,1%
2	29,9%
3	37,6%
2	38,8%
3	30,5%
2	15,2%
3	21,3%
3	55,8%
2	25,9%
1	36%

Under the null hypothesis, the T statistic follows an F (Snedecor) distribution with (k-1,n-k) degrees of freedom, i.e., $T \sim F_{(k-1,n-k)} = F_{(2,26)}$

We reject H0, for a given level of significance α, if the calculated value of the calculated F is greater than the upper critical value for the F distribution that can be found in a table with 3 entries[22]. In our case, if we use α=5%, we get:

$$F_{calc} > F_{1-\alpha\ (k-1,n-k)} = F_{95\%\ (2,26)} = 3.369$$

Since F_{calc} = 0.54, we cannot reject the null hypothesis, so we can say that:

There is no statistical evidence that allows denying that the team size influences the reversed model percentage.

---

[22] http://www.itl.nist.gov/div898/handbook/eda/section3/eda3673.htm

In Table 24 we can see the final results, we had a 35,43% produced model average, the largest produced percentage was 64,24% and the lowest was 6,6%.

Table 24 – Test two final results

	.use files	Average	Largest produced percentage	Lowest produced percentage
Number of tests	29	35,43%	64,2%	6,6%

Overall and based on these results we believe that our generated apps do represent very well the supplied models. Given the extreme conditions: (i) candidates never had used our generated apps; (ii) short five minute presentation of basic domain concepts; (iii) thirty minutes time frame limit to perform the experiment; (iv) and finally the relatively large size model used; a 35,43% average for model reverse is a very good indicator that the candidates understood their work environment, that is, the environment that the app was supposed to deliver.

8.2 – JUSE4Android

As shown in section one, we placed our approach under the category of *application generators*, the same category as one of the leaders in the *Gartner's magic quadrant*.

8.2.1 – Generator tool based on a PIM

While looking at *Gartner's magic quadrant* reproduced in Figure 2 (section 1.2 – Motivation), the use of the PIM is expected to provide support for multiple platforms, thus enhancing our "technology vision". Regarding our "focus on tomorrow", we expect that by following the visitor pattern in the app generator, we are better equipped to face the future extensibility ⚙ challenges. We reach the ability to execute by reducing the development cost and time, and increasing the quality due to reduced risk of human error, as shown in the next section.

8.2.2 – Code production – Time to market

As represented in Table 25, even for a moderate small sized model such as the *Projects World* example, our tool generated *Android* app code is considerably large, in both number of files and code length. If this source code were produced manually, it would certainly corroborate the "time-consuming app creation problem" that we referred to in the introduction (Parada and Brisolara 2012). Notice that for this specific small case scenario the tool transformed around 163 lines of PIM "code" (the UML class diagram representing the domain model) into 16877 lines of *Java* + XML, corresponding to the generated *Android* application.

Table 25 – Code generated – *ProjectsWorld* example

Layer	Type	Files	LOC
Business layer (Model)	Java	17 (10,2%)	4517 (26,8%)
Control layer (View-Model)	Java	4 (2,4%)	420 (2,5%)
Presentation layer (View)	XML	117 (70,1%)	3345 (19,8%)
	Java	28 (16,8%)	8365 (49,6%)
Persistency layer	Java	1 (0,6%)	230 (1,4%)
Total	Java + XML	167 (100,0%)	16877 (100,0%)

It is worth mentioning that more than two thirds of the source code relates to the presentation layer (see Table 25). This is mainly due to the need of supporting a considerable range of screen sizes and resolutions for both orientations that characterize the multiple mobile devices that run *Android* nowadays and data biding. Without adequate code generation facilities, like the one we presented herein, *Android* application programmers can face "massive" code development. By following a generative approach we were able to greatly reduce the development costs and time, and grant a good usability 👤 👥, as observed in two experiments.

8.2.3 – Scalability

To test the scalability of tool and also to enforce the aforementioned statements we tested our tool with considerable larger models, namely the *Royal & Loyal*, a middle scale model shown in Figure 46 (Models section of the Appendix), and the *BPMN2.0*, a very large model, which due to its size it is not represented in this dissertation.

In Table 26, the amount of generated files and lines of code (LOC) for the *Royal & Loyal* model.

Table 26 – Code generated – Royal & Loyal

Layer	Type	Files	LOC
Business layer (Model)	Java	26 (7,9%)	8682 (25,4%)
Control layer (View-Model)	Java	4 (1,2%)	427 (1,2%)
Presentation layer (View)	XML	254 (77,0%)	8568 (25,0%)
	Java	45 (13,6%)	16339 (47,7%)
Persistency layer	Java	1 (0,3%)	229 (0,7%)
Total	Java + XML	330 (100,0%)	34245 (100,0%)

In Table 27 we can see the amount of generated code for the *BPMN2.0* model. Notice that, percentually, as we increase the size of the model the presentation layer also increases its presence within the LOC, while maintaining a similar number of files growth.

Table 27 – Code generated – BPMN 2.0

Layer	Type	Files	LOC
Business layer (Model)	Java	314 (8,8%)	108477 (16,3%)
Control layer (View-Model)	Java	4 (0,1%)	1687 (0,3%)
Presentation layer (View)	XML	2665 (74,3%)	167252 (25,1%)
	Java	604 (16,8%)	388293 (58,2%)
Persistency layer	Java	1 (0,0%)	924 (0,1%)
Total	Java + XML	3588 (100,0%)	666633 (100,0%)

This means that the effort for data representation and data binding increases with the growth of the models. This validates our decision to use simple annotations to obtain the attributes that would be used in the presentation layer but, does not validate the fact that a more specific set of rules should be available to increase the model descriptive capabilities. The conclusion that we may take from this scalability test is the fact that these rules, if implemented, should be specified without the need to reference every specific data types present in the model. Instead generic rules should be applied to all the model, except if a specific target is explicitly set. Therefore avoiding a scalable specification effort.

9 – Conclusions and Future Work

9.1 – Conclusions

In conclusion, by combining several open-source tools, we were able to address our given main problem – **How to reduce development time and cost by transforming a given domain model into an *Android* BIS application?**

We used the COCOMOII[23] tool to calculate the estimated gained value in both time and cost for our *ProjectsWorld* case study, based on the generated LOC. We set its parameters to simulate a best-case scenario. For instance, we left every parameter with the default nominal value and set the following parameters settings:

- Required Software Reliability – very low.
- Data Base Size – low.
- Product Complexity – very low.
- Developed for Reusability – very low
- Documentation Match to Lifecycle Needs – very low

We also left the default value for every other feature checking turned off and set the *Cost per Person-Month (Dollars)* with 500$. Thus granting a possible result outcome of our approach for this case study. The tool showed a needed effort of 28 person per month, a required 11 months' time frame with a cost of 13891$.

Therefore by using our tool we could decrease time and cost in the development process by 11 months and 13891$.

As aforementioned, our generative approach is implemented on top of the *USE* tool. Since the latter is a standalone tool, with a GUI and other components of its own that we do not require for our model-driven approach, we used and improved a façade component (*J-USE*) for *USE* that allows accessing its services conveniently. We dubbed *JUSE4Android* our generator that, from a domain model, generates a full working *Android app* supporting the proposed model-based navigation

---

[23] http://csse.usc.edu/tools/COCOMOII.php

metaphor. The visitor pattern was used in the code generator component to allow maintainability ⚙ and extensibility ⚙.

The MVVM-based architecture of the generated *Android* BIS apps, grants a separation of concerns through loosely coupled layers (business layer, presentation layer, control layer, persistence layer) which increases maintainability ⚙. While we could not find any related work applying MVVM in the context of *Android* we strongly believe that our architecture is a good choice by comparison to other mobile related implementations, namely by granting a *"strong separation between data, behavior, and presentation, making it easier to control the chaos that is software development"* (Smith 2009).

As for the paradigm, our approach uses the object paradigm all the way through, from user navigation to objects persistence, thus achieving the seamlessness desideratum. The latter was made possible by using an object database (*Versant DB4O*) where the persistence programming paradigm is the same as in the remaining generated app code. We showed how easy it can be to do the mapping of an UML class diagram in order to guarantee persistence. As showed in (Ambler 2013), this is a much better approach than the recurrent alternative where a relational database is used (e.g. *SQLite*), thus requiring the usage of some ORM middleware or hard-coded transformations in the app code that decreases its understandability ⚙👤 and hampers extensibility ⚙.

Finally we showed our proposed GUI generation principles and navigability approach, aiming at producing BIS apps. As shown in (Pires Silva and Brito e Abreu 2014), we do not require the description of every possible scenario to generate a lot of screen sizes for both orientations and different devices running *Android*. We have shown how easily we can change one view for all possible configurations, based upon a template, thus avoiding "massive" code development. We accessed our model-to-app transformation understandability 👤, by performing a reverse engineering experiment (app-to-model) on real candidates. The latter results also showed high understandability 👤👥.

9.2 – Future Work

A set of problems are open for future research, as described in (Pires da Silva 2014; Pires da Silva and Brito e Abreu 2014) and in the following subsections.

9.2.1 – Systematic comparison/Software evolution

We plan to perform a systematic comparison on available generative approaches for *Android,* by using the same initial model as input and then assess the effect of requirements volatility. We are particularly concerned with the efforts required for:

(i) producing the input specification;

(ii) generating a baseline app;

(iii) adding extra requisites or removing existing requisites from the baseline app;

(iv) understanding the code of generated apps for maintenance sake.

9.2.2 – Business rules enforcement

BIS applications require the definition of business. The latter can be as simple as setting a lower limit for marriage age or as complex as the preconditions for granting a bank loan or being refunded by an insurance in case of an auto collision. Thus, any BIS app generative approach will be incomplete unless that support is provided. At the model side we will enrich our UML class diagrams with OCL clauses to specify the required BIS rules. Some interesting research problems then arise regarding where those rules will be verified (client or server) and how to grant state consistency on a distributed environment. If every verification is only done in the server side, the user will only go as far as the simplest operation since none of its data is valid until the server validates it, performance wise, such verifications in both client and server side may be considered avoidable. Another issue will be generating automatic error dialogs with context-sensitive help. Lastly all of the latter concerns must be researched, having the maintainability feature as our main unaffected feature, which in turn also arise another problem, given our mobility factor and the greater changeability of the business rules, i.e. in comparison to the model itself, will OCL have enough specification characteristics to allow differentiating a more temporarily rule from a more perdurable rule, in order to decide in which side this rule should definitely be.

9.2.3 – Scalability

Regarding the scalability issue and its effect on performance. We plan to produce a model-driven workload generator for our BIS apps. Taking our PIM as input we will perform a transformation to generate a

The user will be queried to indicate the desired cardinalities for the instances of each concept in the domain (i.e. number of objects for each class and number of lines for each association). The latter will then allow generating USE objects. A second transformation, using reflection techniques will generate POJO objects from the USE objects. This workload generator will then be used to develop scalability experiments that we expect to automate as much as possible. In fact, we also plan to generate a performance evolution (benchmarking) test battery that will profile the resources spent on performing CRUD and navigation operations model-wise.

Besides scalability degradation curves we expect to identify the component/layers responsible for performance bottlenecks. For instance, the integration of multi-threading capabilities, i.e. the clearly separation of background process work versus UI thread work thus making the app more responsive and therefore providing greater usability. The latter can be implemented without the need of any model specification techniques but, such implementations may affect the already established architecture, namely we should validate that the DB4O can handle multi-threading (multi-tasking) in a combination with user actions and still maintain consistent local data.

9.2.4 – Internationalization

Regarding internationalization, we need to provide support for different languages at the GUI side. As shown previously, we can change the layout in a single XML (corresponding to a domain entity) and that change will be propagated to all screen sizes and orientations. We intend to apply the same approach to the language support, since different languages will require different styles to adjust, namely in size, due to different average word lengths and desired verbosities (e.g. in contextual help).

9.2.5 – Portability

Finally, we also intend to enable the generative capabilities to other mobile platforms such as *iOS* and *Windows Mobile*. This issue raises a set of problems, namely the possible different requirements of each different platform, which in turn will require a PSM. Thus, a deeper research in MDD is needed, like applying language engineering transformation techniques (Santiago et al. 2012) in two-steps. In the first step (model-to-model transformation) we will go from the PIM to a PSM, corresponding to the desired platform (e.g. *iOS, Windows mobile* device). In the second step (model-to-code transformation) we will take as input the PSM and generate source code for the client-side (the BIS app running on the users' mobile device). The code generation process for the server-side will hopefully be performed directly from the PIM, since we cannot envisage dependencies on the platform. The portability of the DB4O component on all the required mobile platforms is also an issue here, but we expect the *Versant* open-source community [24] will find a solution for that problem

9.2.6 – Reliability

Our reliability tests, described in section 8 – Validation, were not exhaustive. First we plan to generate a JUnit test battery (white box testing approach) that will provide a 100% coverage of the business layer (measured with the Eclemma[25] plugin). This will be great help for developers extending specific features manually. Second, we want to address the reliability of the presentation layer by using a black-box approach that will exercise the GUI extensively. We will survey existing techniques such as "Monkey-Testing" (Amalfitano et al. 2012; Hu and Neamtiu 2011; Takala et al. 2011) and model-driven techniques such as the one proposed in (Barbosa et al. 2011; Cunha et al. 2010).

---

[24] http://community.versant.com/
[25] http://www.eclemma.org/

Bibliography

ALP BALKAN, AHMET, OGUZ KARTAL AND BERKER PEKSAG. ORMAN - Lightweight and practical ORM in Java for your Android apps. 2012. Available at: https://github.com/ahmetalpbalkan/orman.

AMALFITANO, DOMENICO, ANNA RITA FASOLINO, PORFIRIO TRAMONTANA, SALVATORE DE CARMINE, et al. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering.* ACM, 2012, p. 258-261.

AMBLER, SCOTT. Agile Techniques for Object Databases. 2013. Available at:http://www.db4o.com/about/productinformation/whitepapers/db4o%20Whitepaper%20-%20Agile%20Techniques%20for%20Object%20Databases.pdf.

AUMASSON, A, V BONNEAU, T LEIMBACH AND M GÖDEL. Economic and Social Impact of Software & Software-Based Services. Cordis (Online): 2010. Available at: http://cordis.europa.eu/fp7/ict/ssai/docs/study-sw-report-final.pdf.

BARBOSA, ANA, ANA C. R. PAIVA AND JOSÉ CREISSAC CAMPOS. Test case generation from mutated task models. In *Proceedings of the 3rd ACM SIGCHI symposium on Engineering interactive computing systems.* ACM, 2011, p. 175-184.

BASOLE, RAHULC AND JÜRGEN KARLA. On the Evolution of Mobile Platform Ecosystem Structure and Strategy. Business & Information Systems Engineering, 2011, 3(5), 313-322. Available at: http://dx.doi.org/10.1007/s12599-011-0174-4.

BRITO E ABREU, FERNANDO. J-USE. Version: 1.0. Google Code: Google, 2011. Available at: https://code.google.com/p/j-use/.

CODD F. , EDGAR. A relational model of data for large shared data banks. Communications of the ACM, 1970, 13(6).

CROCKFORD, DOUGLAS, JSON, Available at: http://www.json.org/.

CUNHA, MARCO, ANA C. R. PAIVA, HUGO SERENO FERREIRA AND RUI ABREU. PETTool: A pattern-based GUI testing tool. In *International Conference on Software Technology and Engineering.* San Juan, PR: IEEE, 2010, vol. 1.

D. HOLSTEIN, BEERY. Speed delivery of Android devices and applications with model-driven development. 2011. Available at:http://www.ibm.com/developerworks/rational/library/model-driven-development-speed-delivery/model-driven-development-speed-delivery-pdf.pdf.

DEMUTH, BIRGIT. DresdenOCL:Documentation 2013. Available at: http://www.dresden-ocl.org/index.php/DresdenOCL:Documentation.

ECLIPSE, Eclipse market place, Available at: http://marketplace.eclipse.org/.

FABIAN, BUETTNER, HAMANN LARS AND HOFRICHTER OLIVER. USE: UML-based Specification Environment. Version: 3.0.6. http://sourceforge.net/projects/useocl/, 2013. Available at: http://sourceforge.net/projects/useocl/.

FOWLER, MARTIN, Presentation Model, Available at: http://martinfowler.com/eaaDev/PresentationModel.html.

GIESE, PHILIPP. androrm. 2012. Available at: http://www.androrm.com/.

GOGOLLA, MARTIN, FABIAN BUTTNER AND MARK RICHTERS. USE: A UML-based specification environment for validating UML and OCL. Science of Computer Programming, December 2007, 69(1-3), 27–34. Available at: http://dx.doi.org/10.1016/j.scico.2007.01.013.

GOOGLE, Android Developers, Available at: http://developer.android.com/.

GOOGLE, Google Cloud SQL, Available at: https://developers.google.com/cloud-sql/.

GOSSMAN, JOHN, Model-View-ViewModel, Available at: http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx.

GREHAN, RICK. The Database Behind the Brains. 2006. Available at:http://www.db4o.com/about/productinformation/whitepapers/#database_brains.

GUY, ROMAIN AND ADAM POWELL, The world of ListView Google, 2010, Multimedia Online at: http://www.youtube.com/watch?v=wDBM6wVEO70

HAYWOOD, DAN, Apache Isis, Available at: http://isis.apache.org/.

HEVNER, ALAN AND SAMIR CHATTERJEE. Design Research in Information Systems: Theory and Practice. In *Design Research in Information Systems: Theory and Practice.* Springer, 2010.

HU, CUIXIONG AND IULIAN NEAMTIU. Automating GUI testing for Android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test.* ACM, 2011, p. 77-83.

HUNTER, JASON, ROLF LEAR AND BRETT MCLAUGHLIN, JDOM, Available at: http://www.jdom.org/.

IBM. Rational Rhapsody. 2013. Available at: http://www-03.ibm.com/software/products/us/en/ratirhapfami.

JUNGINGER, MARKUS AND VIVIEN DOLLINGER. greenDAO – Android ORM for SQLite. 2013. Available at: http://greendao-orm.com/.

JUNIOR, ASSIS. jpa-android. 2011. Available at: http://code.google.com/p/jpa-android/.

KOMATINENI, SATYA AND DAVE MACLEAN. *Pro Android 4*. 4th ed.: Apress, 2012.

LCI_TEAM. OCLE. Version: 2.0.4. 2005. Available at: http://lci.cs.ubbcluj.ro/ocle/overview.htm.

MEDNIEKS, ZIGURD, LAIRD DORNIN, BLAKE MEIKE AND MASUMI NAKAMURA. *Programming Android*. 2nd ed.: O'Reilly Media, 2012.

MIT. App Inventor. 2013. Available at: http://appinventor.mit.edu/.

OBJECT_MANAGEMENT_GROUP, MDA - Model-Driven Architecture Available at: http://www.omg.org/mda/.

OWENS, MIKE AND GRANT ALLEN. *The Definitive Guide to SQLite*. 2nd ed.: Apress, 2010.

PAGE, MARK, MARIA MOLINA AND GORDON JONES. The Mobile Economy. London, United Kingdom 2013.

PARADA, ABILIO G. AND LISANE B. BRISOLARA 2012. A Model Driven Approach for Android Applications Development. In *Proceedings of the Brazilian Symposium on Computing System Engineering (SBESC'2012)*, Natal, Brazil2012, 192-197.

PARR, TERENCE. A Functional Language For Generating Structured Text. 2006. Available at: http://www.cs.usfca.edu/~parrt/papers/ST.pdf.

106

PAWSON, RICHARD. Naked objects. Doctor University of Dublin, Trinity College, 2004, Available at: http://isis.apache.org/intro/learning-more/Pawson-Naked-Objects-thesis.pdf.

PAWSON, RICHARD, Naked objects, Available at: http://nakedobjects.codeplex.com/.

PIRES DA SILVA, LUÍS 2014. A Model-Driven Approach for Mobile Business Information Systems Applications. In *Proceedings of the MODELS'2014 Doctoral Symposium*, Valencia, Spain2014 CEUR.

PIRES DA SILVA, LUÍS AND FERNANDO BRITO E ABREU 2014. A MDE Generative Approach for Mobile Business Apps. In *Proceedings of the Simpósio de Estudantes de Doutoramento em Engenharia de Software (SEDES'2014)*, Guimarães, Portugal2014 IEEE Computer Society.

PIRES SILVA, LUÍS AND FERNANDO BRITO E ABREU 2014. Model-Driven GUI Generation and Navigation for Android BIS Apps. In *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD'2014)*, Lisbon, Portugal2014, P. LISBON ed. SCITEPRESS Digital Library, 400-407.

POCATILU, PAUL. Building Database-Powered Mobile Applications. Informatica Economică, 2012, 16, 132-141.

POLEVOY, IGOR. ActiveJDBC. 2012. Available at: http://javalite.io/activejdbc.

RAMAGE, RYAN, umlc, Available at: https://code.google.com/p/umlc/.

RIZA BABAOĞLAN, ALI. Gartner's 2013 Magic Quadrant for Mobile Application Development Platforms. 2013. Available at:http://www.alibabaoglan.com/blog/gartner-2013-magic-quadrant-mobile-application-development-platforms/.

ROQUE, RICAROSE. OpenBlocks: An Extendable Framework for Graphical Block Programming Systems. Master Massachusetts Institute of Technology, 2007, Available at: http://dspace.mit.edu/bitstream/handle/1721.1/41550/220927290.pdf?sequence=1.

SANTIAGO, IVÁN, ÁLVARO JIMÉNEZ, JUAN MANUEL VARA, VALERIA DE CASTRO, et al. Model-Driven Engineering as a new landscape for traceability management: A systematic literature review December 2012, 54, 1340–1356. Available at: http://www.sciencedirect.com/science/article/pii/S0950584912001346.

SINGH, INDERJEET, JOEL LEITCH AND JESSE WILSON, GSON, Available at: http://code.google.com/p/google-gson/.

SMITH, JOSH. WPF Apps With The Model-View-ViewModel Design Pattern. *MSDN Magazine* [Type of Work]. 2009. Available at:http://msdn.microsoft.com/en-us/magazine/dd419663.aspx.

SUEZ, EITAN, JMatter, Available at: http://www.jmatter.org/.

TAKALA, T, M KATARA AND J HARTY. Experiences of System-Level Model-Based GUI Testing of an Android Application. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on.* Berlin: IEEE, 2011, p. 377 - 386.

UZIEL, EREL. Basic4android. 2013. Available at: http://www.basic4ppc.com/.

VERSANT, db4o Reference Documentation, Available at: http://community.versant.com/documentation/reference/db4o-8.0/java/reference/.

VERSANT, db4objects, Available at: http://www.db4o.com/.

VOGEL, LARS. Vogella - Expert Android and Eclipse development knowledge 2013. Available at: http://www.vogella.com/.

WARMER, JOS AND ANNEKE KLEPPE. *Object Constraint Language, The: Getting Your Models Ready for MDA, Second Edition*. ed.: Addison Wesley, 2003. 240 p.

WATSON, GRAY. OrmLite - Lightweight Object Relational Mapping (ORM) Java Package. 2013. Available at: http://ormlite.com/.

WIKIPEDIA, Flat file database, Available at: http://en.wikipedia.org/wiki/Flat_file_database.

WIKIPEDIA, Object database, Available at: http://en.wikipedia.org/wiki/Object_database.

WIKIPEDIA, Relational database management system, Available at: http://en.wikipedia.org/wiki/Relational_database_management_system.

WIKIPEDIA, XML database, Available at: http://en.wikipedia.org/wiki/XML_database.

WIKIPEDIA, Android (operating system), Available at: http://en.wikipedia.org/wiki/Android_%28operating_system%29.

WIKIPEDIA, Command pattern, Available at: http://en.wikipedia.org/wiki/Command_pattern.

WIKIPEDIA, Computer-aided software engineering, Available at: http://en.wikipedia.org/wiki/Computer-aided_software_engineering.

WIKIPEDIA, Observer pattern, Available at: http://en.wikipedia.org/wiki/Observer_pattern.

WILKE, CLAAS. Java Code Generation for Dresden OCL2 for Eclipse. Master Technische Universität Dresden, 2009, Available at: http://www.claaswilke.de/publications/study/beleg.pdf.

108

Appendix

Luís Silva – July, 2014

A. Projects World – USE specification

```
-----------------------------------------------------------------------
model ProjectsWorld
-----------------------------------------------------------------------
enum ProjectSize {small,medium,big}
enum ProjectStatus {planned, active, finished, suspended}


@list(year="1",month="2",day="3")
@creation(year="1",month="2",day="3")
@display(year="1",month="2",day="3")
@unique(year="1",month="2",day="3")
@domain()
class CalendarDate
      attributes
            day: Integer
            month: Integer
            year: Integer
end

-----------------------------------------------------------------------

@StartingPoint(NameToDisplay="Qualifications",
ImageToDisplay="qualification")
@list(description="1")
@creation(description="1")
@display(description="1")
@unique(description="1")
@domain()
class Qualification

      attributes
            description : String
end
-----------------------------------------------------------------------
@StartingPoint(NameToDisplay="Companies", ImageToDisplay="company")
@list(name="1")
@creation(name="1")
@display(name="1")
@unique(name="1")
@domain()
class Company

      attributes
            name : String
end
```

```
@StartingPoint(NameToDisplay="Workers", ImageToDisplay="worker")
@list(nickname="1")
@creation(nickname="1",salary="2")
@display(nickname="1",salary="2")
@unique(nickname="1",salary="2")
@domain()
class Worker

     attributes
            nickname: String
            salary: Integer
end
----------------------------------------------------------------------
@StartingPoint(NameToDisplay="Projects", ImageToDisplay="project")
@list(name="1")
@creation(name="1",size="2",status="3",months="2",start="3")
@display(name="1",size="2",status="3",months="4",start="5")
@unique(name="1",size="2",status="3",months="4",start="5")
@domain()
class Project

     attributes
            name : String
            size : ProjectSize
            status : ProjectStatus
            months: Integer
end
----------------------------------------------------------------------
@StartingPoint(NameToDisplay="Training Courses",
ImageToDisplay="training")
@list(name="1", size="2")
@creation()
@display()
@unique()
@domain()
class Training < Project
     attributes

end
----------------------------------------------------------------------
@list(startDate="1", endDate="2")
@creation(startDate="1", endDate="2")
@display(startDate="1", endDate="2")
@unique(startDate="1", endDate="2")
@domain()
associationclass Member
     between
            Project[0..*] role projects
            Worker[1..*] role members
     attributes
            startDate: CalendarDate
            endDate: CalendarDate
end
```

```
composition CarriesOut between
 Company[1]
 Project[0..*] role projects
end

association Employs between
 Company[0..1] role employer
 Worker[1..*] role employees
end

association IsQualified between
 Worker[0..*] role workers
 Qualification[1..*] role qualifications
end

association Requires between
 Project[0..*] role projects
 Qualification[1..*] role requirements
end

association Trains between
  Training[0..*] role trainings
  Qualification[1..*] role trained
end
```

B. Persistency – DB4O

1.Usage

Since it uses *Java*, *DB4O* seamlessly integrates itself in *Android,* with the exception of only one or two needed *Android* specific commands. Let us see what commands are specific and see some simple commands.

First we need to prepare our database. To do this we must supply a path, a name and the configuration settings that we want to use. In order to create, a database or just open it if already created, an *ObjectContainer* must be created through the following method:

```
ObjectContainer oc = Db4oEmbedded.openFile(dbConfig(),
db4oDBFullPath(context));
```

We also need to give a path. In *Android* there are several ways of doing storage, in this case we are using the internal memory but the external memory can also be used, namely from an SD Card which if it would be the case the necessary code for setting the path, would be a little different but everything else is the same. Another note regarding this code, is the Context type. This is a specific *Android* type, which in this case is always needed to get access to files system.

```
public static String db4oDBFullPath(Context ctx) {
   return ctx.getDir("data", Context.MODE_PRIVATE) + "/" + DataBaseName +
DataBaseExtension;
}
```

Finally we set any configuration we need, in this case and since is for *Android* we add the "*AndroidSupport*" feature, so *DB4O* better adapt to the system.

```
public static EmbeddedConfiguration dbConfig() {
   EmbeddedConfiguration configuration = Db4oEmbedded.newConfiguration();
   configuration.common().add(new AndroidSupport());
   configuration.file().lockDatabaseFile(false);
   return configuration;
}
```

To do any operation we must use the *ObjectContainer* variable that we initialized before, here it will be showed the simpler ones. For storage purposes, that is for the typical commands INSERT and UPDATE it is only used the store operation. If the object is not already in the database it will be inserted, otherwise is updated to the new state. Regarding the update, the *DB4O* uses its own reference system, so, to be safe, we should always make sure that we are working with the intended object. Consider for instance that we are using adapters to represent lists. It might happen that we may be working with an object stored in the adapter and the reference to the *DB4O* was lost meanwhile, due to garbage collecting. In order to make sure we have the proper object before we make any operation we query the database for the object in question.

```
oc.store(object);
```

For DELETE purposes we just have to call the operation delete.

```
oc.delete(object);
```

After we are done with all the operations, we can call commit in order to persist all changes we have made. We can also close the session with one database by calling the close operation.

```
oc.commit();
oc.close();
```

Finally, we can rollback any changes that were already stored but not committed by calling the rollback operation.

```
oc.rollback();
```

2.DB4O vs SQLite – seamlessness

In this example, let us consider the classes *Qualification* and *Worker*. Let us also assume that for the SQLite case scenario we already have all the tables defined and created.

To create, associate and persist objects with SQLite we would do like the following:

```
INSERT INTO qualification VALUES("1","programmer");
INSERT INTO worker VALUES("1","Bob the Builder","200");
```

The first parameter would be for identifying the association. But in a relational approach we would also need a third table, therefore we would also need to have a, let us call it, "qualification_worker" table. Now we would be able to persist the link like so:

```
INSERT INTO qualification_worker VALUES("1","1");
```

After this action we would just need to commit, to have the data persisted.

The same actions in the DB4O would look like so:

```
Qualification qualification = new Qualification("programmer");
Worker worker = new Worker("Bob the Builder", 200);
```

Now we just need to call either the add qualification or add worker for one of the latter like so:

```
qualification.addWorkers(worker);
```

or

```
worker.addQualification(qualification);
```

If the called function belongs to the holder class it adds the object to the instance list. Otherwise it calls the other method. In this scenario the worker class is the holder therefore each method would look like the following:

In the Worker class:

```
/******************************************************************
 * MANY2MANY single setter for Worker[*] <-> Set(Qualification)[*]
 * @param qualification the qualification to add
 ******************************************************************/
public void addQualifications(Qualification qualification)
{
    this.qualifications.add(qualification);
}
```

In the Qualification class:

114

```
/****************************************************************
* MANY2MANY single setter for Qualification[1..*] <-> Worker[*]
* @param worker the worker to add
****************************************************************/
public void addWorkers(Worker worker)
{
      worker.addQualifications(this);
}
```

So given that the *Worker* class is the holder. Using the, previously explained, already opened ObjectContainer (oc), we just need to store the object and commit.

```
oc.store(worker);
oc.commit();
```

Using the *Querybyexample* to read, update and delete objects

READ:

SQLite:

```
SELECT * FROM qualification WHERE name="programmer";
```

DB4O:

```
Qualification qualification = new Qualification("programmer");
ObjectSet results = oc.queryByExample(qualification);
```

UPDATE:

SQLite:

```
UPDATE qualification SET description = "medic" WHERE name = "programmer";
```

DB4O:

```
ObjectSet result = oc.queryByExample(new Qualification ("Transactions"));
Qualification qualification = (Qualification) result.next();
```

Then we change the data through a method call on the retrieved object. The updated object is then stored with a call to set:

```
qualification.setName("medic");
container.store(qualification);
```

DELETE:

SQLite:

```
DELETE FROM qualification WHERE name = "medic";
```

DB4O:

```
ObjectSet result = oc.queryByExample(new Qualification ("medic"));
Qualification qualification = (Qualification) result.next();
container.delete(qualification);
```

C. Static Generation Process – identifiers

1.Utils layer

Classes	Package Identifiers		Other identifiers	
	Identifier	Used to access	Identifier	Replace
AndroidTransaction	DATABASE_PACKAGE	Database	none	-
Command	none	-	none	-
CommandTargetLayer	none	-	none	-
CommandType	none	-	none	-
DetailFragment	none	-	none	-
FragmentMethods	none	-	none	-
InheritanceListFragment	none	-	none	-
LauncherGridViewAdapter	none	-	none	-
ListAdapter	none	-	none	-
ListFragmentController	TARGET_PACKAGE	R	MAIN_APPLICATION	Application class
	UTILS_PACKAGE	ListAdapter		
		PropertyChangeEvent		
		PropertyChangeListener		
	BUSINESS_PACKAGE	ModelMusts		
ListViewHolder	none	-	none	-
ModelContracts	none	-	none	-
ModelMusts	BUSINESS_PACKAGE	CommandType	none	-
NavigationBarFragment	none		none	-
PropertyChangeEvent	UTILS_PACKAGE	CommandType	none	-

Luís Silva – July, 2014

PropertyChangeListener	none		none	-
ServerActions	TARGET_PACKAGE	MAIN_APPLICATION	none	-
	DATABASE_PACKAGE	Database		
	BUSINESS_PACKAGE	ModelMusts		
	UTILS_PACKAGE	UtilNavigate		
ServerInfo	none	-	GENERATION-IP	Given IP address
			GENERATION-PORT	Given Port
			GENERATION-USER	Given user name
			GENERATION-PASS	Given password
StartServer	none	-	none	-
StopServer	none	-	none	-
Transactions	DATABASE_PACKAGE	Database	none	-
	BUSINESS_PACKAGE	ModelMusts		
UtilNavigate	UTILS_PACKAGE	InheritanceListFragment	none	-
		WarningDialogFragment		
Utils	none	-	none	-
WarningDialogFragment	TARGET_PACKAGE	R	none	-

2.View layer

XML files	folder
actionbar_compat_item	drawable
actionbar_compat_item_focused	drawable
actionbar_compat_item_pressed	drawable
default_list_selector	drawable
<model name>_launcher_gridview_round_borders	drawable
<model name>_launcher_gridview_selector	drawable
navigationbar_association_new_object_state	drawable
navigationbar_divider	drawable
navigationbar_error_state	drawable
navigationbar_new_object_state	drawable
navigationbar_original_state	drawable
navigationbar_selector_error	drawable
navigationbar_selector	drawable
default_blank_fragment	layout
default_navigationbar	layout
default_okcancel_buttons	layout
default_warning_fragment	layout
<model name>_launcher_activity	layout
<model name>_launcher_gridview_row	layout
menu_launcher	menu
menu_read	menu
menu_write	menu
colors	values

Luís Silva – July, 2014

D. POJO – Relational getters and setters example

Generated many-to-many association between the class *Worker* and the class *Project* given that we have the Member as an associative class. In this scenario the *Member* class is the holder class and therefore holds the instances of both *Project* and *Worker* neighbours.

Worker class code to represent the association between himself and *Project*:

```
/*********************************************************************
* MEMBER2MEMBER getter for Worker[1..*] <-> Project[*]
* @return the projects of the members
*********************************************************************/
public Set<Project> projects()
{
      Set<Project> result = new HashSet<Project>();
      for (Member x : Member.allInstances())
            if (x.members()  ==  this && x. projects() != null)
                  result.add(x.projects());
      return result;
}

/*********************************************************************
* MEMBER2MEMBER setter for Worker[1..*] <-> Project[*]
* @param projects the projects to set
*********************************************************************/
public void setProjects(Set<Project> projects)
{
for (Project t : projects)
            for (Member x : Member.allInstances())
                  if (x.members() == this)
                        x.setProjects(t);
}
```

Projects class code to represent the association between himself and *Worker*:

```
/*********************************************************************
* MEMBER2MEMBER getter for Project[*] <-> Worker[1..*]
* @return the members of the projects
*********************************************************************/
public Set<Worker> members()
{
      Set<Worker> result = new HashSet<Worker>();
      for (Member x : Member.allInstances())
            if (x.projects()  ==  this && x. members() != null)
                  result.add(x.members());
      return result;
}
```

Luís Silva – July, 2014

```
/************************************************************************
* MEMBER2MEMBER setter for Project[*] <-> Worker[1..*]
* @param members the members to set
************************************************************************/
public void setMembers(Set<Worker> members)
{
      for (Worker t : members)
            for (Member x : Member.allInstances())
                  if (x.projects() == this)
                        x.setMembers(t);
}
```

Member class code to represent the association between himself and both *Project* and *Worker*:

```
/************************************************************************
* ASSOCIATIVE2MEMBER getter for Member[*] <-> Project[1]
* @return the projects of the member
************************************************************************/
public Project projects()
{
      return projects;
}

/************************************************************************
* ASSOCIATIVE2MEMBER setter for Member[*] <-> Project[1]
* @param projects the projects to set
************************************************************************/
public void setProjects(Project projects)
{
      this.projects = projects;
}

/************************************************************************
* ASSOCIATIVE2MEMBER getter for Member[*] <-> Worker[1]
* @return the members of the member
************************************************************************/
public Worker members()
{
      return members;
}

/************************************************************************
* ASSOCIATIVE2MEMBER setter for Member[*] <-> Worker[1]
* @param members the members to set
************************************************************************/
public void setMembers(Worker members)
{
      this.members = members;
}
```
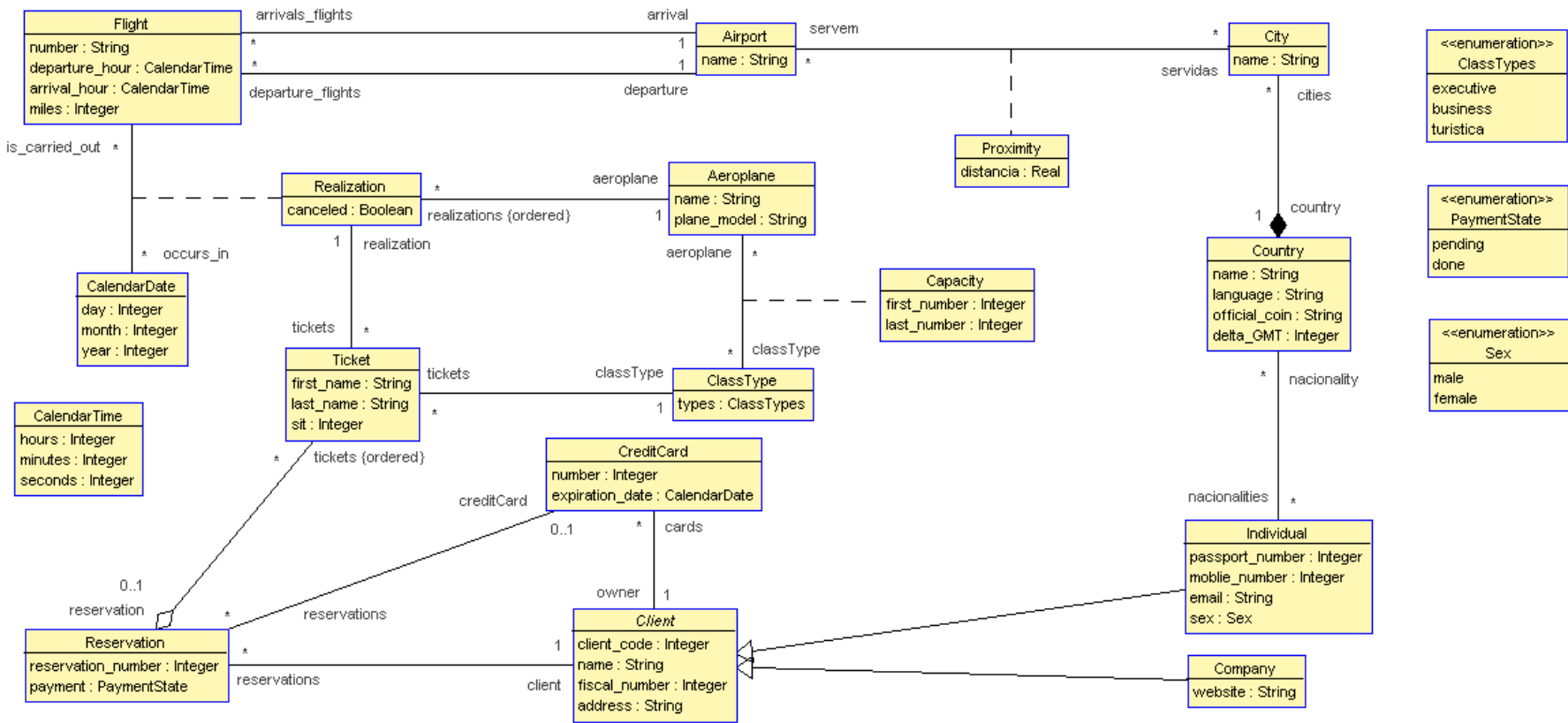
E. Models

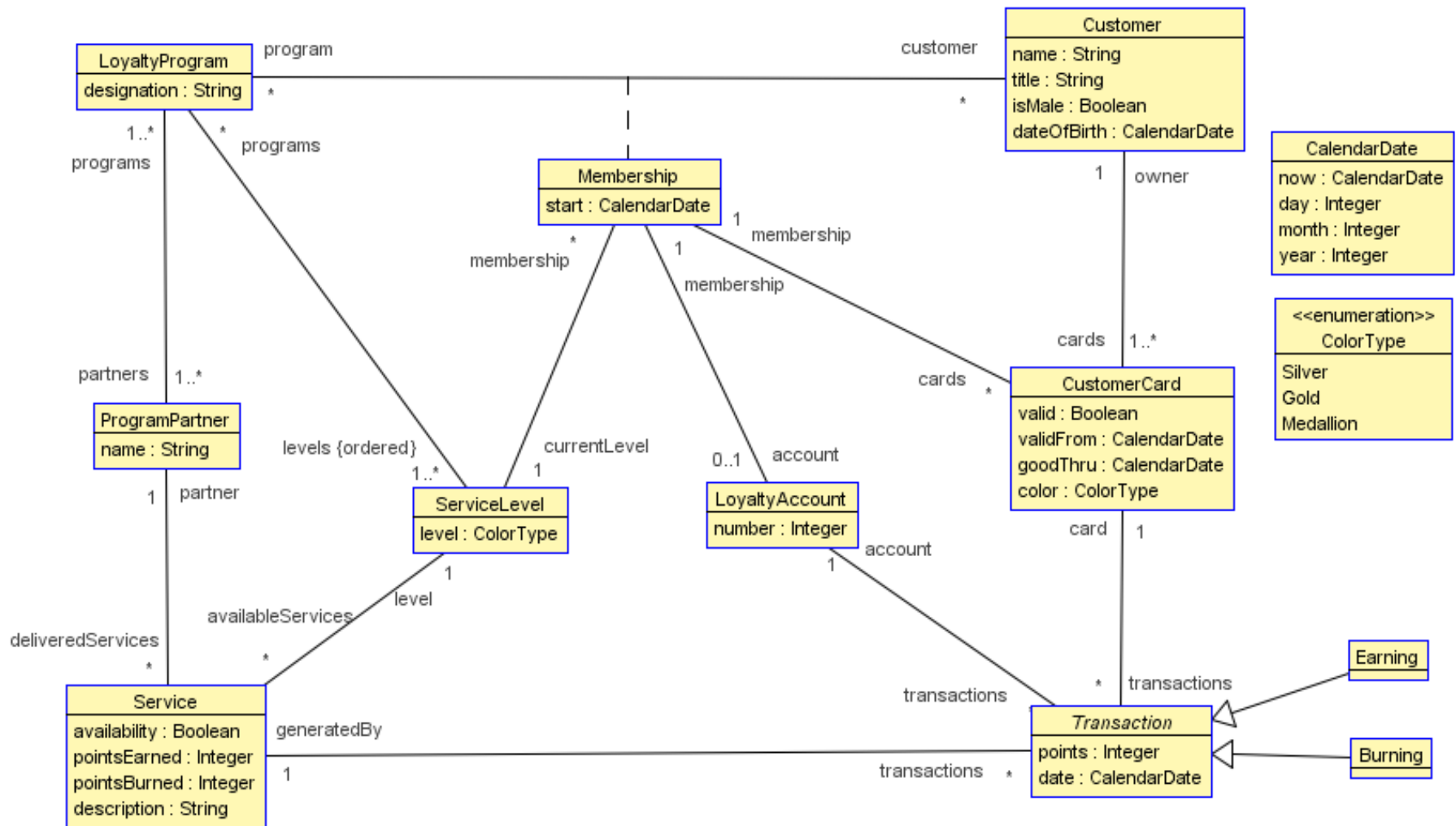Figure 45 – *AirNova* UML class diagram

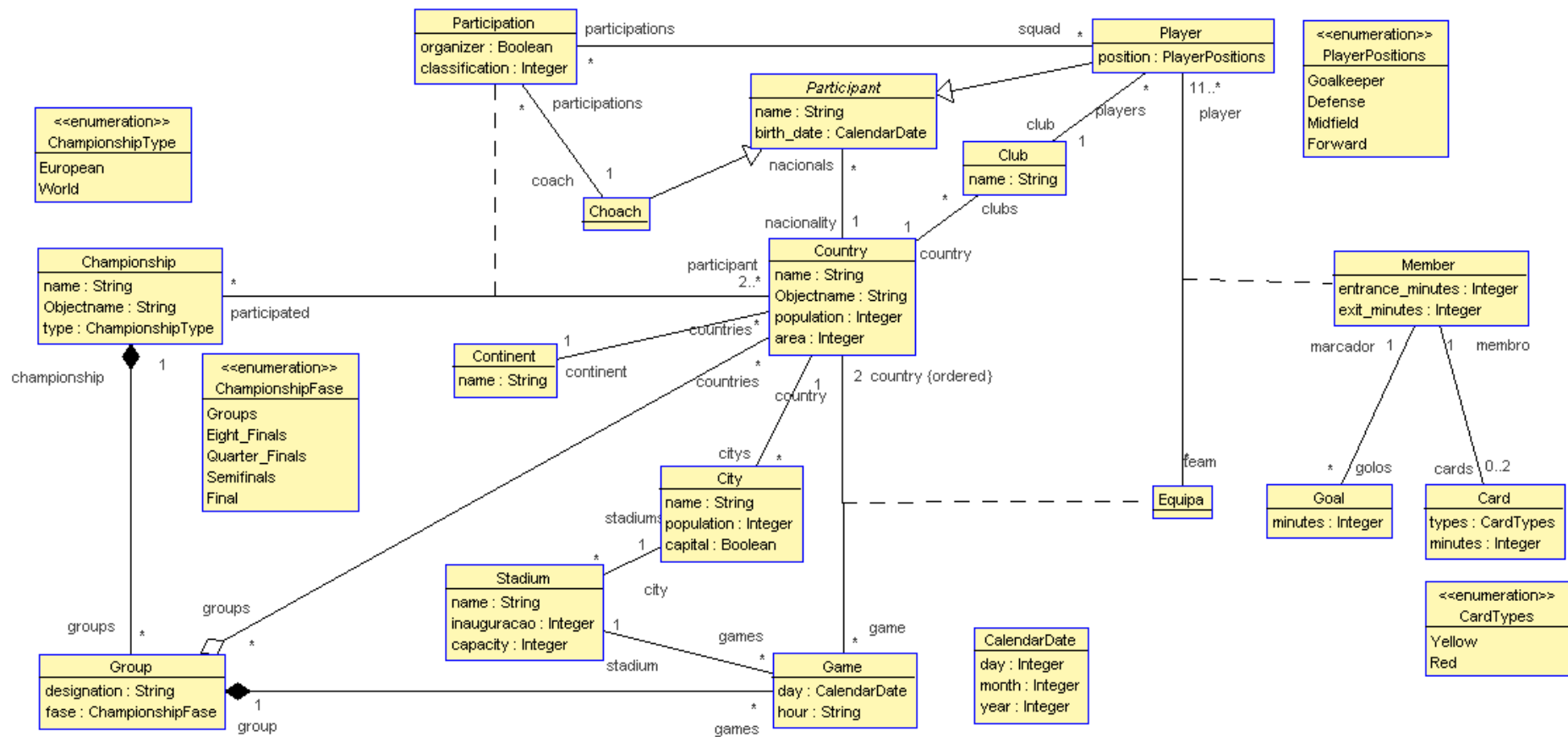Figure 46 – *Royal & Loyal* UML class diagram

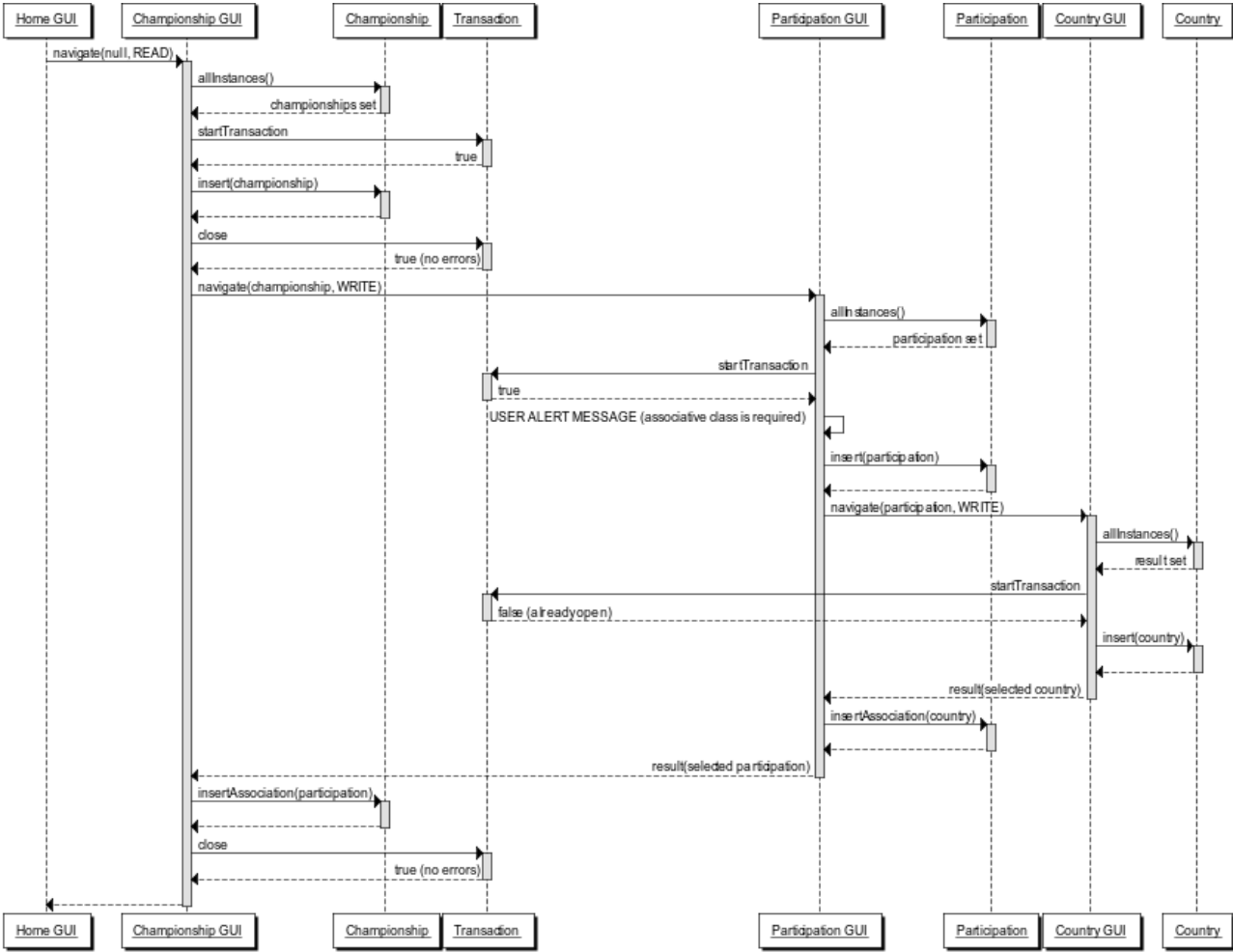Figure 47 – Football Leagues UML class diagram

F. Experiment one – result example

Figure 48 – Experiment one – result example

Luís Silva – July, 2014