Code Generation based on UML Model Transformation

Carlos Eduardo Pereira {carlospereira221@gmail.com},
Fábio Fagundes Lima {fabio.lesi@gmail.com},
João Pedro Granja {joaopedrogranja@gmail.com}

Universidade do Minho, Departamento de Informática
4710-057, Braga, Portugal

Abstract The use of software development methodologies such as model driven engineering is very important to software engineering. This project aims to present a proposal for the transformation of UML models onto a meta model through the creation of a canonical data structure. That canonical structure will allow to make semantic analysis of models and code generation. In this way, the manipulation and reutilization of data in the model can be made in an easy way and extensions to it can be provided independently of UML modelation tools. For the code generation process, Android and IPhone mobile devices were chosen, therefore allowing the installation of applications based on Java and Objective-C respectively.

1 Introduction

Unified Modeling Language (UML)[10] is a modeling language based on Model-driven engineering (MDE)[6] concepts and consists on the application of models to raise the level of abstraction in which developers create software with the objective of making easier to cope development processes with the required standardized methodologies. The application of these techniques improves software quality, reduces the problem and creates a possible solution in the developer perspective.

This higher level of abstraction offered by models also leads to a better reuse of software business logic. In addition to these advantages, it is possible to use tools to transform UML models into other models (meta-models) as well as source code. This will lead to a reduction of software development time, costs and preventing diagram’s data losses due to misinterpretation of the model during code generation.
UML model transformation to either model or code is frequently difficult to achieve. This however, is explained by their complexity, coverage and lack of specification at a semantic level. In this we can particularly point out use case diagrams as an example of the dangers of misinterpretation.

The fact that UML modeling tools did not promote a consensus about the information format contained in the diagrams, normally done by XML metadata interchange (XMI), leads to an almost null interoperability between tools, as well as minor model reutilization if developers have the need to change of modeling tool.

In an attempt to resolve the problem, it was proposed the creation of a meta-model which is basically a canonical data structure to store the data that can be extracted from UML diagrams, specifically class diagrams. The purpose takes into consideration the lack of consensus on the XMI format leading to the creation of a parsing method to handle the data contained on XMI files to be inserted into the canonical structure.

Since it is not practicable to enclose all XMI formats that might exist, the structure was based in a set of Java classes linked between them, developed in a way that makes possible its extension to new techniques of XMI parsing.

The presented solution will simplify the reutilization of data contained by the model, increase the meta-model maintenance, as well as its extensibility and compatibility, once it is based on a well known subset of data in UML diagrams.

For the code generation process we chose as target platforms Android and iPhone mobile devices, since the developed applications use, respectively, Java and Objective-C programming languages, which despite being object oriented languages have very different characteristics.

That selection was also based on the fact that the development of mobile applications is a complex and slow process, where automatic code generation can contribute to improve the process.
2 Transformation approach

The transformation process begins with the XMI generation from UML Models on which no restrictions were placed on the modeling tool to use, as long as it possessed the option to export models in XMI format.

Created from UML modeling, XMI[^1] is a standard created by Object Management Group (OMG) based on XML, which allows quick interchange of metadata for software modeling. Through the years, numerous versions of XMI have been created, from the early 1.0 to the radically different 2.1.

XMI is not a typical standard, as we became to realize the non-existing consensus on the exported formats. In fact, XMI formats do not even have a common structure that could be based upon, leaving no alternative but to analyze, separately, the main modeling tools and its exportation format.

The difference is so obvious, that the sole purpose of validating a XMI file through a schema became a very complicated task, as will be discussed in the next section.

The parsing into the canonical structure is done by a specific process where its entities are designed to support a particular modeling tool exportation format.

Nevertheless, taking into account the possibility of integration of a non supported modeling tool, an option was implemented allowing the developer to create the required capability.

This way, and although the innumerable existing exportation structures of the XMI standard, to a developer it will be enough to create a mapping module between the XMI file and the canonical structure enabling this way the support for the modeling tool.

The canonical structure is mainly, an easy access structure, designed so that its information can be obtained quickly either for reading purposes or processing ones. Thus, it is a global model of all current capabilities of UML models, namely class diagrams, being this way, independent of the modulation tools. This is due to possible the existing differences of interpretation on various tools, namely occurring the case that something might exist in a given tool and be inexistent in another. Nevertheless since the structure is based on
UML models, there is a guarantee that all extracted information will be stored.

Mapping data into the canonical structure is probably one of the most important stages of our process. Occasionally this process may come along with some incoherencies at the semantic level. In order to avoid that, a semantic analysis is also made to the structure, being this process better explained in the next sections.

![Figure 1. Platform Structure](image)

The final phase of the platform constitution, as we can see in Figure 1, we can find the generation modules, namely for Android and iPhone. The construction of these modules has been composed in a way that the final output of the platform could be a package of easy import into development tools to the previously referred devices.

However, once again using the modularity of the platform development language, Java, it is possible that the code generation process can be easily widened to other devices, or even for other programming languages.

This way, we believe that the exploitation of the designed structure can be higher, originating a bigger applicability of code generation capacity from the platform.

In order to exemplify the platform’s transformation process, a case study will be explained throughout the article. This transformation process is based on the UML diagram shown on figure 2, developed in Altova UModel, and merely consists on one class and one interface with the purpose of better exemplifying the different transformation stages.
Figure 2. UML Class Diagram Example

3 XMI Validation

XMI validation by the platform, not only makes possible to realize if XMI is well formed, but also if it has an appropriate structure. This way, it allows us to acknowledge if the platform can recognize the structure in the file or if it will be performed the "best effort technique" in the code generation. So, in order to develop the validation of XMI files, schemas have been chosen, due to the fact that this is the recommended method by W3C [16].

One advantage of schemas is the fact that they validate XMI files that use namespaces[4], through which is possible to determinate the identifiers shape, namely the XMI version and the program that has generated it.

However, as previously referred, one of the biggest difficulties found is related to the fact that there is no standardization for XMI structure. In this way, it becomes practically impossible to create a generic schema that validates any submitted XMI file.

So, in this sense, we chose to have a schema for each program and corresponding version, so that it is possible to give guarantees of a more specific validation, according to the tool used.

In order to widen the options of the developer and not compelling him to the tools supported by the platform, it has been added the possibility to submit a new schema to validate a XMI file for a new tool.

4 XMI Parsing

As stated above, due to the differences on XMI files exported by various applications, a generic structure was created in order to ex-
tract all the possible data from a XMI file. Thus it became apparent
the need for a modular structure capable of supporting the ability
to parse those same files. In this way, the XMI document read is
first stored into a Document class (Java) and then mapped into the
canonical structure.

The mapping process depends on the information retrieved by
the XMI validation process. As such, after the validation is con-
cluded, the information containing the version of the XMI or the
program’s name is transferred to the parsing module, which selects,
as seen on Figure 3, the most appropriate entity to process the map-
ping.

![Parsing Structure](image)

Each one of these entities, possesses its own set of rules for
mapping the XMI file to the generic structure, normally using Node-
Parsing options from Java or even XPath[17].

Therefore, the process mainly consists of recreating the inform-
ation contained in the XMI file into the structure, automatically
linking the existing associations or other relational parts.

5 The Canonical Structure

The Canonical Structure is, in a way, the main foundation of the
platform. Due to the lack of consensus in the various XMI formats,
this structure was created with the purpose of being capable to store
a subset of information that a XMI (originated from a UML model)
file could have, and as stated before serving as a bridge between the parsing module and the generation module.

Therefore, the goal was that no matter the modeling platform on which the XMI file was created, the mapping process would occur. So, for that purpose, a rigorous examination process occurred on numerous UML modeling platforms such as Visual Paradigm, Altova UModel, Enterprise Architect among others, in order that we could apprehend the capabilities of each of them and directly mapping them into the canonical structure.

![Figure 4. Partial Representation of the Canonical Structure](image-url)
This architecture of classes is prepared for future growth if needed and contains all relevant elements existing on class diagrams.

However, despite the structure only being capable, at this moment, of storing effectively class diagrams, its modularity easily would allow the addition of a specific module for activity diagrams or any other UML diagram. In fact, if a parsing module would to be created for the purpose of mapping information retrieved from a sequence diagram, the structure would be capable, even without any additional effort, of storing the given’s diagram most relevant information. However, for the main objective, class diagrams were UML’s most required part.

Choosing Java was due to its high modularity and portability, and the structure was made accordingly to the hierarchical properties of class diagrams as represented on Figure 3, where each UML entity and attributes were mapped into corresponding Java classes. Each entity property was transformed into variables of the corresponding Java class.

However, besides storing all information related to the class diagrams on the XMI file, the structure is also able to save unique identification items, associations, primitive variables, datatypes, comments, hierarchical positions (dependencies) among others, which means that, even if the information is not important to the generation process, it will be stored, enabling future treatment.

This basically means that if besides the loading of a XMI file supported in this platform, it could be added an exporting capability, that is, the capability of recreating a new XMI file. With it, this platform would greatly help improve the interoperability between the tools, basically inexistent as was referred, allowing it to be also a platform for interchange of models between tools.

6 Mobile Code Generation

In this section it will be explained the different stages of the generation process for iPhone and Android. This process is always composed of two stages, the first where the data is analyzed on a semantic level, and the second where accordingly to the device, Android or iPhone, a package is created.
6.1 Semantic Analysis

Although some modelling tools perform some kind of control in the model’s logic and do not allow the users to commit semantic errors at this level, in fact not all tools are able to do it. In this way, we have decided to verify if the model’s rules are correctly defined, once that a badly constructed model will probably originate code with imperfections.

Therefore, a first semantic analysis is made, completely independent of the programming language, where it is verified if there are errors such as: entities that will be mapped into a file (classes, interfaces, packages and others) with the same name and absolute path; attributes inside classes with the same name; variables which are invoked and are not defined; inadequately defined types of data, among others.

Complementary to this analysis it can be also made a specific semantic analysis to each one of the programming language used in mobile devices. All errors and inconsistencies found throughout the analysis will be reported to the end user. So, in this way, changes on the model or code later generated code can be easily and efficiently accomplished.

6.2 Android

Android is Google’s mobile operating system. Contrary to iPhone, is not a device, but a operating system which is taking its first steps.

Based on Linux kernel, Android achieved some of Linux’s features such as security safety or important mobile issues such as memory management or, even, process scheduling. Adding to this, Android also has a good power management, which is a problem on mobile devices.

Perhaps one of its main advantages is being open source, complemented by features such as a SQLite database or even OpenGL support. It uses Java as its native language, however despite using it, Android is not pure Java. Android’s Java does not run on JVM (Java Virtual Machine) but instead on Dalvik. This platform, despite still being a Virtual Machine, runs .dex files instead, which are
previous Java .class files transformed by a special Android Framework tool, named dx.

Android’s framework is very well structured, being mainly divided into five blocks. However, for the purpose of this work, only two of them are worth referring. A brief explanation of both follows:

– Activity -> Represents a single "screen" in the application where each screen, represents a corresponding activity class.
– Service -> A Service is basically the code part that runs without a UI. This allows for concurrent application tasks such as, for example, listening to a song while writing a sms.

For the work’s purpose, several parts can be grouped. As such, the main objective was the creation of an Android Package, capable of automatically be imported into the Eclipse Platform.

The resulting package, shown in Figure 5, includes all Java code retrieved from the UML models, jars needed, as well as the resources files needed by Android applications.

![Android Package Overview](image)

**Figure 5.** Android Package Overview

In the generation process were included resources files such as AndroidManifest, which is an XML file where you declare your ap-
lication’s components and requirements[7]. As such, the Resources folder is created, being capable of storing XML, PNG or JPG files, which normally are auto-generated by Android tools for efficiency purposes. Inside the Resources folder, files such as string.XML needed for several declarations in Activity classes are also included.

On the other hand, all Java code is generated accordingly to Java’s specifications, being put inside the src folder. This happens because, despite all the existing information on the UML models, it is impossible to transform UI model parts into the corresponding Activity classes used on Android. Therefore, all generated code is considered to be simple Java code with no associations to Android’s class types.

As for the Activity part, which represents the simple user interface view, a generic one is created, allowing the future developer to not have to connect the code portions. In this way, the created Generic Activity class, includes already the necessary references to the model based code. Given that, a developer can easily generate Android’s back-end code (through the also generated Generic Service class) using the platform and later creating the interface for it.

Being the source files pure Java code, inherent parts such as constructors or even imports are automatically calculated and thus presented to the developer. As stated in previous sections, semantic issues are also reported by means of a warnings file placed at the final package.

The generated code is done thanks to the canonical structure already referred. As such, Java items such as packages, classes, interfaces, enumerations, methods, attributes, method parameters, comments, datatypes or even primitive ones are all mapped into the final source code.

Figure 6 represents the generated files based on the UML model example referred in the previous sections. The process generated two files, one class and one interface, with the corresponding contents.

6.3 iPhone

The iPhone is a very advanced technological mobile device which allows the load of previously built applications. These applications,
when well developed, may positively respond to market needs in some aspects like the utilization of non-centered applications where various technology inclusion and processing power is required.

Despite its advantages, the iPhone possesses some normal limitations inherent to a mobile device such as limited disk space, memory and battery. Because of those limitations, iPhone developers had to take measures in an attempt to minimize these effects, such as limiting the number of applications concurrently running to only one, disabling exchange of information between applications, forcing the applications to run in a "sandbox" and denying all access to external contents.

On the developer behalf, relies the duty of developing compact applications, not cpu intensive and particularly memory wise in a way that the device can correctly work, fulfilling its users expectations.

Aware of the problems that might emerge, the code generator was designed in a "best-effort" attempt, but always with the main objective of facilitating the iPhone developer’s work through well organized code generation and self-awareness of the device’s limitations.

Figure 6. Java Generated Skeleton Files based on the Class Diagram Example
The generated code is written in an object-oriented language, Objective-C. This language is mainly C with some Smalltalk characteristics created to simply acquire all object-oriented aspects. Aspects such as Objective-C rules, application construction rules, canonical structure contained data or even required files for application construction on iPhone were all taken into account. These files, as seen on Figure 8, such as the main file or even the RootViewController file which is responsible for showing the screen’s main view and all subview creation, are all of them essential for launching applications on the iPhone.

![Figure 7. Overview of auxiliary files and folders on an iPhone Project](image)

Also visible in Figure 8 is the Classes folder where all the code based on the UML models is placed. Thus, to preserve the hierarchic form on the original model, this folder intends to contain all the existent packages serving as a root of the hierarchy. Together with each package, is also created a comment file containing extra information which belongs to the original model.

Inside each package, classes, enumerations and protocols based on the canonical structure are generated.

For every class implementation and for each referenced class on them, imports are created and in an attempt to increase efficiency, the "@class" symbol is quoted on interfaces’ referenced classes. This
way, the load of unnecessary classes in the compilation process is avoided.

In these classes, method signatures are also generated, such as class constructors (init), memory deallocation method (dealloc) or even a fault memory alert method (didReceiveMemoryWarning) where all unnecessary resources would be deallocated, preventing critical faults. Due to Objective-C characteristics, methods’ parameters were created containing two names, one used inside the method and the other used by who calls it.

As a final remark, on each instance variable created, annotations (declared properties) are declared so that the compiler can generate automatically getters and setters.

During this process, and similarly to Android, a comment structure has also been created in order to help future programmers to understand the generated source code.

Also, all generated code intends to easily integrate with xCode. This is due to the fact that xCode is the most used tool in the iPhone field, allowing users to easily develop code. As such, the integration of a resulting code package into xCode will be as simple as importing the package onto it.

Figure 8. Objective-C Generated Skeleton Files based on the Class Diagram Example

Figure 8 represents the generated files based on the UML model example referred in the previous sections. The process generated the class files (implementation and interface) and also the protocol file.
7 Related Work

There are several tools and tasks carried out in the context of the generation of code from UML models, each one with its specific characteristics. Namely the language for which they are going to generate the packet of code, the type of UML supported diagrams and the version of XMI they are able to read.

With Visual Paradigm, its possible to produce Java code from UML diagrams developed in the program itself. Rational Rose offers, yet indirectly, through the usage of Add-ons, the possibility of generate code for different languages.

While the tools, referred previously, try to map directly the attributes and associations into programming language’s variables, a different approach is used in [8]. In this work, mapping is made so it leads the programmer to follow a certain interface generated by the tool, maintaining this way, the skeleton of the model from possible changes made by the programmer.

In [15] is presented a framework that through a XML file, containing formal specification, generates a web-based system for data insertion. The framework adopt the Model-View-Controller (MVC) design pattern.

An approach for transforming UML sequence and class diagrams into Java code is presented in [13]. This approach uses a mapping strategy to convert those diagrams into a meta model for sequence diagram and transformation rules. It is to point out that this approach assumes sequence and class diagrams are syntactically and static-semantically correct.

A pattern-based transformation algorithm, explained in [5], identify patterns in a given collaboration diagram and translate those patterns into Java code. Also in this approach, are assumed that collaboration diagram are syntactically and static-semantically correct.

A Framework for generating code from class diagrams using XML ruled based approach, is presented in [12]. A description of how to translate model data to source code are mapped in a XML file. This file will be used by a rule interpreter to construct a generation tree.
based on structural elements and primitive generators corresponding to primitive elements.

Other work performed in the context of this subject can be found in \[9\], \[1\], \[14\] and \[3\].

However, we point out that, after a careful research, we haven’t found any tool that generates a packet of code for any type of mobile device.

8 Conclusion

Along this article we explored the wide amount of information present in the UML models. It was also addressed the lack of consensus about the data export format standard which leads to difficulties when working with it. The solution presented reduces these problems because the proposed model format is a subset of all the others allowing mappings between them. In this way the model can be reused and extended, taking advantage of its modular structure, and even enable some degree of compatibility between applications.

Even if a new standard arises the solution keeps making sense in the way it simplifies some information from the UML models which is not necessary to code generation.

Concerning code generation it was possible to notice that the data from the canonical structure reached out every important aspect of the Java and Objective-C languages and that frequently the code generated was visibly different in each of them. The generated code follows every rule from both languages and will easy the software development process for mobile devices.

Left for future work was the analysis of the exported XMI by other tools, the inclusion in the canonical structure of the information from other types of UML diagrams (sequence diagrams, activity diagrams among others) and the generation of code for others mobile devices. Another future objective is to enhance the tool in order to be able to export XMI files from the canonical structure to increase the compatibility between modelation tools.
References

5. Gregor Engels, Roland Hucking, Stefan Sauer, and Annika Wagner, Uml collaboration diagrams and their transformation to java, Tech. report, University of Paderborn, Dept. of Computer Science, D 33095 Paderborn, Germany.
8. William Harrison, Charles Barton, and Mukung Raghavachari, Mapping uml designs to java, Tech. report, IBM T.J. Watson Research Center.
12. Dong Hyuk Park and Soo Dong Kim, Xml rule based source code generator for uml case tool, Tech. report, Department of Computer Science, Soongsil University.
15. Volker Turau, A framework for automatic generation of web-based data entry applications based on xml, Tech. report, University of Applied Sciences, FH Wiesbaden, Department of Computer Science, Kurt Schumacher Ring 18, 65197 Wiesbaden, Germany.
17. ______, Xml path language (xpath), http://www.w3.org/TR/xpath