MapIt: A model based pattern recovery tool*

Rui Couto[1,2], António Nestor Ribeiro[1,2], and José Creissac Campos[1,2]

[1] Departamento de Informática/Universidade do Minho, Braga, Portugal,
{rui.couto,anr,jose.campos}@di.uminho.pt
[2] HASLab/INESC TEC, Portugal

Abstract. Design patterns provide a means to reuse proven solutions during development, but also to identify good practices during analysis. These are particularly relevant in complex and critical software, such as is the case of ubiquitous and pervasive systems. Model Driven Engineering (MDE) presents a solution for this problem, with the usage of high level models. As part of an effort to develop approaches to the migration of applications to mobile contexts, this paper reports on a tool that identifies design patterns in source code. Code is transformed into both platform specific and independent models, and from these design patterns are inferred. MapIt, the tool which implements these functionalities is described.

Keywords: Design Patterns; MDA; Mobile; Reverse Engineering; Ubiquitous.

1 Introduction

The amount of embedded software that surrounds us means that the pervasiveness of software is increasingly growing in our daily lives. From automobiles with infotainment capabilities, to fly-by-wire systems or the intelligent monitoring and control of electrical substation automation systems, embedded systems' complexity is growing and raising issues of design, development and maintainability.

The Model Driven Architecture (MDA) methodology has been proposed by the Object Management Group (OMG) to overcome the above issues in the broader area of software engineering [13,16]. In the software development process, paying more attention to the modeling phase has shown multiple benefits. Usually, in traditional approaches, the time spent writing models is considered

* This work is funded by ERDF - European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-015095.

Published in Model-Based Methodologies for Pervasive and Embedded Software, volume 7706 of Lecture Notes in Computer Science, pages 19-37. Springer. 2013. The final publication is available at link.springer.com.

somehow wasted, and this happens because only code writing is considered software production. The MDA defines a standard approach to develop model based software solutions. It turns the time "spent" writing models into software production, allowing the generation of code from those models. This approach's main objective is to put more relevance in the modeling phase, and even, to make the modeling process the main activity in the production of software solutions. It is relevant to note that the MDA's relevance and feasibility has already been proven by some authors [13,16]. However, to achieve such benefits models must be complete and correct. This results in a time consuming task and usually, once written, a model is never updated again. This lack of attention to the models makes them obsolete and therefore useless [15]. So, the full advantages of the modeling process are not attained.

One of the contributing factors to this mismatch is the fact that the OMG does not define the reverse process from code to models (it only defines the models to code process). This prevents the establishment of a fully round-trip engineering approach, making it difficult to keep the models coherent with the code. Another aspect in which a reverse approach might be interesting is in the integration of projects not developed according to an MDA approach into an MDA based modeling process. To support this, tools are needed to generate models based on their source code.

In a forward engineering approach we start with abstract models and refine them into code. Our aim is to be able to reverse engineer code into increasingly abstract models. Design patterns are of crucial importance in this process. They allow us to abstract, from the code, the architectural solutions used in the development process. Based on them, one can view source code, not as a complex set of classes, but as a structured solution to a given problem, where it is possible to instantiate each class with a given role as defined by the patterns.

Christopher Alexander [2] described a design pattern as the core of a solution to a given recurrent problem. Thus, they may be used to solve a problem multiple times, without repetition. Although the original definition was relative to buildings, it is also considered to be valid for software engineering. A design pattern describes a simple and elegant solution to a well known problem. Current design patterns' relevance is due to the work of Erich Gamma, who cataloged a collection of design patterns in [7]. This catalog may be considered as a starting point. Since then, many other authors have cataloged different patterns [3,6].

Reversing the MDA process will allows us to easily create, evolve and migrate software using a model based approach. Having reversed models, Erich Gamma patterns will be considered and inferred on these models to better capture the design of the implemented system [7]. Hence, this work main objectives may be resumed as follows:

- to analyze and extract information from Java source code;
- to generate high level models (cf. Platform Specific Models (PSM)) in the Unified Modeling Language (UML) framework;
- to infer design patterns on those models;
- and to abstract PSM into Platform Independent Models (PIM).

To achieve them, a tool which implements these functionalities was developed. The tool fulfills the above objectives, being able to generate models from code, transform models between different abstraction levels, and identify architectural patterns. We have started with Erich Gamma patterns as we wanted to develop an approach that could be as generic as possible, but other pattern collections can later be considered.

The remaining of the paper is organized as follows. Section 2 presents some work related with our approach. Section 3 presents the expected challenges. In section 4 we present and describe our tool. A case study will be presented in section 5, and section 6 presents some results' discussion. Section 7 concludes this paper addressing some hints on future work.

2 Background

This section covers relevant background information regarding the reverse engineering process of extracting and interpreting models from a system's implementation. It presents the most relevant MDA concepts concerning this work. Both direct and reverse MDA processes are addressed, as well as the pattern inference process. Also, tools which implement such concepts are described.

2.1 Models and transformations

Models' definition and transformation is the essence of the MDA process. To have these models, we need a standard way to define them. This is achieved through a metamodel. The metamodel is a "well defined language" which allows us to create both PIM and PSM. A PIM is a model which describes a software system at a higher abstraction level. It is independent from the technology or platform where the system will be deployed. These models are usually transformed into PSMs. A PSM is a model at a lower abstraction level than a PIM. It describes software systems for a specific platform or technology [18]. Usually, PSM models are transformed into source code. The standard modeling language adopted to create these models is the UML, but that is not a restriction. Having these models, the transformations are the next step.

A transformation definition is a set of transformation rules. The information about how a model element (a PIM element, for instance) should be represented on other model (PSM, for instance) consists in a transformation rule. Having transformation definitions, it is then possible to transform one model into another at a different abstraction level [5].

Transformation rules, as defined by the MDA, typically have the intent of lowering the abstraction level of the model. That is, bringing it closer to a working system [14]. These rules may have a reverse mode, which allows a reverse transformation and raise a model's abstraction level. Typical transformations are from abstract to concrete but by reversing these rules, we may expect reverse transformations.

2.2 Reversing the MDA process

Several MDA implementations have already been proposed, resulting in tools and algorithms to go from models to code [13,11]. Reversing the MDA process is very useful as a way not only to include existent software in the MDA process, but also to keep consistency between source code and models. However, the OMG does not define the reverse process from code to models, it only defines the models to code process. The reverse MDA process has also been subject of study, but usually in a limited way. A common approach to include legacy software systems in a new project, is by treating them, for instance, as a class in the new system's models, whose methods are the functionalities provided by this software. A complete integration of these systems would be more useful, and some authors have also studied this issue.

The most common methodologies to reverse the MDA process suggest that:

- First, the source code must be analyzed (be it text or bytecode);
- Second, relevant information must be extracted to create an intermediary representation (as a syntactic tree or a graph, for instance) [19];
- Finally, as these representations contain a high level of detail, they must be simplified [17,4] by abstracting away unnecessary details.

This approach allows not only to integrate the models in the MDA process, but also to execute high level operations in the models, such as performing software changes, high level code analysis, pattern inference and formal verification.

To execute the reverse MDA process there are two possible approaches. The first one consists in a static analysis which mainly extracts the structural aspects of the code, producing PSM and PIM. Our tool is based in this first approach. The second approach consists on a dynamic analysis which is focused on software behavioral aspects [1].

Performing the basic structural analysis is a relatively simple task, because it relies only in the (textual) source code. On the other hand, inferring the relation between the model elements is the hardest task (apart from the dynamic information) [10]. Some studies stated that while binary associations can be directly extracted from source code, $n - ary$ associations requires more work to be inferred. The static analysis process allows us to create (possible incomplete) UML class diagrams. To illustrate this process' difficulty, we note that one of the most precise recovering tools (according to the author), is capable of extracting only 62% of UML elements [8].

The reverse engineering process can be applied at any level of abstraction. However, considering that we are interested in reversing existing systems, there are two possible main starting points. One is starting from the text of the source code, as programmed by the developer. The other, is starting from the compiled code (in the case of Java, from the bytecode). The textual approach has shown to be more adequate in this context. First, the source represents the system without compiler optimizations. Second, using Java bytecode requires a bigger effort to understand and extract information from it. Also, by using the textual source code, it is possible to more easily integrate this tool in the development process.

2.3 Pattern Inference

The importance of design patterns is proved by the number of patterns found on software developed nowadays. Some of the advantages they offer are to allow a vision of the system at a higher abstraction level, or to be used as quality (or lack of quality) measures. When patterns are not documented in a software, an inference process is needed to retrieve them [8].

The pattern inference functionality offers multiple advantages. It can be used as a quality measure, to obtain extra information, among other possibilities. Also, it is useful during a project maintenance phase, offering higher level analysis, making it easier to understand the systems. The pattern inference capability is our tool's second proposed component. After reversing source code into PSM and PIM, the tool should be able to infer patterns based on these models.

Existing studies on pattern inference provide some guidelines about how to implement this functionality [13,16,11]. The suggested approach propose, that a given software should be analyzed and mapped onto an adequate, previously defined, metamodel [20]. This metamodel should contain static and structural information, along with some dynamic information (such as method invocation). From this representation, a knowledge base (of facts and rules representing the system information) will be extracted to an external format. These facts will be analyzed searching for design patterns. A set of rules representing design patterns should be defined beforehand, to allow pattern searching.

There are several proposed intermediary data representations, resulting from different approaches. These approaches may be organized in multiple categories, being them: graphs which preserves the elements hierarchy, matrix, syntactic form and programming language (such as Prolog). Having this intermediary representation is then possible to start the inference pattern. This process consists in the comparison of the intermediary representation, against previously defined patterns, in the same language.

2.4 Available tools

A number of "round-trip" and "reverse engineering" MDA tools were analyzed. Some of them offer Integrated Development Environment (IDE) capabilities, allowing some degree of code and model manipulation during software development (such as ArgoUML and Fujaba [12]). Other tools are focused on a more efficient analysis, offering a higher level of details on the models (like Ptidej [8]). Still other tools combine both functionalities (for instance Together and Visual Paradigm). The Ptidej tool claims to be the most precise tool, being able to recover the highest level of UML elements [8].

Regarding pattern identification, the most relevant tools are Reclipse [19] and Ptidej, which are able to infer patterns on UML models. Also, only Fujaba and Ptidej are able to perform high level operations on the models.

ArgoUML offers high level diagrams based on source code, as well as some IDE functionalities, such as a code editor, a compiler and a debugger. However,

this tools' diagrams are very simple, and it provides no patterns inference. Reclipse is also a plugin for an IDE, so it offers the common functionalities (code editor, compiler, debugger, deployment, etc.), it provides also pattern inference functionalities, and is able to display UML models. However, the pattern inference is not easy to use, and it is also not very accurate, since there are some problems with the software analysis, shown by the incomplete UML diagrams. Like ArgoUML, jGrasp offers some IDE functionalities, and high level diagrams. However, its diagrams are even more incomplete than ArgoUML, and the code editor is not as complete as those provided by others IDEs (such as NetBeans or Eclipse). Ptidej focus in model analysis and pattern inference. However, it offers neither IDE features, nor model transformations functionalities.

In conclusion, we have found that none of these tools is able to fully cover the range of functionalities desired for our proposes. Also, using these tools separately may not be convenient since they receive the input files in distinct formats (some in class files, other in text files, and others in Java archives).

3 Requirements for a pattern recovery tool

As seen in the previous sections, no tool is able to conjugate all the proposed functionalities in a single development environment. The tools closer to such objective are Ptidej, Fujaba and jGrasp, however they are an "incomplete" solution. By developing MapIt, our pattern recovery tool, we aim to provide an alternative for these tools, by merging the proposed functionalities in a single environment.

The topics discussed above can be reduced to three main functionalities that the tool implements (see Figure 1). They are resumed as: the reverse MDA process from source code into PSM, the design patterns inference on a PSM, and the reverse MDA process from PSM to PIM.

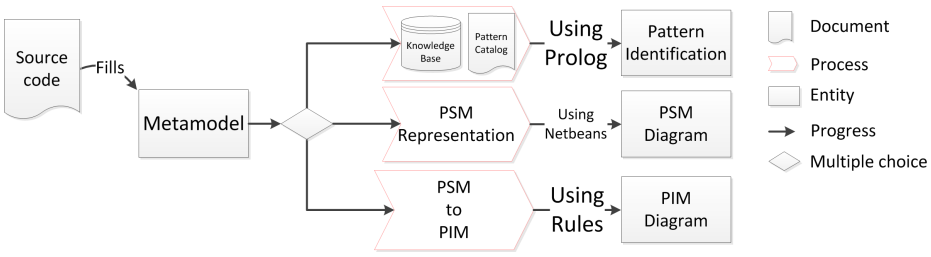

Fig. 1. Representation of the tool functionalities.

The tool's functionalities operate independently from each others. The developer may chose to use one, two or all of the functionalities in the desired order, accordingly to his/her needs. The functionalities are meant to be used separately, and at the same time complement each other.

3.1 Source code Metamodel

This functionality corresponds to the source code to class diagram (PSM) abstraction, represented in Figure 1 on the left. The process to do such should start with the structural analysis of the project's source code, analyzing then each of the Java files. To achieve this, semantic analysis is used on the source code, using a Java parser. The extracted information should be mapped onto an intermediary representation. An adequate way to represent it is by a metamodel, which is complete, accurate and at the same time not overloaded with useless information. But, even if not all of the software information is needed for the proposed objectives, the metamodel is ready for future functionalities. The source code to metamodel process is common to all of the three functionalities provided by the tool, since all of them occur over the inferred metamodel.

3.2 Source code to PSM

The second functionality consists in achieving a PSM, departing from the source code. In practice, this process will start from the metamodel, taking advantage of the functionality presented above. This functionality is represented in Figure 1, in the middle.

Since the metamodel contains some irrelevant information when considering a PSM, its elements must be processed in order to select the relevant information. To implement this functionality, a graphical property was assigned to each metamodel element. The metamodel elements must be then visually represented according to the UML notation. An effort was made to make this diagram interactive, allowing the user to visually rearrange the model elements.

In conclusion, a relation between metamodel elements, UML entities, and a visual representation must be defined. This will allow us to define how each Java element will be represented and displayed to the user.

3.3 PSM pattern inference

Pattern inference on models allows us to have a high level information representation. This process, represented in Figure 1, on the top right, occurs over previously reversed information. This information is also used for other results. In this case, to generate the PIM (see below). Not only was the inference process specified, but also how to customize it. Pattern inference is then parameterized and easily extensible.

The pattern inference process starts with the knowledge base creation. This process depends upon two major factors. First, it depends upon correctly defining the patterns used in the inference process. Second, it depends upon achieving an appropriate knowledge base, correctly representing all the system's information. Creating the knowledge base is a critical step, since overloading the knowledge base with information will slow down the inference process, while having too little information will hinder the pattern recognition capabilities of the tool.

It is easy to understand that the intermediary representation, the knowledge base, and the patterns are somehow related. This relation is the possibility to achieve transformations between them, in the presented order. To start, a knowledge base, based on source code is created. After an analysis of possible alternatives, it was decided that a Prolog interpreter could handle the knowledge base. So, a tool which implements information mapping and exchange with that external tool was developed.

Good results on this process are only expectable if there is a standard and well defined way to represent a design pattern. Additionally, this representation will make it possible to interpret an external pattern definition, and use it on the analysis process. These representations are then used on the external tool, which should contain the knowledge base. This external definition of patterns is called the "pattern catalog". It consists of a set of user defined patterns. A customized set of rules may be created and used to identify these patterns on a model. A set of rules based on Erich Gamma book was developed for test purposes. The catalog implementation is done in an external user defined file.

The patterns visual representation is similar to the PSM representation. Its major distinctive characteristic is that of highlighting patterns on the representation.

3.4 PSM to PIM

The model abstraction process is usually related with information simplification or reduction. This process mainly consists in reducing the elements' information, quantity, or even change their information. As result, a PSM is transformed in a more generic model, the PIM, represented in Figure 1, on the bottom right.

The abstraction process could be treated as model filtering or as model transformation. Also, this process produces a model, and as such, a visual representation is needed. Considering it as a transformation process, it is somehow based on the MDA Explained book [11] approach. That book presents a set of transformation rules, in the forward direction. Their reverse form allows a fully automated reverse transformation. For each metamodel element, a transformation rule was then defined, and that rule transforms that element considering the whole of the application context.

4 The MapIt tool

To achieve source code into PSM transformations, the mapping process was completely defined. To start, a parser will analyze the source code, and preserve the extracted information in a metamodel. That metamodel has a representation for each Java element, as well as the information needed to preserve their hierarchy.

The used metamodel has a mapping for each Java attribute and method, which are part of a Java element. Java Classes and Interfaces have a respective mapping as well. The notion of Java element was created to represent the Class

and Interface abstractions. All these elements are assembled in a Java package representation.

Only two different relationship cardinalities were considered for simplification purposes, since it is widely recognized that their inference is difficult [9]. These difficulties are more evident when performing static analysis, since many times the collections are "raw", meaning that they don't specify their type. Since their type is unknown until software execution, these collections are therefore impossible to infer. One of the considered relationships was the association, understood as a "one-to-one" relation (1..1). This relationship is present when a class contains a reference to another class or interface. The other relationship was the aggregation, understood as a "one-to-many" relation (1..$*$). In this case, we represent the property of a class containing a reference to a set/list of classes (or interfaces). Also, we enrich our metamodel by adding method invocation information, from one class to another class (or interface). This provides us with more information not only for PSM and PIM, but also for the pattern inference process.

As static analysis results, a set of Prolog facts are produced creating the knowledge base, by parsing all the elements and filtering them. These facts are used on the pattern inference process.

Incomplete patterns inference may also be achieved, resorting to Prolog anonymous variables. To do such, for a given pattern, each of its components may be changed by an anonymous variable, in its representation. Making all the permutations among all the pattern elements, all possible incomplete patterns will be found.

4.1 PSM to PIM abstraction

To achieve PIM from PSM, a set of transformation rules is needed. Also, PIM and PSM must both be completely specified to allow transformations from PIM to PSM. To achieve this, they are both modeled in UML. In short, it is possible to say that the PIM models derive from the PSM, by removing the Java platform specific elements from the PSM.

As described in [11], models transformation is achieved by applying well defined rules on model elements. These rules describe how elements from one model are mapped into another model, at a different abstraction level. That book presents a set of rules to achieve PIM into PSM transformations. The adopted approach was to define the book rules on the reverse form. The transformation process is then defined as the process of removing or filtering elements for a given model. All of the PSM elements will be processed and transformed into higher level elements, resulting in a PIM model. This PIM still contains some platform specific details, since it is obtained from a PSM.

The used transformation rules are encoded in Java methods, and for each element a different rule exists. Each method (for each PSM element) receives as argument not only the information about the element being processed, but also the information about all its upper elements in the hierarchy. This is the only way to achieve model-wide transformations.

4.2 PSM pattern inference

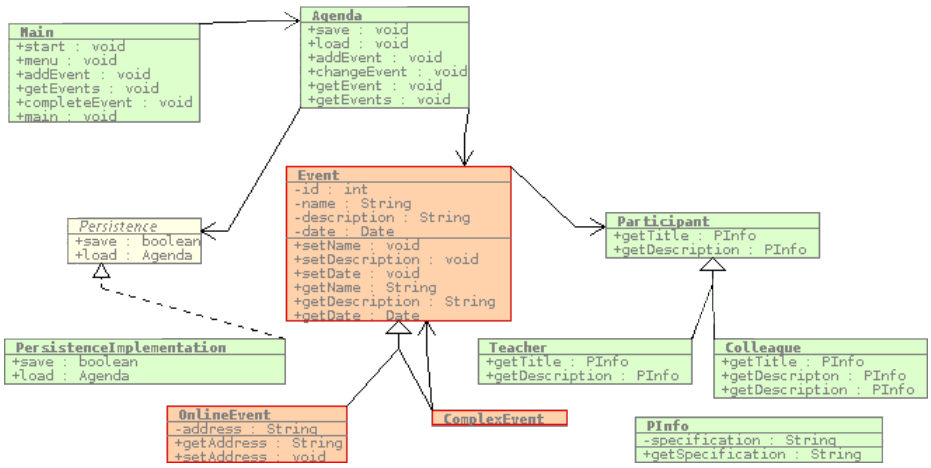

Fig. 2. Example of identified pattern in a PSM diagram (produced by MapIt).

In the pattern inference process, the PSM is analyzed, and the relevant structural information is preserved. This information is preserved in an internal representation, which is the basis for the pattern inference process. This representation is handled by a Prolog interpreter, which creates a knowledge base from the information therein.

Here, the solution was to include the Prolog technology to handle these representations. Thus, the Prolog knowledge base keeps all the software facts, extracted from an intermediary PSM representation. There are a set of Prolog facts that are generated from the software representation. A class existence information is expressed in a `class/1` fact. For an interface, the same approach is taken with `interface/1`. Aggregation, composition or association information is represented as `contains/2`, between two classes or a class and an interface. Heritage relationships are represented with the `extends/2` fact. Implementation properties are also considered, as `implements/2`. The additional invocation information is represented with the fact `calls/3`, between two classes and one method.

Once the Prolog knowledge base is populated, it can be questioned for patterns. As soon as the pattern catalog is parsed, this module will interact with the Prolog tool to search matching patterns with the provided catalog rules. It is possible to conclude that this may be considered a property satisfaction problem, for a selected rule, on a given knowledge base. Figure 2 presents an example of a pattern identified in a PSM. In this figure the composite pattern has been inferred, it can be seen that `OnlineEvent` and `ComplexEvent` extend `Event`,

and `ComplexEvent` contains one-or-more `Events`, creating then the composite pattern among these classes.

Different precision levels on rules definitions will result in distinct results. While a lenient rule will find more patterns, their precision level will be lower. A more strict rule will find less, but more precise, patterns (i.e. it will identify less "false" patterns instances).

4.3 Implementation

The parsing process is composed by two phases. First the information is extracted as represented from the source code, converted in metamodel elements. Only then is it possible to establish the relations between the elements. The parser extracted information is then mapped into an adequate metamodel, since metamodels are recognized to be the best information representation for pattern inference [8]. The Prolog integration uses the GNUProlog interpreter and engine to handle the knowledge base. The two previously presented functionalities are abstracted by a module which handles interaction with the interpreter, and loads also the pattern catalog.

The referred pattern catalog is a textual file, containing a set of Prolog rules (with some restrictions). On each line, a rule should be defined as follows: `pattern_name/arity#(prolog_rule)`, were `pattern_name` is the design pattern name, `arity` is the rule's number of arguments and `prolog_rule` is the Prolog rule, representing a design patten. A rule example (for the Composite pattern) is depicted in the Figure 3.

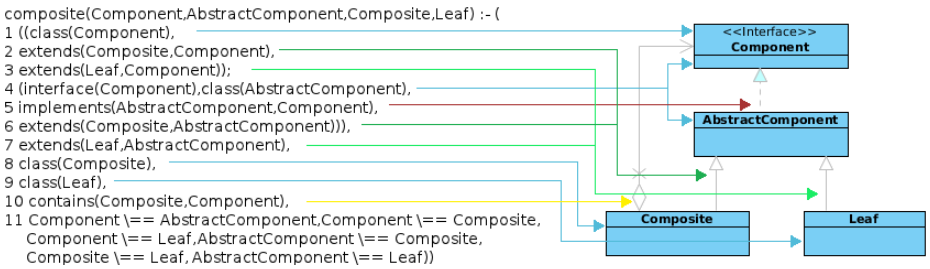

Fig. 3. Prolog rule representation for the Composite pattern.

A NetBeans plugin, depicted in Figure 4, provides the final user interface, enabling access to the presented functionalities. With this approach the use of the tool becomes integrated into the software development environment. Also, the user may take advantage of the IDE functionalities while using this tool. We made an effort to make all these module's functionalities as generic and reusable as possible. This will allow us to easily improve or change it. This was mainly achieved due to the metamodel capability to express both PIM and PSM, so only one metamodel representation was made.

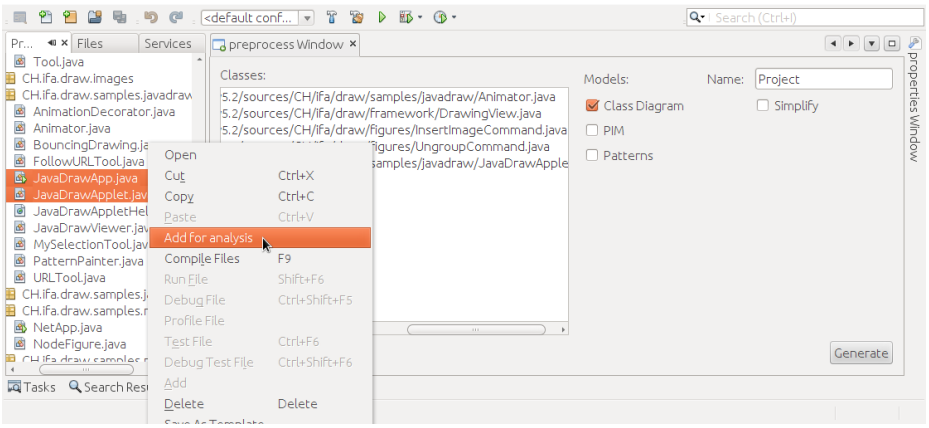

Fig. 4. The MapIt plugin.

The last implemented functionality was the automation of models transformation. It required a set of transformation rules to be fully specified (which may be easily extended). These rules were implemented as follows. A Java class is mapped onto a PIM class. Also, a Java association is mapped on a PIM association. For a Java public attribute, a PIM public attribute is created. A Java method is transformed onto a PIM operation, in an adequate way. A private Java attribute which contains the correspondent getter and setter, is transformed into a public PIM attribute. Also, for each element, properties such data type should be removed or adjusted. Absent PIM elements such as Java interfaces or other project files (Extensible Markup Language (XML) files, for instance) will not exist in the PIM.

Resorting to the presented approach, a fully working tool was achieved. In several tests it was possible to generate models with success for multiple kinds of software. Also, in some of these models design patterns were inferred (some known patterns, and other developed patterns). Finally the achieved PSM successfully provided a higher abstraction level representation of the analyzed systems.

5 Case Study

Embedded and pervasive software ranges from autonomous monitoring systems to highly graphical and interactive systems such as automobile infotainment systems. We are particularly interested in this kind of systems and the challenges they present. As an example of a graphical and interactive application with reasonable complexity we decided to analyze JHotDraw.

JHotDraw is an image manipulation software, that allows composing simple shapes (like squares, circles, text, etc.). A number of factors lead to the choice of this particular tool. First, since it is open-source software, the source code is easy

to obtain. Second, this software has an adequate size, with about 160 classes and 9000 code lines, which makes it a good case study. Also, this software has the property of being developed by a team which includes Erich Gamma, resulting in a design pattern based software. This software allowed us to test the tool's behavior in a reasonably complex environment. Also, the software was used to test the analyzed tools, allowing a comparison of the achieved results.

Our plugin was installed in NetBeans 6.9, and the software was imported into our IDE. That is to say that the conclusions were obtained with all the tools in the same conditions. The analyzed tools are the ones considered more relevant in this area, consisting in ArgoUML, jGrasp, Ptidej and Relipse (or Fujaba). At the moment, ArgoUML is the most outdated tool, updated in December 2011 (less than a year ago, considering the time of writing). On the other hand, jGrasp was updated in September 2012. Some of these tools were released some years ago, but they are being constantly updated. These frequent updates are somehow indications about the tools' importance and need.

After importing the software with each of the tools, each one of the three MapIt main functionalities were tested. With this approach it is possible to compare all of the tools. The MapIt tool successfully applied each of the functionalities to the source code, producing the three expected outputs, that is: the PSM diagrams, the patterns inference and the PIM diagrams.

When testing the MapIt tool, both source code to PSM and PSM to PIM functionalities produced the expected results, allowing us to obtain high level diagrams based on the source code. These diagrams contain the Java elements and their relations, as represented in Figure 2. Also, both functionalities are fully automated as proposed, with no software size restrictions. The code to PSM functionality produced an UML class diagram as expected, and all the expected UML elements and their relations were represented, based on the metamodel instance. Also, this process did not require user interaction. As expected, the JHotDraw software analysis resulted in a large number of elements being shown in the PSM diagram. As the representation is interactive, it allows for the adjustment of the elements positions. This functionality eases the analysis of large diagrams, meaning there is no restriction on the maximum size of the analyzed software. Another functionality developed to ease the analysis process supports the simplification of diagrams, by representing the metamodels elements only by their name. The PSM to PIM module produced similar results, producing a higher lever UML class diagram. This model is obtained by applying previous defined rules to the PSM elements. Here again, the software dimension is not a restriction to the use of this functionality, and the produced results are similar.

When using the pattern inference functionality, the user must select whether to use the embedded pattern catalog or, otherwise, select an external one. All the inferred patterns are identified and listed to the user. The higher the number of patterns in a software, the more useful this functionality becomes. Some analyzed tools, like Fujaba, present all the patterns at once, and that makes it hard to understand the patterns arrangement. The achieved results proved the viability to include the pattern inference functionality on a model analysis tool. Also,

this shows how it is possible to include an external technology (Prolog) in this process, taking advantage of that language's capabilities.

6 Discussion

After testing the MapIt and the selected tools with the described approach it was possible to identify some topics for discussion. Comparing the resulting tool with the proposed objectives some considerations can be made. First, the proposed functional objectives were generally achieved, and the viability of the approach is considered as proven. Then, other non-functional objectives such as usability improvement (in comparison to other tools), or facilitating the tool's installation and usage were also addressed by resorting to Netbeans. This IDE is a widely known tool, so installing and using the plugin will be familiar to developers.

As presented here, integrating the proposed functionalities in one single tool was an overall achieved objective. All the three previously proposed functionalities were implemented, and are available in a single tool. The modules which handle the Prolog and parsing functionalities where implemented in a convenient way, allowing to easily change them in further work.

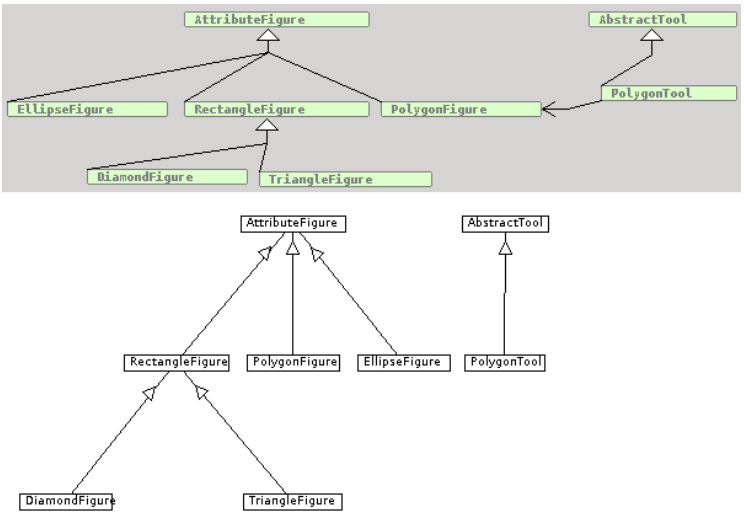

Fig. 5. MapIt diagram (top) and Ptidej diagram (bottom) for the same classes subset.

When comparing the obtained results against the other tools' results, we achieved some interesting conclusions. Comparing to the detail of the obtained models, the other tools' models were generally less elaborated. Even if all the tools (apart from Reclipse/Fujaba) were able to recognize all the Java elements (classes and interfaces), many of them were not able to correctly recognize all

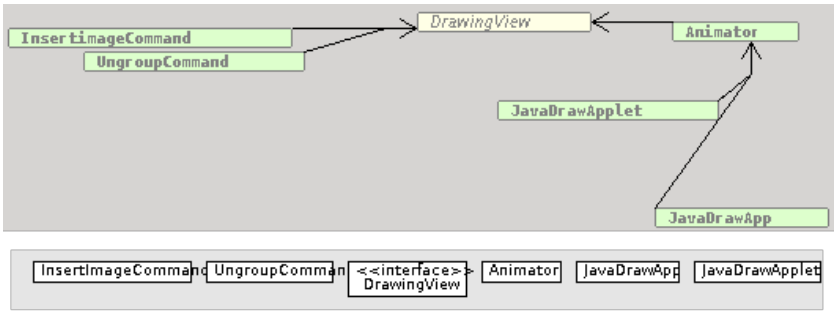

Fig. 6. Another example of MapIt diagram (top) and Ptidej diagram (bottom) for the same classes subset.

their relations. Specifically ArgoUML, jGrasp and Ptidej, failed to recognize some of the relationships. The Reclipse/Fujaba tool was only able to represent the inheritance relationship, and the Ptidej tool missed some other relations. Once again it was possible to conclude that collection inference is hard to achieve, since all the tools showed difficulties when doing it. Also, regarding relations between models elements', ArgoUML and Reclipse were not able to recognize typed collections. In the presented tool, none of these problems are present, so the results concerning models' elements and their relations are considered satisfactory. Figure 5 compare a subset of classes of JHotDraw, in both MapIt (top) and Ptidej (bottom). It is possible to identify that even if the hierarchical relations are present, a relation is missing between `PoligonTool` and `PoligonFigure` in the Ptidej diagram. Another example is shown in Figure 6, where the Ptidej diagram shows no relationships. These are examples of several missing relationships, showing that, in this case, our tool achieves better results than Ptidej when creating PSM diagrams. Moreover, selecting the classes for analysis in Ptidej is not as straightforward as in MapIt, since it requires a jar file or a set of class files, while MapIt requires only to select the files in the IDE, as shown in Figure 4.

The quality of the elements in the model varies depending on the tool. Only two tools achieved satisfactory results, producing model elements in UML notation, being them Reclipse/Fujaba and Ptidej. It was possible to conclude that Ptidej achieved the best results. Even if Fujaba/Reclipse represented detailed diagrams, some information (about relations) was missing. Ptidej tool achieved the best results, however the produced diagrams are static (being then impossible to rearrange the elements on the screen). Only a few analyzed tools were able to infer patterns in the diagrams, specifically Reclipse/Fujaba and Ptidej.

The pattern inference (and representation) process occurred based upon the obtained models. The two tools allowed the use of external catalogs to define the patterns to infer. However, these tools used Java representations (or other specific formats) to represent the patterns. In the presented tool a more flexible format is proposed (using Prolog rules). We were not able to test the pattern

inference functionality in the other tools. In some of them the functionalities does not provide any output, in others it was not clear how to use it. Our tool was able to infer patterns within the JHotDraw code.

None of the analyzed tools has the model abstraction functionality, so it is not possible to compare it. However, regarding the examples presented in the MDA Explained book [11], it is possible to conclude that the obtained models have a high abstraction level, close to a PIM (as expected).

Finally, one last issue regarding improvements to the tool should be addressed. The pattern inference module may be improved expanding the pattern catalog, simply by adding new rules. The model abstraction module can handle different implementations for the transformation process. However, it requires these transformations to be written at the code level. Changing the tool supported language may have one of two consequences: changing the parsing module, or, changing the parsing module and the metamodel (if the new language does not match the metamodel).

Since we used the same software to test all of the tools, this allowed us to achieve some conclusions. As stated before, all the analyzed tools have some points where they may be improved. Also, in some cases (as the relationship inference, or the UML diagrams representation), our tool achieved better results by providing more accurate or detailed models.

Our tool takes also advantage of the integration of the three different functionalities in a single environment. Ptidej offers UML diagrams and pattern inference, however lacks code editing functionalities. ArgoUML and jGrasp includes code editing functionalities and UML diagrams, but no pattern inference. Reclipse includes code editing functionalities, UML diagrams and pattern inference, however the achieved models were very incomplete. Also, none of the tools contains PSM to PIM abstraction functionalities. We developed the MapIt as a tool which aims to cover all the developers needs in a single tool by combining all the needed functionalities, and at the same time in a single development environment.

7 Conclusions and Future Work

In this paper the reverse MDA process and pattern inference were approached. Three distinct model functionalities were analyzed, detailed and implemented in form of the presented tool. The first functionality was the generation of (UML) diagrams from source code. The second one, was the inference of design patterns from the generated models. The third and last functionality was the PSM into PIM model abstraction. The PIM is a model which has no relation with the target platform details. However, the PIM obtained with this tool are not exact PIM in the sense that they still contain some degree of platform details. However, those models' objective remains the same - to raise the abstraction level.

Like in other software areas, ubiquitous and pervasive systems have evolved in many ways. They are present in our lives more than ever, and offering increasingly more sophisticated functionalities. As the computational power evolves, so

does these applications complexity. Nowadays ubiquitous and pervasive applications are complex, offer more possibilities and the users' demands require such complexity. The presented tool's major purpose is to help in two distinct scenarios. The first one is to help in the maintenance of legacy systems, by helping software analysis. The second purpose is to help model oriented software migration, by integrating software in the MDA process, and by enabling analysis at more abstract level. Our tool tends to facilitate application migration by providing high level analysis of the software, for example, using patterns.

During this work, some tools were analyzed and it was possible to conclude that some of them aim to implement the presented functionalities. However, they present shortcomings (as described) on several points, where this tool is able to succeed. The use of Prolog (and the catalog customization) to help the pattern inference process improved the achieved results. As final result of this work, a fully functional tool which implements the described functionalities was achieved. Despite having some room for improvement, the proposed objectives were achieved and implemented as tool functionalities. Considering that the MapIt might provide more functionalities than these tools (such as the diagrams abstraction), or at least, tries to provide an improved development environment (by improving the functionalities) and focusing in distinct issues (like the software migration), we consider that our tool might be an interesting contribution.

The support for other languages (such as C#, C++, etc.) is left as future work. Also, extending the pattern catalog (by using other pattern catalogs) is suggested. Finally, migrating the plugin to other IDEs (such as Eclipse) allows more users to have access to it. The integration of the Prolog inference engine on the plugin should also be considered.

Also, for future work, we are interested in scenarios such as integrating a system like JHotDraw into a mobile computing platform. To do such, we are considering performing a high level analysis, to identify relevant software components (patterns). Here, we could use a mobile computing oriented pattern catalog, possibly developing a new one, to identify relevant patterns (rather than using a generic catalog). Once the patterns are identified, we would know which code sections would need to be changed, refactored and which could remain unchanged when migrating the code. This approach would help us migrating the code into another environment, by selecting the crucial code elements. The growth in the mobile computing paradigm leads us to focus our development on software migration issues, particularly to mobile computing environments.

Another considered scenario was the JHotDraw migration into a cloud and mobile computing environment. In this case, we considered the possibility to split the tool into two components. The first component would be the interface, which will be in the mobile application, displaying the graphical editor. The second component would be the backend, where the image processing and storing would be handled, which would be stored in a cloud environment. In this scenario we consider a similar approach, by raising the abstraction level. However, in the pattern inference process, we would consider two kind of patterns (or two catalogs). One kind of patterns would help us to identify the components which

should be migrated to the mobile environment. The other kind would guide us to select the components to migrate to the cloud environment. With this approach would be possible to divide the software in two components and adapt it to a different context, even if the original software is a common desktop application.

References

1. Moria Abadi and Yishai A. Feldman. Automatic recovery of statecharts from procedural code. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 238–241, New York, NY, USA, 2012. ACM.
2. Christopher Alexander and Sara Ishikawa Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, 1977.
3. Frank Buschmann, Regine Meunier, Hans Rohnert, and Peter Sommerlad. *Pattern-Oriented Software Architecture.* John Wiley & Sons, 1996.
4. James Corbett, Matthew Dwyer, John Hatcliff, Shwan Laubach, Corina Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from Java source code. In *Proceedings of the 22nd international conference on Software engineering*, pages 439–448. ACM, 2000.
5. Liliana Favre. *Model Driven Architecture for Reverse Engineering Technologies: Strategic Directions and System Evolution.* IGI Global, 2010.
6. Martin Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, 2002.
7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
8. Yann-Gaël Guéhéneuc. *Un cadre pour la tra caabilité des motifs de conception.* PhD thesis, Université de Nantes, 2003.
9. Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Recovering binary class relationships: putting icing on the uml cake. In *Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '04, pages 301–314, New York, NY, USA, 2004. ACM.
10. K. Jinto and Y. Limpiyakorn. Java code reviewer for verifying object-oriented design in class diagrams. In *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on*, pages 471–475. IEEE, 2010.
11. Anneke Kleppe, Jos Warmer, and Wim Bast. *MDA Explained - The Model Driven Architecture: Practice and Promise.* Addison-Wesley, 2003.
12. R. Kollman, P. Selonen, E. Stroulia, T. Systä, and A. Zundorf. A Study on the Current State of the Art in Tool-Supported UML-Based Static Reverse Engineering. In *Proceedings of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, pages 22–. IEEE, 2002.
13. Stephen Mellor and Marc Balcer. *Executable UML: A Foundation for Model-Driven Architecture.* Addison-Wesley, 2002.
14. Joaquin Miller and Jishnu Mukerji. *MDA Guide Version 1.0.1.* Object Management Group, 2003.
15. Naouel Moha and Yann-Gaël Guéhéneuc. Ptidej and décor: identification of design patterns and design defects. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 868–869, New York, NY, USA, 2007. ACM.

16. Oscar Pastor and Juan Carlos Molina. *Model-Driven Architecture in Practice*. Springer-Verlag, 2007.
17. Tarja Systä. *Static and Dynamic Reverse Engineering Techniques for Java Software Systems*. University of Tampere, 2000.
18. Frank Truyen. *The Fast Guide to Model Driven Architecture - The Basics of Model Driven Architecture*. Object Management Group, 2006.
19. Markus von Detten, Matthias Meyer, and Dietrich Travkin. Reverse engineering with the reclipse tool suite. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 299–300. ACM, 2010.
20. Roel Wuyts. Declarative reasoning about the structure of object-oriented systems. In *In Proceedings of the TOOLS USA '98 Conference*, pages 112–124. IEEE, 1998.