Pattern Based User Interface Generation

André Lopes Barbosa
Universidade do Minho

User interface development is probably one of the most important phases in software development. It's a very complex and time consuming process which embraces the work of people with different backgrounds like developers and designers. With the emergence of mobile devices also came the need for developing several interfaces for multiple platforms. For all this reasons, user interface development is probably the most time consuming phase of software development.

This paper proposes a different process for developing user interfaces. This process takes a user interface pattern specified in UsiXML and maps it to the business layer source code of the software, generating a concrete user interface.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.3 [**Software Engineering**]: Coding Tools and Techniques; D.2.13 [**Software Engineering**]: Reusable Software

General Terms: Documentation, Human factors, Reliability, Standardization

Additional Key Words and Phrases: Software Engineering, User interface, Patterns, Human computer interaction

1. INTRODUCTION

Nowadays what makes a software product stand out is less technological and more related to how it handles the human computer interaction. This topic has been highly discussed in the software engineering community in the past few years.

Even with all the available tools and comprehensive bibliography it's still hard to build good user interfaces. In other fields of engineering there's a set of tested and robust patterns that can be used to build good products. In software engineering there's already a set of patterns for software architects to design their applications, but there's still little use of patterns in user interfaces.

This work needs a way to describe and store a set of patterns. UsiXML and UsiPXML are two languages that fit the requirements and were studied for this paper.

UsiXML is a user interface description language aimed at expressing user interfaces built with various modalities of interaction and independently of them. UsiXML is XML compliant to enable flexible exchange of information and powerful communication between models and tools used in user interface engineering [Usixml Consortium 2007]. UsiXML has a specification in UML that we'll see in detail on section 4.1. One of the great advantages of UsiXML is platform independence providing a multi-path development of user interfaces [Limbourg et al. 2004].

UsiPXML resulted from merging two languages, PLML [Fincher 2006] and UsiXML [Forbrig and Wolff 2010]. Pattern Language Markup Language (PLML) was introduced as an attempt to uniformly represent user interface patterns. It's natural

pg17322@alunos.uminho.pt
Departamento de Informática, Universidade do Minho Master PreThesis Reports 2012

language based so it suffers from intrinsic problems like ambiguity and inconsistency.

The main goal of this work is to study and develop a tool that can interpret a set of patterns specified in either UsiXML or UsiPXML, link them to the source code of the business layer through code annotations and generate a user interface based on the implemented functionality and the base pattern. This can help developers to build user interfaces with little effort and little knowledge on human computer interaction only by using good patterns that have been tested and are known to be robust and compliant with HCI rules.

This paper serves as an initial study for the development of a tool that generates user interfaces based on patterns. Section 2 explains how developers currently build their user interfaces and identifies the need of better tool support on this area. The third section gives some theory in the field of patterns engineering. On section 4 we'll see some languages capable of successfully describe user interface patterns, namely UsiXML and UsiPXML and a light initial specification of the proposed tool. The last section of this paper handles the conclusions of this study.

2. HOW USER INTERFACES ARE BUILT

There are several techniques and several tools to build user interfaces. Some are more intuitive and easy to learn while others are more flexible but harder to learn and thus more time consuming.

In this chapter we'll be focusing on some of the most used techniques for building user interfaces. It will be explained what are the main advantages and disadvantages related to each technique while using some examples to justify them.

Probably the most used technique is also the oldest one, manually coding interfaces. It's hard and time consuming but it's usually preferred by most experienced developers because it's more flexible and if they're good at what they're doing the final code can be very good and maintainable.

The second technique we'll see in this chapter is code generation through what you see is what you get (WYSIWYG) tools. There are many tools of this kind that support the most popular languages and frameworks for developing user interfaces. They're very used mainly by novice developers and designers. The final code isn't always the best but if you're using a robust tool there's little chance of finding bugs in it.

The third and last technique we'll see isn't the most popular in the industry but there's a lot of work on this theme in the academic world. Model driven development is widely used for the bottom layers in software development but is not that popular for the presentation layer. Although this technique is not as widely used as the previous ones it brings many advantages such as platform independence.

2.1 Manually coding user interfaces

Before more advanced tools were created, user interfaces had to be manually coded. Although the evolution that the industry has suffered since the beginnings, most developers still prefer to code manually user interfaces because it gives them more control over their work.

This is a very time consuming technique because humans have to do most of the work but in the end it really depends on what language or framework you're

working on. Most popular and modern programming languages give developers access to frameworks for building graphical user interfaces (GUI) like GTK+[1], Swing[2] or Windows Forms[3]. These examples are for the desktop side. On the web side everything is (X)HTML, CSS and JavaScript but there are a lot of frameworks to abstract from these languages like JSF[4], Struts[5] or ASP.net[6].

Most desktop GUI frameworks use the same language for views and the other software layers. This means that a lot of code has to be written in order to get things done. Frameworks like GTK+, Swing and Windows forms are very hard to use without help from more advanced tools.

Figure 1 shows a simple login window with two textboxes for username and password and a couple of buttons to login or cancel the operation.

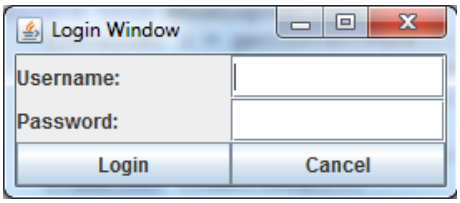

Fig. 1. Login window

On listing 1 we can see the code written to build this window using Swing. Additional information about swing can be found in [Elliott et al. 2002]

```
1  public static void main(String[] args) {
2          SwingManualTest sm = new SwingManualTest();
3          sm.showLoginWindow();
4      }
5
6      private void showLoginWindow(){
7          Container c = getContentPane();
8          c.setLayout(new GridLayout(3, 2));
9          c.add(new JLabel("Username:"));
10         c.add(new JTextField());
11         c.add(new JLabel("Password:"));
12         c.add(new JTextField());
13         c.add(new JButton("Login"));
14         c.add(new JButton("Cancel"));
15         setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
16         pack();
17         setVisible(true);
18     }
```

Listing 1. Login window using Swing, coded manually

[1] http://www.gtk.org/
[2] http://docs.oracle.com/javase/tutorial/uiswing/
[3] http://msdn.microsoft.com/en-us/library/dd30h2yb.aspx
[4] http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html
[5] http://struts.apache.org/
[6] http://www.asp.net/

It's plain Java so every control is an object. For some object oriented programming (OOP) enthusiasts this is a good thing but it's incomprehensible for designers and even for most developers this is very hard and thus very time consuming.

Fortunately, on the web side things are simpler. Most frameworks use HTML with some specific extensions to specify the views. This offer developers a more declarative paradigm which makes a lot more sense when building interfaces. This approach also produces a lot less code which makes maintenance a lot easier. Listing 2 shows the code to build the login window using JSF. Additional information about JSF can be found in [Dudney et al. 2004].

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html">
    <h:head>
        <title>Login</title>
    </h:head>
    <h:body>
        <h:form>
            <h:outputLabel value="Username:" />
            <h:inputText />
            <h:outputLabel value="Password:" />
            <h:inputSecret />
            <h:button value="Login" />
            <h:button value="Cancel" />
        </h:form>
    </h:body>
</html>
```

Listing 2. Login window using JSF

This is very different from the first example. With JSF, views are specified in a specific language. This makes the code more readable and easier to maintain.

Recently have been developed new frameworks for desktop GUI's that resemble the web ones referenced earlier. One good example is WPF[7]. It uses XAML[8] to specify views. It's a markup language based on XML and, thus, more similar to HTML than a programming language like Java or C#. Let's take a look at the login window coded for WPF. Additional information about WPF and XAML can be found in [Feldman and Daymon 2008].

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Login Window" >
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition />
            <RowDefinition />
            <RowDefinition />
        </Grid.RowDefinitions>
        <TextBlock Text="Username:" Grid.Column="0" Grid.Row="0" />
```

[7] http://msdn.microsoft.com/en-us/library/aa970268.aspx
[8] http://msdn.microsoft.com/en-us/library/ms752059.aspx

```
16          <TextBox Grid.Column="1" Grid.Row="0" Width="150" />
17          <TextBlock Text="Password:" Grid.Column="0" Grid.Row="1" />
18          <TextBox Grid.Column="1" Grid.Row="1" Width="150" />
19          <Button Grid.Column="0" Grid.Row="2" Width="150">Login</Button>
20          <Button Grid.Column="1" Grid.Row="2" Width="150">Cancel</Button>
21      </Grid>
22  </Window>
```

Listing 3. Login window using WPF

Even though this language is a lot more verbose than HTML and other markup languages it's a very good alternative for building desktop GUI's, especially if you're going to write all the code manually.

The conclusion of this section is that manually coding user interfaces isn't always a good idea depending on the technology you're using. The first frameworks that were presented use programming languages to specify the views. That doesn't look like a very good approach because it's not intuitive for the developer and incomprehensible for designers. On the other hand the later solutions use specific languages for specifying views which are more intuitive and easy to write but they force developers to learn these new languages. The other problem is that all the code produced is platform specific. If you're planning on porting your application to other devices, all the code has to be written all over again.

2.2 Code generation through WYSIWYG tools

The concept of WYSIWYG is used in a variety of situations. From text processing to building user interfaces. One of the most recognized tools of this kind is Microsoft Word for text processing. What tools of this kind attempt to do is offer the user an interface that shows exactly the final result of what they're doing.

In software development the most popular WYSIWYG environments are the ones provided by Java IDE's like *Netbeans* to build Swing interfaces or Microsoft Visual Studio that provides WYSIWYG tools for a variety of frameworks like Windows Forms, WPF or ASP.net. Figure 2 shows an example using *Netbeans*.

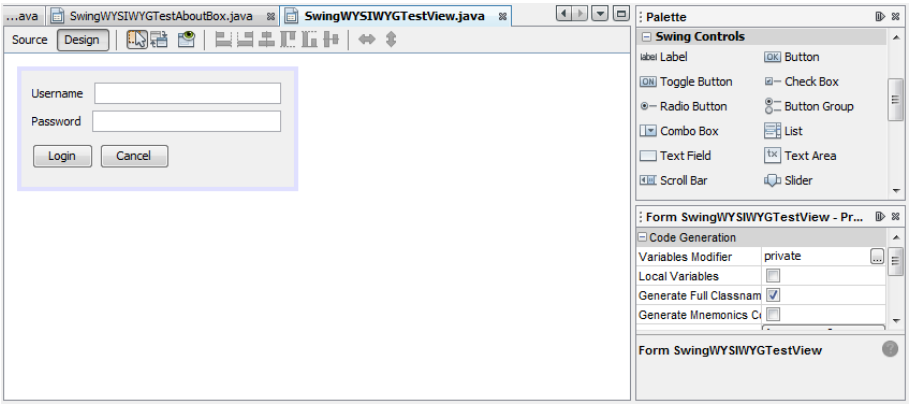

Fig. 2. Netbeans Swing WYSIWYG tool

This tool presents a GUI with a canvas where the final GUI appears and a side menu from where you can drag controls and drop them into the canvas. It's very simple and intuitive and thus very attractive for novices. The problem with this kind of tools is maintainability. It's very easy and quick to build something, if it's not very advanced, but it's a real challenge when there's a need to change the layout. Making a manual change in the generated code is not an option, mainly because is too complex but also because it's often blocked by the IDE itself. The other tools are very similar so there is no need to give further examples.

In conclusion, WYSIWYG tools are good for novices but don't suppress all the needs of the software industry where maintainability is a very important issue. There's also the problem of portability, the produced code is platform specific. Other important issue is re-usability. GUI's frameworks usually offer some way to reuse components in different contexts. This can be easily achieved while manually coding everything but it's a lot harder with a higher level of abstraction.

2.3 Model driven development of user interfaces

Model driven development defining characteristic is that software development's primary focus and products are models rather than computer programs. The major advantage of this is that we express models using concepts that are much less bound to the underlying implementation technology and are much closer to the problem domain relative to most popular programming languages [Selic 2003].

Models are easier to maintain than the code itself and, most importantly, they're platform independent. This means that the same model can be used to generate code that runs on a desktop environment, a web environment or even a mobile environment. This makes a lot of sense for user interfaces because modern applications are becoming more and more ubiquitous and it's highly complex and time consuming to build a GUI for every supported platform.

UML [Rumbaugh et al. 1999] is the industry standard for software modelling but, unfortunately, is not fit to model user interfaces. With this in mind, the software engineering community has developed some new modelling languages in the past few years to overcome this problem. The most relevant are probably UMLi [da Silva and Paton 2003], an extension to UML and ConcurTaskTrees (CTT) [Paternò et al. 1997] which aims task modelling. UMLi provides an alternative diagram notation for describing abstract interaction objects. Figure 3 shows our login window example modelled using UMLi.

With this notation you can specify inputs, outputs and actions in a way that classic UML notation doesn't support. Tasks can also be specified in UMLi, but without any extension to UML. Tasks can be modeled using Use Cases and Activity Diagrams which are part of the UML specification.

In figure 3, the upper container has four entities, *username* and *password* represent input controls while *UsernameParam* and *PasswordParam* are bindings to where the content of inputs will be stored. On the lower container are represented the actions of the window.

Task modelling has become very popular for modelling interactive systems and it's, probably, the most important method right now. A task consists on how a user can reach a goal in a specific context. CTT is the most popular language for task modelling. With CTT the task model is built in three phases:

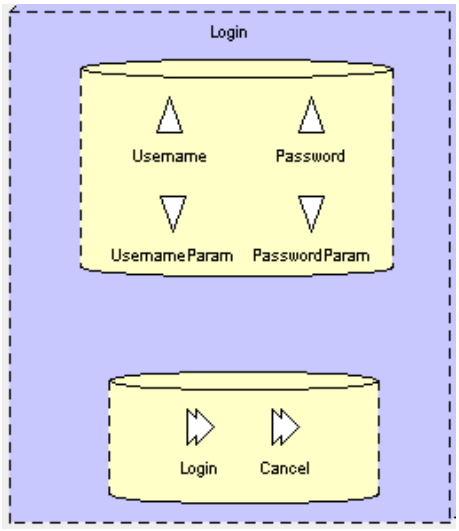

Fig. 3. Login window modelled in UMLi

—First a hierarchical logical decomposition of the tasks represented by a tree-like structure;
—Then an identification of the temporal relationships among tasks at the same level;
—And finally an identification of the objects associated with each task and of the actions which allow them to communicate with each other.

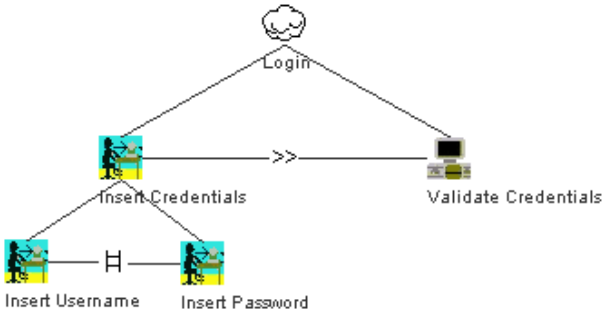

Fig. 4. Login task modelled in CTT

Figure 4 shows the login task modelled in CTT. The tool used to create this model was CTTE [HIIS Laboratory 2010] which is one of the most popular tools for the CTT language. This tool supports the creation and animation of models but doesn't offer any feature to perform any transformation to a more specific format.

Another well known tool for CTT is *IdealXML* [Montero and López-Jaquero 2006]. This tool can also be used to model tasks using CTT but it also has the

capability to transform the models into more specific ones, namely, user interface specifications in *UsiXML*.

In conclusion, there is a lot of work regarding model driven development for user interfaces and the idea that models can simplify the development process is becoming more consensual. The biggest problem with this methodology is the tool support that still isn't mature enough to be adopted by the industry. Being a method where the product of engineer's work is platform independent and both easily maintainable and reusable, model driven development will surely play an important role on the future of software development and more specifically on the development of user interfaces.

The tool proposed by this project fits in the model driven paradigm because it takes two inputs, the pattern specification, which is a model and the existing source code which is a concretion of other model and thus can also be viewed as a model. The output is a transformation of the first model based on information derived from the second one.

3. PATTERNS IN SOFTWARE ENGINEERING

Patterns are widely used in every field of engineering. One of the earlier definitions of patterns can be found on [Alexander et al. 1977]. Almost twenty years later patterns were brought to software engineering by [Gamma et al. 1995].

Patterns bring many advantages, not only they make the development of a product less time consuming and thus less expensive but can also guarantee a higher level of quality because patterns are solutions that have been tested and used in other projects.

Particularly on user interfaces, these are very important features because building a good user interface is a very complex and time consuming process. On most software projects it takes about half of the time frame allocated to that project, so patterns can help to make this process more efficient. Also there's the problem of usability. This is one of the most important aspects of software projects but its still very difficult to build a user interface compliant with human computer interaction (HCI) rules. By using patterns this can be easily achieved if the patterns are already compliant with these rules.

3.1 How are patterns documented

Patterns are usually stored in catalogues. In [Gamma et al. 1995] a pattern is composed by the following fields:

—The **Pattern name** resumes the pattern in one or two words that we use to refer to named pattern.

—The **Problem** describes in which situations the pattern should be applied.

—The **Solution** describes how the pattern works, what elements it has and how they relate to each other.

—The **Consequences** describe the side effects of using the pattern.

This is the specification used for software design patterns but it's also used in most user interface patterns catalogs. In [Vanderdonckt and Simarro 2010] documentation of patterns is divided in two categories, descriptive patterns and generative

patterns. Descriptive patterns are meant to be interpreted by humans so they describe the solution in a generic way so that the pattern can be used in a wide range of contexts while generative patterns maximize *expressivity* over *genericity* thus, they can be used in more restricted range of contexts but the solution is specific enough to be interpreted by machines.

Design patterns like the ones described in [Gamma et al. 1995] are generative patterns because their solution is specified in UML which is a formal language that can be easily interpreted by machines to perform transformations.

A list of catalogues for user interfaces can be found in [Erickson 2011]. Most of these catalogs define their solutions with text and images because there isn't a reference language to specify user interfaces. Thus most of these patterns are descriptive patterns that can only be used by humans.

In conclusion, in order to take full advantage of patterns we need a way to document them. Generative patterns are the most useful in the context of this project but to use them we need to find a language to specify these patterns so that they can be interpreted by a machine to generate a concrete user interface.

4. HOW TO SPECIFY USER INTERFACE PATTERNS

The patterns we're looking to specify are generative patterns as described in [Vanderdonckt and Simarro 2010]. These patterns have to be specified in a formal language. UML is the reference for modeling software but, as we saw on earlier sections is not ideal for user interfaces. The languages we'll see in this section are UsiXML and UsiPXML. These are high level languages that can be used to specify platform independent user interfaces.

4.1 UsiXML

UsiXML is a user interface description language aimed at expressing user interfaces built with various modalities of interaction and independently of them. UsiXML is XML compliant to enable flexible exchange of information and powerful communication between models and tools used in user interface engineering [Usixml Consortium 2007]. UsiXML is specified in UML, as seen on image 5. One of the great advantages of UsiXML is platform independence providing a multi-path development of user interfaces [Limbourg et al. 2004]. The core component of a user interface specified in UsiXML consists on the user interface model, which is itself composed by several models namely:

—**Transformation model**: contains a set of rules in order to enable a transformation of one specification to another.

—**Domain model**: describes the classes of the objects manipulated by the users while interacting with the system.

—**Task model**: describes the interactive task as viewed by the user interacting with the system. The task model is expressed according to the CTT specification [Paternò et al. 1997].

—**Abstract user interface model**: represents the view and behavior of the domain concepts and functions in platform independent way.

—**Concrete user interface model**: represents a concretization of the abstract user interface model.

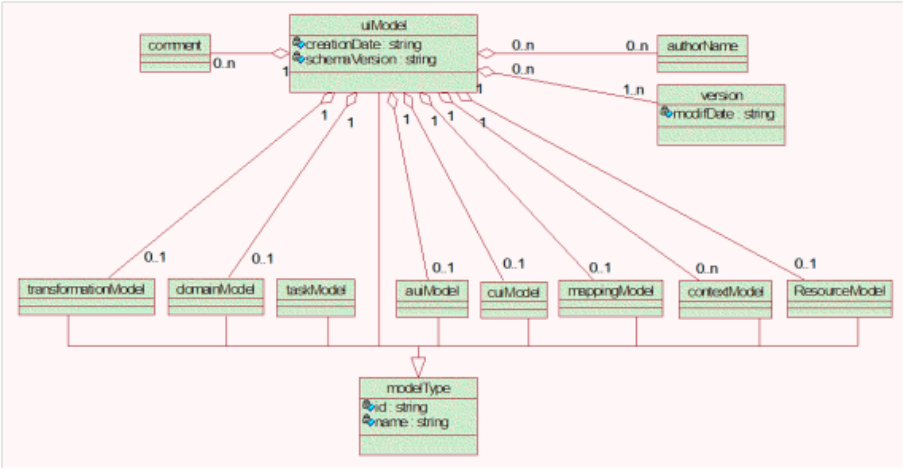

Fig. 5. UsiXML specified as a UML class diagram.

—**Mapping model**: contains a series of related mappings between models or elements of models.

—**Context model**: describes the three aspects of a context of use, which is a user carrying out an interactive task using a specific computing platform in a given surrounding environment.

—**Resource model**: contains definitions of resources attached to abstract or concrete interaction objects.

The user interface model consists of a list of component models (described above) in any order and any number. It doesn't need to include one of each model component and there can be more than one of a particular kind of model component. It's also composed by a creation date, a list of modification dates, a list of authors and a schema version.

The objective of studying UsiXML was to find out if it was suitable to specify user interface patterns. After analyzing all its components we can conclude that it's indeed suitable to describe the solution of a pattern. The pattern specification fits in the task, abstract user interface and context models. The domain and mapping models will be used to link the model with the source code that as already been written.

4.2 UsiPXML

UsiPXML results from the fusion of two languages, PLML [Fincher 2006] and UsiXML [Forbrig and Wolff 2010]. UsiXML was already studied in the last subsection so in the present subsection we'll focus on the other components of UsiPXML. PLML provides the contextual information of a pattern in UsiPXML. The main goal of PLML is to bring structure and consistency to the way patterns are described. PLML is natural language-based so it implements descriptive patterns.

It wouldn't make sense to use PLML alone with the objective of creating generative patterns but using it along with UsiXML seems a good idea because these

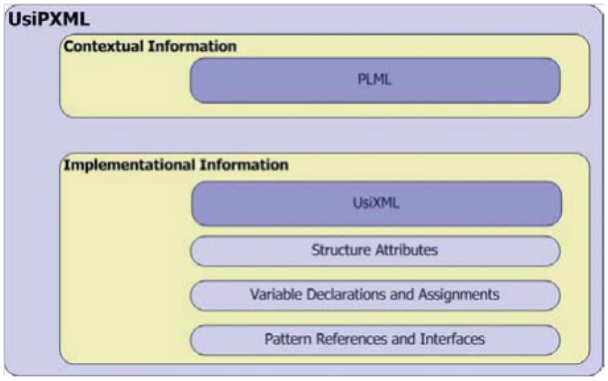

Fig. 6. Structure of UsiPXML.

two languages complement each other in this context. On section 3 was stated that a pattern was composed by a name, a problem, a solution and a list of consequences. Using UsiPXML the solution can be described in UsiXML while the other components fit in the structure of PLML.

4.3 Proposed tool

Earlyer in this section we studied a couple of languages with the potential to become standard in user interface patterns specification. UsiXML is a more concise language while UsiPXML can store more information in a structured way by merging UsiXML with PLML which is a language for descriptive patterns. In conclusion UsiPXML seems to be more suited to describe patterns but UsiXML is more generic and thus has more potential to become a standard in the software engineering community so it's probably the best option on the table.

The tool proposed by this paper should have the following list of key features:

—Read and interpret patterns specified in UsiXML models. These models have to specify at least the abstract user interface model but a task and context models should also be used if present. There should also be present a domain model and a mapping model to bind with the source code.

—Read and interpret source code of one or more OOP language with annotations in a separate XML file. These annotations should contain information to help the binding process with the pattern and provide additional information such as regarding to validations.

—Generate a concrete user interface resulting from a transformation of the pattern, taking into account the information gathered from the source code, in one or more programming languages and frameworks.

5. CONCLUSIONS

User interface development is one of the most important phases in software development but still's very hard for developers to manage this process efficiently. The data from section 2 shows that there is a need for better tools and methodologies to build user interfaces.

Patterns are very important for engineers. In section 3 we studied how are patterns used, documented and stored. We divided patterns in two categories, descriptive patterns and generative patterns. Although the ones that are more interesting for this work are generative pattens, there is more abundance of descriptive patters for user interfaces. The main reason for this is the lack of a standard language for specifying user interfaces, like UML for general software.

The future work for this project will be a more profound study of the UsiXML specification in order to develop a tool capable of reading a pattern described in UsiXML, link it with existing source code (business layer) and generate a concrete user interface.

REFERENCES

Alexander, C., Ishikawa, S., and Silverstein, M. 1977. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, New York.

da Silva, P. P. and Paton, N. W. 2003. User interface modeling in umli. *IEEE Softw. 20*, 62–69.

Dudney, B., Lehr, J., Willis, B., and Mattingly, L. 2004. *Mastering JavaServer Faces.* Wiley.

Elliott, J., Eckstein, R., Loy, M., Wood, D., and Cole, B. 2002. *Java Swing.* O'Reilly Media.

Erickson, T. 2011. The interaction design patterns page. `http://www.visi.com/~snowfall/InteractionPatterns.html`.

Feldman, A. and Daymon, M. 2008. *WPF in Action with Visual Studio 2008.* Manning Publications Co., Greenwich, CT, USA.

Fincher, S. 2006. Plml: Pattern language markup language. `http://www.cs.kent.ac.uk/people/staff/saf/patterns/plml.html`.

Forbrig, P. and Wolff, A. 2010. Different kinds of pattern support for interactive systems. In *Proceedings of the 1st International Workshop on Pattern-Driven Engineering of Interactive Computing Systems.* PEICS '10. ACM, New York, NY, USA, 36–39.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. *Design Patterns.* Addison-Wesley, Boston, MA.

HIIS Laboratory. 2010. Concurtasktrees environment. `http://giove.isti.cnr.it/ctte.html`.

Limbourg, Q., Vanderdonckt, J., Michotte, B., Bouillon, L., and López-Jaquero, V. 2004. Usixml: a language supporting multi-path development of user interfaces. Springer-Verlag, 11–13.

Montero, F. and López-Jaquero, V. 2006. Idealxml: An interaction design tool. In *CADUI'06.* 245–252.

Paternò, F., Mancini, C., and Meniconi, S. 1997. Concurtasktrees: A diagrammatic notation for specifying task models. In *Proceedings of the IFIP TC13 Interantional Conference on Human-Computer Interaction.* INTERACT '97. Chapman & Hall, Ltd., London, UK, UK, 362–369.

Rumbaugh, J., Jacobson, I., and Booch, G. 1999. *The Unified Modeling Language Reference Manual.* Addison-Wesley Professional.

Selic, B. 2003. The pragmatics of model-driven development. *IEEE Softw. 20*, 19–25.

Usixml Consortium. 2007. Usixml, user interface extensible markup language.

Vanderdonckt, J. and Simarro, F. M. 2010. Generative pattern-based design of user interfaces. In *Proceedings of the 1st International Workshop on Pattern-Driven Engineering of Interactive Computing Systems.* PEICS '10. ACM, New York, NY, USA, 12–19.