User Interface Conceptual Patterns

Pedro J. Molina[1], Santiago Meliá[1], and Oscar Pastor[2]

[1] CARE Technologies S.A.
Pda. Madrigueres, 44.
03700 Denia, Alicante, Spain
{pjmolina|smelia}@care-t.com
[2] Information System and Computation Dept.
Technical University of Valencia
Valencia, Spain
opastor@dsic.upv.es

Abstract. User Interface Patterns are not sufficiently explored at the Conceptual phase. Work in area of User Interface patterns is predominantly done at design phase but not enough work is dedicated to analysis patterns. This paper shows different examples of abstract user interface patterns and explores the impact of such patterns in the software life-cycle. Conceptual User Interface Patterns can be used for direct specification of device independent interfaces that can be refined using UI design patterns, or moreover, used to automatically obtain prototypes of the user interface specified in several devices.

1 Introduction

Patterns have proven their utility in different fields of appliance. Design patterns [6] or architectural patterns [2] are well known uses of successful patterns in computing. Patterns provide, in a given field, a common *interlingua* (or *lingua franca* [4]). At the same time, patterns document problems and its correspondent best solutions. Pattern languages are excellent tools for expressing the concepts involved in a given domain. Therefore, they constitute a valuable way to express distilled experience from real life.

In the User Interface field, most works deal with design patterns [12,14]. However, in this paper we advocate for starting employing patterns in early phases (requirements and analysis) and propagating them in later phases (design and implementation). Covering in this way, a pattern oriented development over the whole life-cycle following a similar approach as proposed in PSA[7].

In the last four years, we have been exploring first at the Technical University of Valencia and later at CARE Technologies the field of User Interface Specification based on patterns. As a result of such research, we have discovered a collection of patterns useful to deal with the specification of UI of information systems. The discovered patterns are useful to describe abstract user interfaces, guiding the design phase and providing automatic code generation strategies for fast prototyping. Such patterns are the elemental concepts described in Just-UI [9] (a model for abstract UI specification). A CASE tool implements the

P. Forbrig et al. (Eds.): DSV-IS 2002, LNCS 2545, pp. 159–172, 2002.

model assistance in the specifications and we have also developed translators to automatically transform abstract specifications to implementations.

This work presents a way for applying patterns in the Conceptual phase in the area of User Interfaces. Section 2 introduces UI Conceptual Patterns and provides examples of them. Section 3 describes the role of the patterns in the specification process. Section 4 introduces a refining approach. Sections 5 and 6 cover design and implementation phases, respectively. Section 7 provides examples of code generation from conceptual patterns. Then, related work is presented. Eventually, conclusions and references are given.

2 UI Conceptual Patterns

Patterns are widely used in the User Interfaces area. Some works like Trætteberg [12], and van Welie [14] provide collections of patterns to deal with problems in the design and implementation of user interfaces. However, we are going to focus on the conceptual phase.

Following the approach of Fowler [5], we propose to apply the concept of pattern at the conceptual phase. Moreover, we follow a pattern-oriented approach in each phase of the software development as it is used in PSA[7].

At the analysis phase, while developing the Conceptual Modelling, we can detect and document valuable patterns. Such patterns appear in the domain where users and analysts talk about the requirements of the system to be built. Particularly, applied to the User Interface context, we can typify the User Interface Conceptual Patterns such as those patterns related to user interface topics that appear in the requirements gathering at the conceptual phase.

Conceptual patterns are clearly focused on *what* (the problem space) and not focused in *how* (the solution space). Accordingly, they expressly discard design and implementation issues. The rationale is to prevent taking any design or implementation decision at the analysis phase. Such topics will be covered after at the correspondent design and implementation phases.

The pattern language developed (Fig. 1) is oriented to detect common necessities or requirements from users. The patterns help analysts indicating what information must be gathered and how it must be organized. In this sense, they provide a common language between users and analysts to describe the requirements in terms of patterns. At this phase, each pattern supplies the accumulated experience of previous UI analysis. Furthermore, such patterns imprint structural, static and behavioural restrictions that must be taken into account in the following phases (design, implementation, and maintenance).

In the classical software life-cycle, it is better to detect errors in early phases than in the later ones. Patterns, like errors, provide more advantages if they are detected and applied in early phases.

Finally, the use of User Interface Conceptual Patterns provide the same advantages as patterns, but applied at the Conceptual phase: documented analysis concepts, provide a common language for analysis, and easy reuse of analysis concepts.

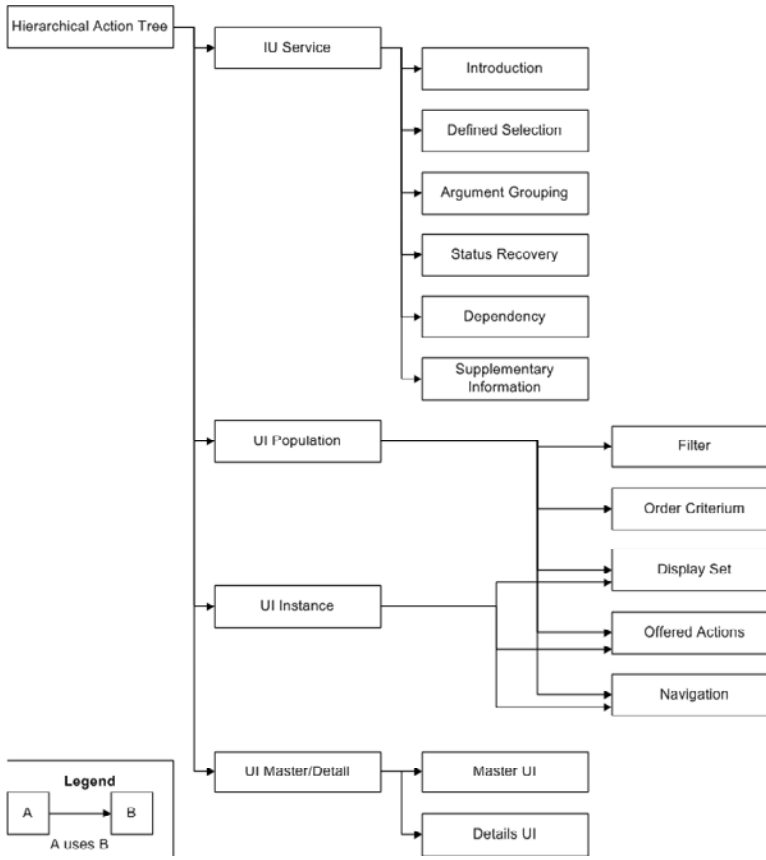

Fig. 1. Conceptual pattern language and use relationships.

2.1 Examples of UI Conceptual Patterns

To illustrate the idea of UI Conceptual Patterns we are going to describe three examples using a classical tabular form. For each pattern we will provide a *name*, an *also known as* (other names), the *problem* to be resolved, the *context* (express the conditions needed for pattern application), the *forces* (different opposing requirements), the *solution* (a core solution, not details), the *restrictions*, an *example*, the *rationale* and other *related patterns*.

Filter Pattern. A filter is a condition for searching objects. In information systems, users need very frequently specific searching tools. In the conceptual phase, analysts must capture such requirement.

Name	*Filter*
Also known as	Query
Problem	The user needs to browse and search objects belonging to a large set.
Context	In information systems is a very frequent task to search for objects. Powerful search mechanisms are needed to help the user.
Forces	The number of objects in the set may hinder the searching process. A complex query interface can be hard to understand for not experienced users.
Solution	Provides a mechanism to query the objects satisfying certain conditions. The analyst can express it in a OQL-like syntax with variables, letting the user introduce data in such variables in run time.
Restrictions	Objects to be searched must be comparable (in other words, objects have a common type to be comparable).
Example	Web searching engines, library searching facilities, etc.
Rationale	Provides a mechanism to reduce the complexity. The user can incrementally narrow the searching scope.
Related Patterns	Order Criterium, Display Set, Population observation.

Master/Detail Pattern. This pattern is typical for business applications. Head information and depicted data in a composite presentation increases the usability with respect to a solution with two single presentations.

Name	*Master/Detail*
Also known as	Master/Slave, Director/Details
Problem	The user needs to work with different sets of information units linked by a relationship. The main information unit (the master) determines the slave information units (details).
Context	Applications normally contain scenarios with several aggregated objects. The user needs to interact with all of them.
Forces	Layout may discourage a master/detail presentation. Less important information can be accessed throughout navigation. On the other hand, Master/Detail maintains the information synchronized and jointly.
Solution	Provides a unique composed scenario presenting master and detail data at the same time. Synchronizes data in the details when master data changes.
Restrictions	Details are synchronized with master information units.
Example	Invoice/Lines is a typical case of master/detail.
Rationale	The joint presentation of two or more information units allows the user to work in a unique scenario capable of performing several tasks and, at the same time, the scenario maintains the details synchronized with respect to the master component.
Related Patterns	Object observation, Population observation.

Defined Selection Pattern. If the analyst can detect a closed set of values for a given range (domain), the use of this pattern allows to reduce user errors, avoiding the introduction of invalid values.

Name	*Defined Selection*
Also known as	Closed selection
Problem	The user needs to select one or more items from a list of elements.
Context	The selection set is closed and known in advance.
Forces	The number of elements could be high. The selection key could be hidden to the user. Localizated aliases should be shown instead of keys.
Solution	Provides a table to gather the codes and aliases for each item. Ask for the default values and the minimum and maximum number of items selectable.
Restrictions	Items must be known in advance.
Example	Marital status or sex are classical examples of defined selection in UIs. Defined selection is frequently implemented as comboboxes or radio-buttons.
Rationale	The user requirements can be gathered using a simple tabular form with an item per row. The number of items may encourage different implementations to maintain the usability. This pattern can prevent user-errors: it only provides to the user the valid values.
Related Patterns	Introduction.

Fig. 1 shows the user relationships among the patterns in the pattern language developed. The complete collection of patterns can be consulted in [8, 9].

3 Using Patterns at the Analysis Phase

The conceptual pattern language can be a key concept in a model for abstract user interface specification. As described in [9] we have developed a Model-based User Interface Development Environment based in conceptual UI patterns. Patterns are employed as *bricks and blocks* (created, composed, and/or reused) to build the UI specification. Such specifications maintain the good properties of the conceptual patterns:

- Target platform independence: No design or implementation decisions have been taken in the specification.
- Describe the problem in terms of well-documented concepts.
- Reuse work form previous cases.

A CASE[1] tool is implemented supporting the gathering of complete object-oriented conceptual specifications comprising: system logic and abstract user interfaces. Such tool is being used in industrial developments with success.

[1] SOSY Modeler is the industrial CASE tool developed at CARE Technologies S.A.

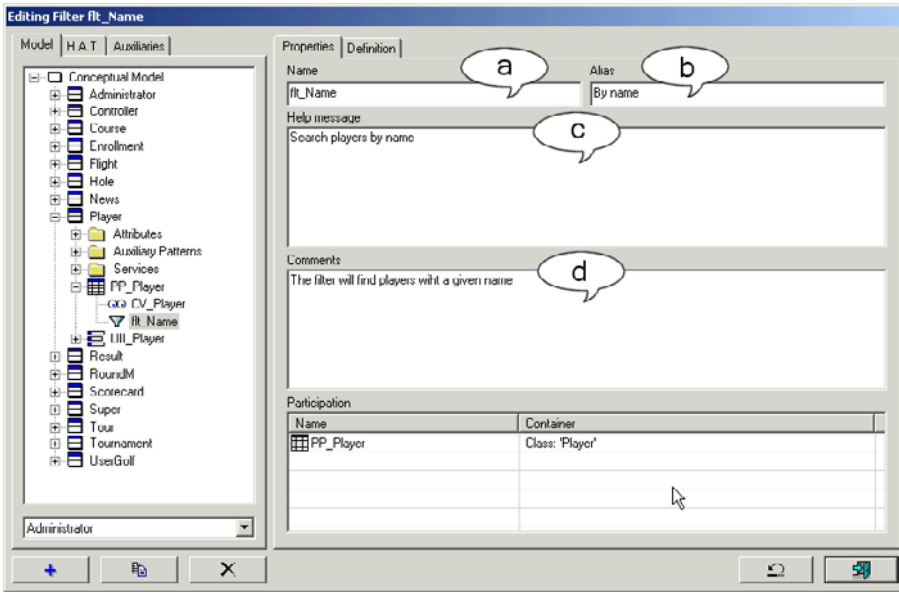

Fig. 2. Example of pattern template.

The capture of each pattern in the modeler tool is driven by a form template. The analyst can fill the template with the appropriate values to instantiate the pattern. Fig. 2 shows an example of template for gathering the *filter pattern*. In Fig. 2, fields labeled as a-d: collect a name (a), an alias (b), a help message(c), and observations(d), respectively. The pattern is instantiated by filling its template. Once instantiated (created), it can be applied to a specific context, composed with others, reused, or destroyed. Finally, the composition of patterns following the structure showed in Fig. 1 constitutes the complete UI specification of a system.

The modeler tool supports the validation of the specification. Each pattern contains a template that must be filled. When a compulsory field is empty the validation process can detect such errors in order to obtain a complete specification. Furthermore, there are rules for correct composition of patterns and type checking[2]. The validation process also checks these rules, thus providing a verification of correctness. The validation process provides a list of errors and warnings. Once the specification is finished and produces zero errors in the validation process, the specification is ready for the next phase.

[2] The validation process is out of the scope of this paper. The number of validation rules is very high, so it is not described here for brevity.

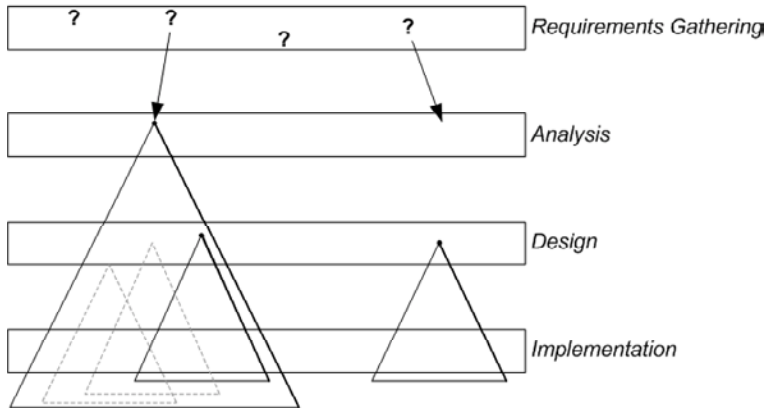

Fig. 3. The Δ effect. Implications of early pattern gathering.

4 Refining the Specification

Once the analysis specification is built and validated, the next phase is the design. At this point, designers must decide about configuration, distribution, and other topics. After that, programmers have to take additional decisions about implementation.

Following an approach similar to [7], it is a good idea to employ patterns in different phases. For example, design patterns [14,12] could be applied at design phase to maintain a pattern-oriented development.

Moreover, we propose to refine patterns in a series of successive steps from analysis to design and from design to implementation. Such refining allows adapting the solution to a specific implementation environment. The initial specification can be refined in these phases to add such decisions for a given design and then for a given implementation. Once again, this working method is an excellent way to maintain the system documented.

4.1 The Δ (Delta) Effect

Some questions arise due to the refinement-based approach: *What implications imposes a decision taken at the conceptual phase? How do UI conceptual patterns applied at the analysis phase influence the subsequent phases?*

As stated before, a pattern can supply semantic, behavioural and structural restrictions. Such restrictions have a *conical scope* or Δ (delta) effect (Fig. 3): the decision has implications from its definition to refined parts in later phases. These parts are constrained by the pattern application decision taken at upper phases. The question marks in Fig. 3 represent questions made in the Requirements phase. Such questions with its correspondent answer can instantiate conceptual

patterns in the analysis phase. Fig. 3 shows the scope of such decisions as Δ-form triangles where parts of the subsequent phases are constrained or guided by such decisions.

In this sense, decisions taken in upper phases (conceptual patterns selected at analysis phase) can guide, or help to resolve decisions at the design phase where design patterns must be selected. A wizard can help to select patterns in the design phase guided from the information captured in previous phases. For example, a wizard could suggest the reification of the Conceptual Pattern *Master/Detail* [9] at the design phase with the design pattern *Container Navigation* [12,14] and *Navigation Spaces* [14].

5 Design Approaches

Decisions taken at the analysis phase have implications and constrain the possible alternatives in the following phases. Nevertheless, to perform the design phase and implementation, there are different approaches. This section describes how conceptual patterns are useful in order to make good design choices. The most important design approaches are: Manual Design, Semiautomatic Design, and Automatic Design.

5.1 Manual Design

At the moment, OO methodologies require additional tuning in the design phase. This refinement must be provided by a designer. Designers base their decisions in knowledge and acquired experience. In this case, conceptual patterns could help the designer to constrain the population of the possible design patterns to choose from. Therefore, conceptual patterns provide experience and simplify the selection of valid alternatives for design.

5.2 Semiautomatic Design

Design is not done manually in this case. On the other hand, it is assisted by design model-based tools or by means of wizards, which guide designers to obtain designs and implementations. The implementation could be obtained in an automatic way by means of code generation. Bell [1] defines a kind of generators named translative in which, analysis domain is accompanied with design templates. It allows to get a set of architectural and design patterns for a certain architecture, and it guides the designer to choose the most adapted templates for the system. The main advantage is that it allows the intersection between a set of M domain analyses with a set of A architectures, obtaining therefore the cartesian product $M \times A$ of different possible sets of implementations. In this approach, mapping decisions between the conceptual and design patterns are not made in an automatic manner. With respect to our approach, the designer could select the templates in the design phase which are compliant to the patterns specified in the analysis phase. In such a task, this phase can be guided by an assistant like in SEGUIA [13,15].

5.3 Automatic Design

Nevertheless, we discuss in Just-UI [9] the mapping that converts from the conceptual patterns to design patterns using different transformation techniques. It normally takes a number of decisions based on design experience, speeding up and simplifying the design and implementation process. This approach is currently been developed at the CARE-T firm, using the OO-Method methodology [10] to automatically obtaining not only the UI but also system logic and persistence. The fundamental advantage is that it only requires collecting information in the requirements and analysis phases, thus obtaining in a very early phase tangible results using automatic code generation. It also eases an early development costs estimation since it allows Function Points counting [11] using these conceptual patterns. Therefore, the final productivity obtained with this approach is better with respect to the previous ones.

Table 1. Design Approaches.

	Manual Design	**Semiautomatic Design**	**Automatic Design**
Design Effort	High	Medium	Low
Design Choices	Many	Many	Few
Error prone	Yes (high)	Yes (moderate)	No
Mappings form Conceptual Patterns	It depends on designer	It depends on designer, but assisted	Always
Prototyping speed	Slow	Medium	Quick
Customization	High	Medium	Low
Fiability	It depends on designer	Medium	High

Table 1 summarizes the main characteristics of the three approaches presented for design.

Automatic design is our preferred choice. However, pure automatic translation could not be applicable or not desirable for each system, especially, when the system is out of the scope of the translator (out of the context it is based in) or an optimization must be introduced. Each system could require additional tuning that may imply changes in the automatically generated artifact. In such cases, manual design will be necessarily complemented with one of the previous approaches.

6 Deriving Implementations

Implementation phase can be also performed following different implementation approaches: Manual Programming, Code Reuse, and Code Generation. Usually, mixed approaches of these three ones are also employed.

6.1 Manual Programming

This is the most traditional but, at the same time, the most error prone and slowest approach. Here, programmers manually write part or all of the application code. A big problem appears when it is necessary to develop a system as quick as possible but the implementation is built from the scratch. Furthermore, manual programming introduces errors in this phase so it is important to complete it with exhaustive debugging and testing tasks.

6.2 Code Reuse

On the other hand, we can use software reuse for speeding up the development. The reused artifacts are obtained from previous developments and must have been highly tested. There are different code reuse artifacts like components, frameworks or idioms. In the user interface field, CIOs (Concrete Interface Objects [13]) or widgets, libraries, ActiveX and JavaBeans components are the natural artifacts to be reused. Nevertheless, for any software reuse technique to be effective, it must be easier to reuse the artifacts than developing the software from scratch. Moreover, in order to reuse an artifact, you must know what it does, where it is, and if it has been tested. Therefore, it is crucial the use of repositories to increase the reuse in development teams.

6.3 Code Generation

Translators and code generation techniques can be applied to transform analysis and design patterns to an executable program in a given platform, architecture and language. Of course, in order to derive the implementation, coding decisions must be taken automatically. Therefore, a code translator must be parameterized to support variations in the implementation produced in order to change decisions about configuration, components to employ or code topics.

In our approach we have adopted a mixed solution. On one hand, we have used code generation in domain dependent software. On the other hand, we have reused frameworks in domain independent software. We have built translators for the most used platforms at CARE Technologies. There is a desktop solution implemented in Visual Basic. The Web platform is covered with ColdFusion and JSP technology. And the Java platform is implemented using JFC/Swing.

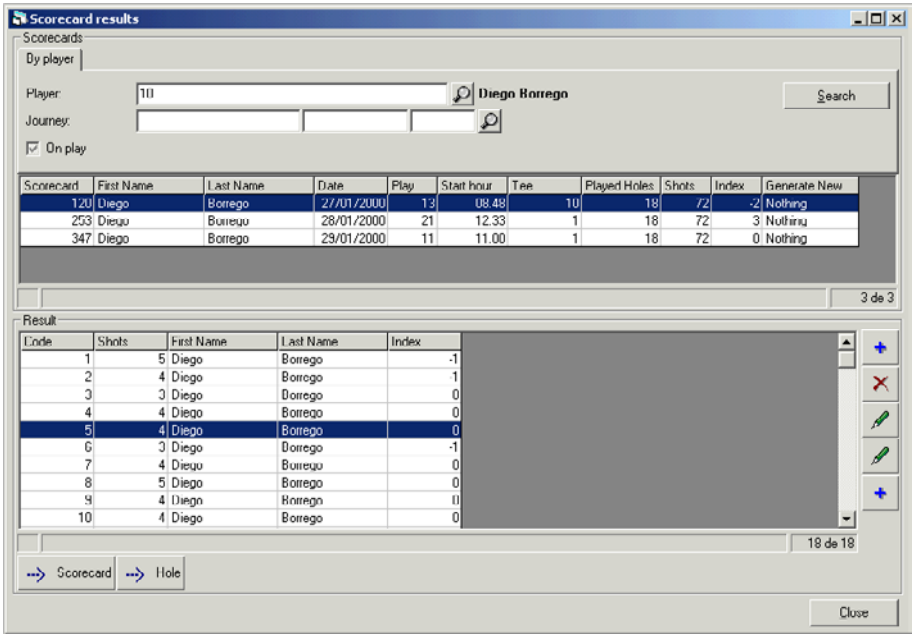

Fig. 4. Example of an implementation for Windows.

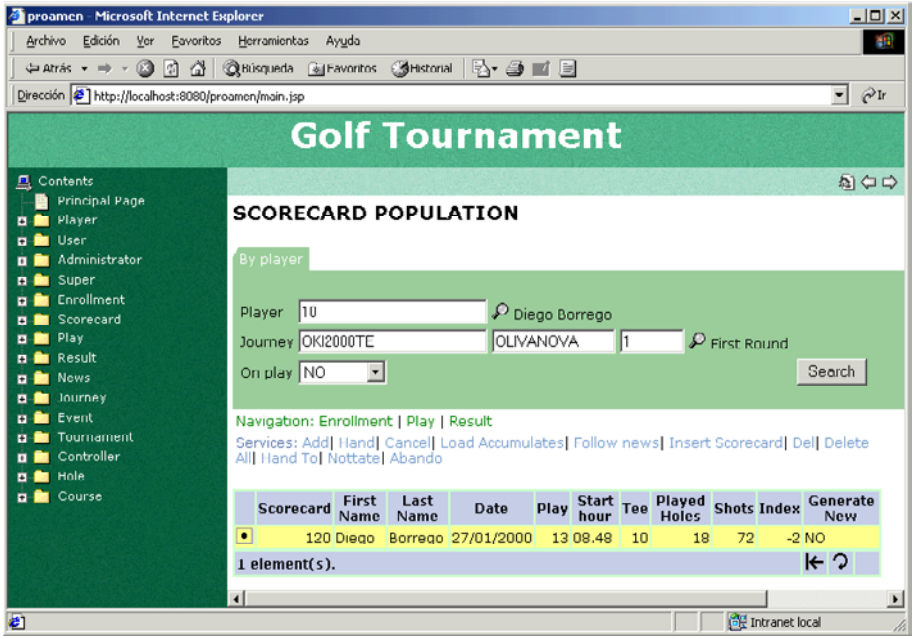

Fig. 5. Example of an implementation for Web.

7 Examples

We provide a couple of examples to show the differences and commonalities of two different implementations for the same system. The developed system manages a golf tournament. Both examples are completely generated and do not contain any manual changes.

Fig. 4 shows an example of implementation for the Windows platform produced automatically using a translator for the Visual Basic 6.0 language. In this figure, a *Master/Detail pattern* can be observed (Scorecards play the role of master and Result plays the role of detail). Note that in the master component, the scorecard number 120 is selected. The detail shows 18 results for such scorecard number 120. In the master component, a *filter* (contained in the tab labeled: 'By Player') is applied to constrain the population of scorecards. In the detail component there are two toolbars. The right vertical bar is used to *offer actions* while the bottom horizontal one is used for additional *navigation*.

On the other hand, Fig. 5 shows an implementation of the same scenario in a Web environment using JSP, HTML, and JavaScript technologies. The implementation is also obtained by means of automatic translation from the system specification. The *filter* provides the same functionality as in the previous counterpart. Here the scenario only shows the scorecards. A limitation in the target platform may discourage the *Master/Detail* presentation, providing a different but equivalent solution: users can *navigate* to Results using the link labeled 'Result' in the navigation bar. *Actions* over objects are accessible from the services bar.

8 Related Work

Martin Fowler [5] has explored the world of analysis patterns and provides lots of examples in his brilliant book. Fowler's patterns are focused on conceptual modelling and the abstraction level is closer to the patterns here presented. However, Fowler does not cover User Interface topics in his book.

On the other hand, van Welie [14] and Trætteberg [12] have developed user interface design patterns collections also focusing in usability issues and Web design. Such collections contain experience accumulated from designers and usability experts. These knowledge sources provides valuable experience and should be employed in the design phase. Accordingly, design patterns collections and languages can also be used as possible reification of Conceptual User Interface Patterns.

Coldewey and Krüger [3] proposed a framework based on a pattern language for developing form-based UIs. They also provide two conceptual patterns: *General Action* and *Dialog Category*. Granlund and Lafrenière [7] have develop the Pattern Supported Approach to cover the User Interface Design Process. In their approach they advocate for using patterns at different phases as medium for collecting experience and document the system. We absolutely agree with them, but we also intent to use patterns for supporting code generation.

Vanderdonckt describes SEGUIA [13,15] as a tool developed for the TRIDENT environment that provides a semi-automatic generation assisted by the analyst. Implemented as an expert system, SEGUIA assists analysts to transform a TRIDENT specification to source code making questions and suggesting responses to the designer.

Finally, in an excellent article, Bell [1] reviews the Object-Oriented Model-Based Code Generation technologies. He reviews specification approaches, tools support and generation strategies. Such approaches are heavily oriented to develop the system logic and not aboard the problem of User Interfaces.

9 Conclusions

We have presented the kind of patterns we have been using for user interface developing. Such patterns have proven to be useful:

1. *At the analysis phase:* Analysts can employ this kind of patterns as a tool for documenting the user interface requirements. A tool based on such patterns helped to build UI specifications for information systems. The patterns were implemented as constructor concepts in a conceptual modeler tool.
2. *At the design phase:* The specification based in such conceptual patterns can be refined applying design patterns as described in [14] and [12]. Moreover, a design tool can suggest a reduced choice of mappings from each conceptual pattern to design patterns.
3. *At the implementation phase:* The design specification can be automatically transformed to source code to build parts or the complete user interface.
4. *During the entire life-cycle:* The documented specification based in conceptual patterns can be used to exchange ideas or knowledge among the members of the development team. It constitutes a *lingua franca* as stated by Erickson [4]. Therefore, they increase the effectiveness of the communication among developers.

Furthermore, the refining of UI conceptual patterns for different platforms has allowed us to provide a systematic approach for obtaining automatic implementations directly from the analysis models. Thus, it constitutes an excellent method for rapid prototyping and for increasing the productivity of software development.

The approach described is appropriate for resolving the problems presented in ubiquitous computing in a natural way: the development team must invest its effort in the abstract specification, minimizing the manual refinements for a given platform and tuning translators for each desired platform or device. Note that such tuning is done only once (the first project for such a device), the next time, the tuning can be reused. The principle followed is:

"Build one specification, translate it to N implementations".

The work is not over, it is just starting. Patterns should be distilled, reviewed and updated. In the next months we expect to discover new patterns and continuously enrich the pattern language that we employ [8,9]. At the same time,

we think that public review is the best method to obtain valuable feedback from the research community to improve the pattern languages.

References

1. Bell R. *"Code Generation from Object Models"*. Embedded Programming Systems Journal, March, 1998.
2. Buschmann F., Meunier R., Rohnert H., Sommerland P., and Stal Michael. *"Pattern-Oriented Software Architecture – A System of Patterns"*. John Wiley & Sons Ltd., Chichester, England, 1996.
3. Coldewey J. and Krüger I. *"Form-Based User Interfaces – A Pattern Language"*, in Buschmann, Riehle (Eds.): Proceedings of the 2nd European Conference on Pattern Languages of Programming, Bad Irsee, 1997.
4. Erickson T. *"Pattern Languages as Languages"*. CHI'2000 Workshop: Pattern Languages for Interaction Design, 2000.
5. Fowler M. *"Analysis Patterns"*. Addison Wesley, 1997.
6. Gamma E., Helm R., Johnson R., and Vlissides J. *"Design Patterns: Elements of Reusable Object-Oriented Software"*. Addison Wesley, 1992.
7. Granlund Å. and Lafrenière D. *"A Pattern-Supported Approach to the User Interface Design"*. In Proceedings of HCI International 2001, 9th International Conference on Human-Computer Interaction, pages 282–286, New Orleans, USA, August, `http://www.sm.luth.se/csee/csn/publications/HCIInt2001Final.pdf`, 2001.
8. Molina P.J., Pastor O., Martí S., Fons J., and Insfrán E. *"Specifying Conceptual Interface Patterns in an Object-Oriented Method with Code Generation"*. In Proceedings of UIDIS'2001, pages 72–79, Zurich, Switzerland, May, IEEE Computer Society, 2001.
9. Molina P.J., Meliá S., and Pastor O. *"JUST-UI: A User Interface Specification Model"*. Computer-Aided Design of User Interfaces III, Ch. Kolski & J. Vanderdonckt (eds.), In Proceedings of the 4th International Conference on Computer-Aided Design of User Interfaces CADUI'2002 (Valenciennes, 15–17 May 2002), Kluwer Academics Publisher, Dordrecht, 2002.
10. Pastor O., Insfrán E., Pelechano V., Romero J., and Merseguer J. *"OO-Method: An OO Software Production Environment Combining Conventional and Formal Methods"*. Proceedings of 9th International Conference, CAiSE97, Lecture Notes in Computer Science 1250, pages 145–159, Barcelona, Spain. June, 1997.
11. Pastor O., Abrahão S., Molina J.C., and Torres I. *"A FPA-like Measure for Object Oriented Systems from Conceptual Models"*. Current Trends in Software Measument, Ed. Shaker Verlag, pages 51–69, Montreal, Canada, 2001.
12. Trætteberg H. *"Model Based Design Patterns"*. Workshop on User Interface Design Patterns (position paper), CHI'2000, The Netherlands, 2000.
13. Vanderdonckt J. *"Advice-Giving Systems for Selecting Interaction Objects"*. Proc. of UIDIS'99, 1999.
14. van Welie M. *"Web Desing Patterns (The Amsterdam Collection of Patterns)"* `http://www.welie.com/patterns/index.html`, 2000.
15. Vanderdonckt J. *"Assisting Designers in Developing Interactive Bussiness Oriented Applications"*. HCI'99, 1999.