FROM PLATO'S DUALISM TO USER INTERFACE DEVELOPMENT

Francisco Montero, Víctor López-Jaquero
LoUISE Research Group
University of Castilla-La Mancha
02071, Albacete – Spain
{fmontero, victor}@dsi.uclm.es

Keywords: user interface development, design patterns, reflection, dual specification

Abstract: Today, many devices are available, and they use different languages. From a user point of view, the system is its user interface, and this idea is used in this paper to provide a pattern-based design solution where platform independent and dependent levels are connected in a model-based user interface development and reflective environment. Platform independent and dependent levels have similarities with Plato's dual view of reality..

1 INTRODUCTION

User Interface (López-Jaquero et al., 2006) is the part of the system that receives input from, and presents information to the user. Strictly speaking, the user interface includes both hardware and software components, although in the context of software design, it refers to the software that manages the interaction with the user.

Nowadays, we have many devices and theirs associated programming languages, to handle this diversity, service providers must either devote considerable resources to developing multiple alternative user interfaces, each specialized to a particular delivery context, or must develop more flexible user interfaces.

In this paper we introduce a design pattern-based framework as a solution to flexible user interface development. This framework is a model-based user interface development environment (MB-UIDE) where a hierarchical structure is identified. This structure consists of a meta level, where MB-UIDE platform-independent models are located, and a base level where platform dependent models are hosted.

This paper is organized into three further sections. Section 2 presents relationships among user interfaces and classic philosophy. Section 3 presents our framework, a reflective-MB-UIDE that can be implemented using design patterns, how these patterns can be used is commented too. Finally, section 4 presents conclusions and future challenges.

2 USER INTERFACES AND PHILOSOPHY

Reality is the configuration of any substance, material or spiritual. Therefore, the definition of reality is looked for in the substance and configurations of existence. To test substance for reality, an example is used. A clay model is a good test. The question then is, is reality in the substance. A model of a tree can be made of a different substance, and it still represents a tree. So the reality is not in the substance.

The question then is whether the reality is in the configuration. When the configuration is changed, the reality changes. Therefore, the reality is in the configuration. This definition is adequate for both material and spiritual substance. It is accurate for thoughts in the mind, for that which is observed and for that which is communicated.

Truth is an attempt to properly represent the characteristics of unified reality. Therefore, its proper definition is the communicated representation of unified reality.

Developers often see the functionality of a system as separate from the UI, with the UI as an add-on. Users, however, do not typically make distinctions between the underlying functionality and the way it is presented in the UI. To users, the UI is the system. Therefore, if the UI is usable, they will see the entire system as usable.

User interface is often thought of as referring only to how screens look. But because users see the UI as the system, this is too narrow a definition. A broader definition of UI includes all aspects of the system design that influence the interaction between the user and the system. It is not simply the screens that the user sees, although these are certainly part of the UI. The UI is made up of everything that the user experiences, sees and does with the computer system.

2.1 Classic philosophy

Plato's allegory of the cave (Roser, 2001) is the best-known of his many metaphors, allegories, and myths. The allegory is told and interpreted at the beginning of Book VII of *The Republic* (514a-520a). The allegory is probably best presented as a story, and then interpreted—as Plato himself does.

Unlike his mentor Socrates, Plato was both a writer and a teacher. His writings are in the form of dialogues, with Socrates as the principal speaker. In the *Allegory of the Cave*, Plato described symbolically the predicament in which mankind finds itself and proposes a way of salvation. The *Allegory* presents, in brief form, most of Plato's major philosophical assumptions: his belief that the world revealed by our senses is not the real world but only a poor copy of it, and that the real world can only be apprehended intellectually; his idea that knowledge cannot be transferred from teacher to student, but rather that education consists in directing student's minds toward what is real and important and allowing them to apprehend it for themselves; his faith that the universe ultimately is good; his conviction that enlightened individuals have an obligation to the rest of society, and that a good society must be one in which the truly wise (the Philosopher-King) are the rulers.
The allegory begins with a graphic picture of the pathetic condition (see Fig. 1) of the majority of mankind. We are like chained slaves living in an underground den, which has a mouth open towards the light and reaching all along the den. Here we have been from our childhood, unable to move or to see beyond, being prevented by the chains from turning round our heads. Above and behind us a fire is blazing at a distance, but between the fire and ourselves there is a low wall like the screen which marionette players have in front of them to foster the illusion necessary for a puppet-show. We are like the strange prisoners in this den who see only their own shadows or the shadows of one another, which the fire throws on the opposite wall of the cave. To them the truth would be literally nothing but the shadows of the images, and they cannot distinguish the voices of one another from the echoes emanating from the surrounding darkness

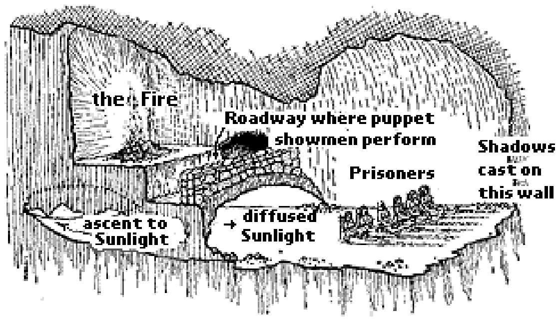

Figure 1: Graphical representation of Plato’s cave

Plato speaks of ascending and descending dialectic in his purpose of Theory of the knowledge. The ascending dialectic awakens the mind and the heart to the presence of the highest principles. Once that is achieved, the descending dialectic is the process of going back into the cave in order to be a beacon pointing others beyond the limitations of particularity. This similar directions, ascending and descending, can be found in Software Engineering and Model Driven Architecture (MDA) and in our framework.

2.2 Software Engineering: Model Driven Architecture

Recently many organizations have begun to focus attention on Model Driven Architecture (MDA) as an approach to application design and implementation. MDA encourages efficient use of system models in the software development process, and it supports reuse of best practices when creating families of systems. As defined by the Object Management Group (OMG), MDA is a way to organize and manage enterprise architectures supported by automated tools and services for both defining the models and facilitating transformations between different model types.

Models provide abstractions of a physical system that allow engineers to reason about that system by ignoring extraneous details while focusing on relevant ones, in a similar way to Plato who had two levels of knowledge (objects and concepts). All forms of engineering rely on models to understand complex, real-world systems. Models are used in many ways: to predict system qualities, reason about specific properties when aspects of the system are changed, and communicate key system characteristics to various stakeholders. The models may be developed as a precursor to implementing the physical system, or they may be derived from an existing system or a system in development as an aid to understanding its behavior.

Four principles underlie the OMG's view of MDA:

- Models expressed in a well-defined notation are a cornerstone to understanding systems for enterprise-scale solutions.
- The building of systems can be organized around a set of models by imposing a series of transformations between models, organized into an architectural framework of layers and transformations.
- A formal underpinning for describing models in a set of metamodels facilitates meaningful integration and transformation among models, and is the basis for automation through tools.
- Acceptance and broad adoption of this model-based approach requires industry standards to provide openness to consumers, and foster competition among vendors.

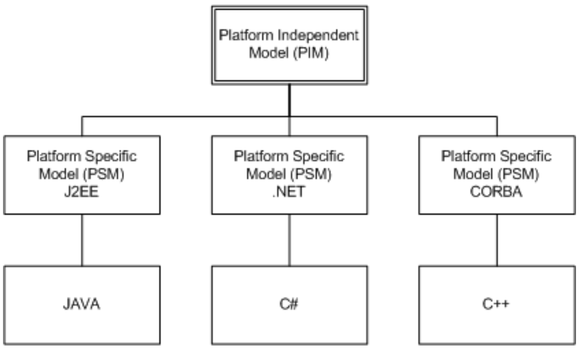

Figure 2: Model-driven architecture philosophy: one origin many destinations

To support these principles, the OMG has defined a specific set of layers and transformations that provide a conceptual framework and vocabulary for MDA. Notably, OMG identifies four types of models: Computation Independent Model (CIM), Platform Independent Model (PIM), Platform Specific Model (PSM) described by a Platform Model (PM), and an Implementation Specific Model (ISM).

2.3 Human-Computer Interaction: Model-Based User Interface Development Environments

Model-Based User Interface Development Environments (MB-UIDEs) provide a context within which declarative models can be constructed and related, as part of the interface design process (Schulungbaum, 1996; Szekely, 1995). MB-UIDEs use an explicit, largely declarative representation capturing application semantics and other knowledge needed to specify the appearance and behavior of an interactive system. The goal of the MB-UIDE is to identify reusable components of a UI and to capture more knowledge in the model, while reducing the amount of new procedural code that has to be written for each new application. In a MB-UIDE we can find several typical models: domain, task, presentation, dialog, user, etc., many of these models are independent of the platform.

Nowadays, we have many user interface description languages (UIDL) and using them we can work in an abstract user interface (AUI) level. The AUI model separates a user interface into concrete and abstract components so that a number of concrete user interface styles may be specified for a single abstract user interface. The AUI notation is an executable specification language used to define the abstract user interface. By only specifying abstract interaction once, it is hoped that development and maintenance costs will be reduced and that interaction semantics of an interactive system will remain consistent across multiple concrete user interfaces.

Domain, task and abstract user interface model are platform independent models that can be used to describe an application in a platform-independent manner. A domain model is an object model of a problem domain. Elements of a domain model are domain classes, and the relationships between them. The user task model is a representation of the tasks that the user can perform on the interface. These tasks may differ from the system, or application, tasks. The presentation model is a view of the static characteristics of an interface, mainly its layout, organization, and attributes such as fonts and colors. Finally, the user model defines the types of users of the interface and the relevant attributes of those users. Its main purpose is to influence interface generation. It is not designed to be a model of the mental state of the user at a particular time during the interaction.

3 OUR FRAMEWORK

In our framework (Montero et al., 2006) MB-UIDEs and reflection, are used together. Reflection has been proposed as a solution to the problem of creating applications able to maintain, use, and change representations of their own designs (Maes, 1987; Smith, 1982). Reflective systems are able to use self-representations to extend, modify, and analyze their own computation.

A reflective architecture yields a flexibility level that allows designers not only to extend a language itself but also to adapt and add functionality to existing systems in a transparent way. Reflection is used in several domains, such as concurrency programming, distributed systems, artificial intelligence, and expert systems. Not only functional requirements can be achieved using computational reflection,

In reflective architectures, components that deal with the self-representation and the application reside in two different software levels organized in a hierarchical manner, such as Plato's cave: *metalevel* and *base level*, respectively.

- *Metalevel*: formed by objects that carry out computation about a system materialized by objects at the base level. The computational domain, or system internal domain, deals with the information relative to structures and mechanisms that fit into the program execution

- *Base Level*: contains program objects that solve a problem and return information about the application domain.

Two processes, abstraction (bottom-up transformations) and reification (top-down transformations), occur between the levels of this hierarchy. We have adopted the word reification to indicate the inverse operation to abstraction. Abstraction implies a many-to-one transformation from the many possible variants to a single invariant form. Reification on the other hand implies *not* a one-to-many transformation, which would potentially produce an infinite variety of variants, but rather a one-to-one-of-many transformation, although the exact variant that is generated could be any one of the infinite variety of variant forms. We implemented these processes, abstraction and reification, using design patterns from (Gamma et al., 1994).

3.1 Abstraction process (bottom-up)

The state of an object is a combination of the current values of its attributes. When you call a set- method, you typically change an object's state, and an object can change its own state as its methods execute.

Objects are often discussed in terms of having a `state` that describes their exact conditions in a given time, based upon the values of their properties. The particular values of the properties affect the object's behavior. For instance, one can say that the exact behavior of an object's `getColor`() method is different if the `color` property of the given object is set to *blue* instead of *red* because `getColor`() returns a different value in the two situations. In this sense, `Context` in Fig. 3 is our application described in a platform-independent manner, that is, in `Context` we can find task, domain and presentation in an abstract way. In State we can find concrete presentation associated with different devices so many as we are considering.

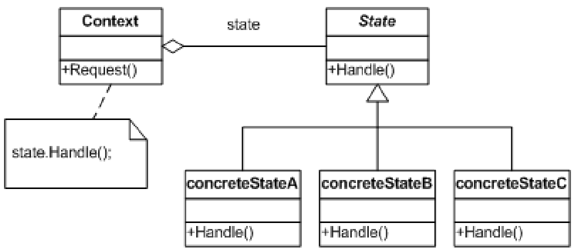

Figure 3: State pattern structure

At base level, the Decorator Pattern is used for adding additional functionality to a particular object as opposed to a class of objects. It is easy to add functionality to an entire class of objects by subclassing an object, but it is impossible to extend a single object this way. With the Decorator Pattern, you can add functionality to a single object and leave others like it unmodified. Decorator pattern can be used at our base level to added additional functionality in concrete presentation reusing concrete interaction objects functionality without use of inheritance (Fig. 4).

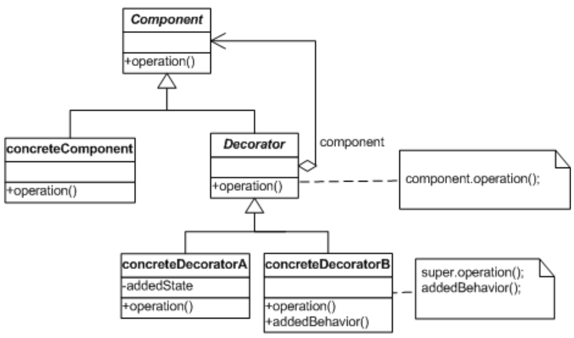

Figure 4: Decorator pattern structure

Fig. 4 shows several classes located at our base level. These classes are related with concrete interaction objects (CIOs) in different devices (`concreteComponent`), and `Decorator` classes are associated with additional functionality that can be added dynamically to CIOs.

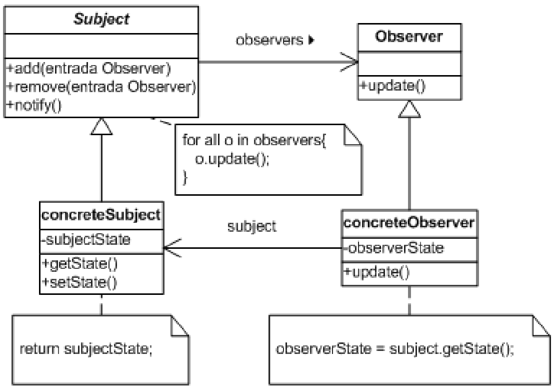

Figure 5: Observer pattern structure

3.2 Reification process (top-down)

The Observer pattern (Gamma et al., 1994) (Fig. 5) allows one object (`Observer`) to watch another (`Subject`). The Observer pattern allows the subject and observer to form a publish-subscribe relationship. Through the Observer pattern, observers can register to receive events from the subject. When the subject needs to inform its observers of an event, it simply sends the event to each observer.

A casual connection, between base and meta level, is implemented using Observer pattern. `Subject` classes are models located at meta level and `observer` classes are platform-dependent descriptions of our application, that is, concrete interaction objects.

4 CONCLUSIONS AND FUTURE WORKS

In the developed world, information technology is being embedded into more and more everyday items, and many people are increasingly reliant on electronically delivered information diversity of devices with which individuals access these electronic services. Abstraction and reification are efficient tools to address the flexible user interface development problem. This paper presented different design patterns that are successfully used in user interface development under a model-based user interface development environment. Future works will include use this approach to implement different kinds of user interfaces and applications. Non-functionality requirements and design patterns is just another challenge.

ACKNOWLEDGEMENTS

This work was supported by the Spanish CICYT project TIN2004-08000-C03-01 and the grant PCC05-005-1 from the Junta de Comunidades de Castilla-La Mancha.

REFERENCES

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusets, US: Addison-Wesley

López Jaquero, V. , Montero, F., Molina, J.P., González, P. Interfaces de Usuario Inteligentes: Pasado, Presente y Futuro. VII Congreso Internacional de Interacción Persona-Ordenador, Interacción 2006, Puertollano, Spain, 13-17, november, 2006.

Maes P. *Concepts and Experiments in Computational Reflection.* In Proceedings of OOPSLA'87, ACM Sigplan Notices, p.147-155, Orlando, Florida, October 1987

Model-Driven Architecture (MDA). http://www.omg.org/mda/

Montero, F., López Jaquero, V., Lozano, M., González, P. *A User Interfaces Development and Abstraction Mechanism*. In HCI related papers of Interacción 2004. Navarro Prieto, Raquel; Lorés Vidal, Jesús (Eds.) Springer.Verlag, Germany, 2005. ISBN: 1-4020-4204-3.

Roser, C. *Platón. Libro VII de la República.* Editorial Diálogo. 2001

Schulungbaum, E. *Model-based user interface software tools – current state of declarative models.* Graphics, visualization and usability centre, Georgia Institute of Technology, GVU Tech #96 #30. 1996

Smith, B.C. *Reflection and Semantics in a Procedural Programming Language.* PhD thesis, MIT, January 1982

Szekely, P. *Retrospective and challenges for model-based interface developments.* In: Bodart, F., Vanderdonckt, J. (eds.) Design, Specification and Verification of Interactive Systems. 1995