

# Modeling UML Software Design Patterns Using Fundamental Modeling Concepts (FMC)

Anthony Spiteri Staines  
Department of Computer Information Systems  
University of Malta  
Msida, MSD 15  
MALTA

*Abstract:* - Software design patterns mainly serve to design and develop software efficiently and effectively. Patterns are traditionally represented using UML notations and formal models. These are not always sufficient to analyze behavior. There are various problems related to visualization, abstracting, design to implementation mapping problems, etc. This paper promotes the use of Fundamental Modeling Concepts to support design patterns which offer better visualization and simplicity over the mainstream approaches. Some examples are presented.

*Key-Words:* - Software Engineering, Design Patterns, UML, Fundamental Modeling Concepts (FMC)

## 1 Introduction

Software design patterns presented in [8]–[15] explain how to resolve software design problems from both the i) system and ii) programming point of view. Design patterns propose to implement efficient and effective inexpensive solutions for recurring problems. They prevent having to re-invent a solution. A well defined design pattern should be implemented regardless of the software platform used. Thinking in terms of patterns simplifies problem solving. Patterns can communicate knowledge and architectural design. This requires that patterns are easily visualized and specified.

## 2 The UML and Design Patterns

The UML is the de facto standard used to describe recurrent design patterns [8],[12]–[13] for various reasons. UML diagrams are based visual notations. Design patterns can be classified into three subcategories [10],[14]–[15]. These are a) behavioral b) creational and c) structural. Behavioral Design Patterns describe behavior. Some behavioral patterns are i) chain of responsibility, ii) command, iii) interpreter, iv) iterator, v) visitor, etc. Creational design patterns focus on using abstract classes to create objects that are managed independently from the originator requesting their use. Some common creational design patterns are i) abstract factory, ii) builder, iii) prototype and iv) singleton. Structural design patterns focus on the interaction structure from the system point of view. These patterns describe the components of a system. Some structural design patterns are i) adapter, ii) bridge, iii) composite, iv) proxy, etc.

Some findings about design patterns and the UML are presented below:

- UML patterns can be implemented as packages
- Patterns are composed of several classes
- Patterns should exist at a higher level of abstraction than the actual class!
- Patterns should be independent of the implementation, technologies and programming languages used
- In the mindset of the software engineer thinking in terms of ‘patterns’ simplifies the solution to a particular problem
- Conceptual thinking is a different way of abstracting a problem

## 3 Design Patterns and Granularity

Fine granularity describes a system in detail, very close to the actual implementation. Conversely coarse granularity depicts a system at a higher level of abstraction. Depending on the application being developed, different levels of granularity are considered. E.g. for a simple application with no more than five classes a fine grain approach is suitable. On the other hand for complex applications exhibiting different types of behavior, a coarse grained approach is preferred.

Design Patterns also describe complex system interaction. Patterns can describe packages, clusters etc. [8]. Design Patterns can be linked to coarse granularity. A ‘class’ might actually be too small to model complex behavior. The UML is well suited to describing fine granularity, because it actually represents the implementation. Design Patterns abstracted at a high level require a different treatment.

### 3 Issues and Problems

Developing information systems for large organizations is a complex task. It is not easy to visualize the complete information system complete with functionality. Systems are composed of several viewpoints. The application domain influences these views. Most system stakeholders are non technical persons. It is possible to create a business model that will be used to develop a system model. Well defined design patterns can support software modeling.

To encourage the use of design patterns these should be well understood. Understanding system functionality implies that: i) the problem statement and ii) the solution are separate issues. Different approaches have been formulated to represent design patterns. These are i) formal models [8]-[11], ii) visual notations. Formal notations are not comprehended by many stakeholders. Consequently they are unsuitable for visualization and any small change requires re-specification. Visual notations are found in the UML and other methods supporting natural languages.

Unfortunately the UML does not clearly specify how to represent these patterns, lacking precise semantics. The UML has a set of notations for specifying both the i) Static Composition and ii) Dynamic behavior of systems. Another issue is that pattern binding to the actual physical software implementation is difficult task to achieve [8].

The UML uses different notations for describing similar system activities, creating a dilemma as which to select. Consistency between different notations involves a lot of work and resolving structural clashes. The UML is designed to communicate at the class level not at the pattern level.

### 4 A Fundamental Modeling Concept Solution

A solution for these issues is to use Fundamental Modeling Concepts (FMC) developed at Hasso-Plattner-Institute Potsdam and presented in [1]-[7]. FMC and FMC-visualization guidelines are useful to create more comprehensible and constructible models. FMC are composed of three main notations: i) Compositional Structures, ii) Dynamic Structures and iii) Value Range Structures. The UML design patterns can be represented as compositional structures. The compositional structures are supported using dynamic structures which are place transition Petri nets. These diagrams focus on system structures making them suitable to describe 'a larger granule of organization'. Recurring patterns can be easily identified. The FMC dynamic structures clearly identify and capture the behavior of the design pattern. FMC notations have evolved over a number of

years. The diagram notations like the dynamic structures are based on place transition Petri nets that have over three decades of coverage and vast literature. FMC diagrams are more practical to use when describing systems at a high level. They support communication between technical people and system stakeholders for different system requirements. FMC are based on important principles which are i) abstraction, ii) simplicity, iii) universality, iv) separation of concerns, v) aesthetics and secondary notation [1].

### 5 Examples

Some different examples of common design patterns have been selected and modeled using both UML class diagrams and FMC notations for comparison. The design patterns presented range from simple client server behavior to the abstract factory.

#### 5.1 Singleton Pattern

The singleton pattern in fig. 1 represents one of the simplest methods of interaction between a client and another object entity or class. The singleton pattern is used in remote method invocation server applications. Only a single instance of the invoked object can exist no matter how many times it is invoked.

The singleton FMC composite structure in fig. 1 and the dynamic structure in fig. 2 indicate very simple client-server interaction pattern. Both the composite structure and the Petri net can be refined and decomposed as required. A software example of the singleton pattern is simple remote method invocation.

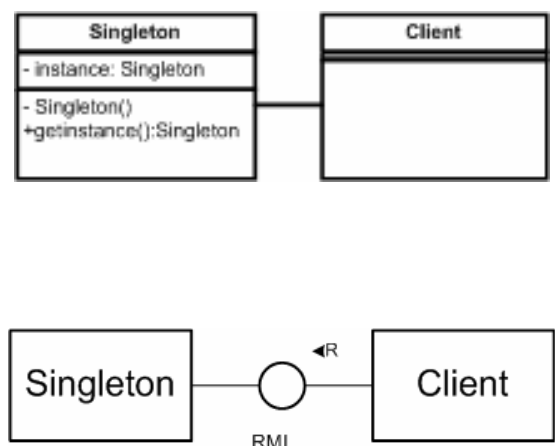


Fig. 1 Singleton Class Diagram and FMC Composite Structure

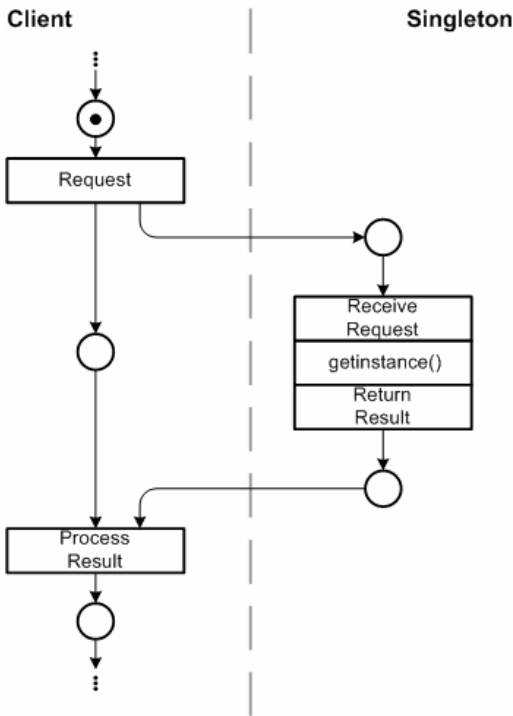


Fig. 2 Singleton Dynamic FMC Structure

## 5.2 Proxy Pattern

The proxy pattern presented in fig. 3 uses a server proxy as an interface to connect to the actual object. It is classified as a structural pattern. The responsibility for connecting to the object is the work of the proxy server not of the client. This pattern is useful for connecting to network distributed components. The proxy pattern describes remote object interaction protocols where a local object requests a remote component in a distributed information system.

The proxy pattern composite structure in fig. 4 and 5 indicates three different levels of abstraction. The basic process steps are i) a client issues a request to the server proxy. ii) the server proxy issues a request to the server. The client is not responsible for invoking the server.

The Petri net clearly indicates that the server is controlled only via the server proxy. Client interaction is with the server proxy. The server proxy issues the requests to the server and receives the reply which in turn is forwarded to the client.



Fig. 3 Proxy Pattern Class Diagram

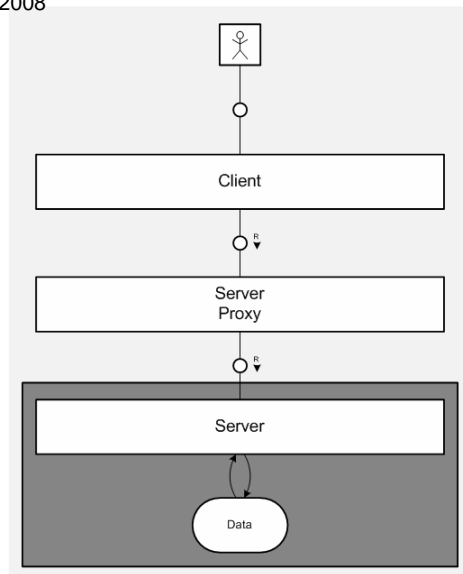


Fig. 4 Proxy Pattern Composite Structure

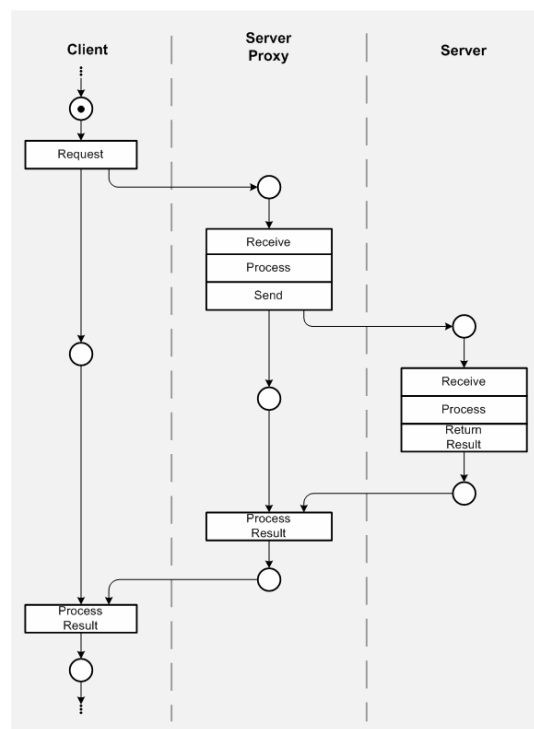


Fig. 5 Proxy Pattern Dynamic Structure

## 5.3 Command Pattern

The command pattern in fig. 6 models command and control like behavior. A client can request data from a number of other objects. This pattern uses a neatly well defined command interface that processes the client's requests connecting to the objects it requires. The client communication is managed via the interface and the

command processor. There are other patterns like the iterator that seem to be variants of this pattern.

The command pattern behavior shown in fig. 8 closely resembles real time command and control systems. E.g. of this pattern are production control systems, possible point of sale terminals connected to software, etc. At the interface level the behavior is similar to that of the server proxy. The interface invokes the command processor which in turn manages the objects and the relative data. At the command processor level it is possible to include as many objects to be controlled as necessary, this is a repeating sub pattern. If required the command processor activity can be further decomposed.

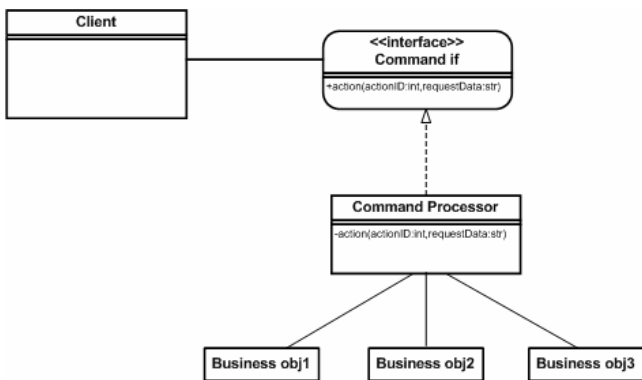


Fig. 6 Command Pattern Class Diagram

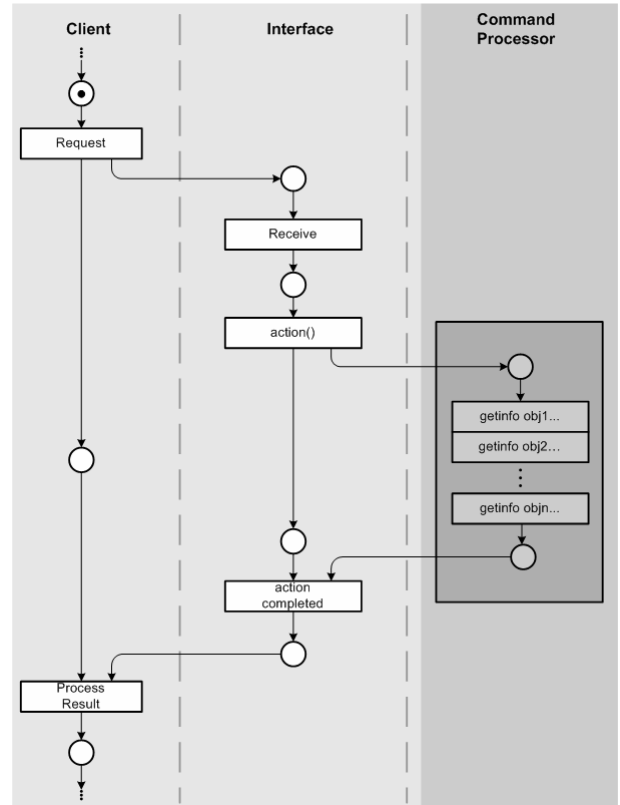


Fig. 8 Command Pattern Dynamic Structure

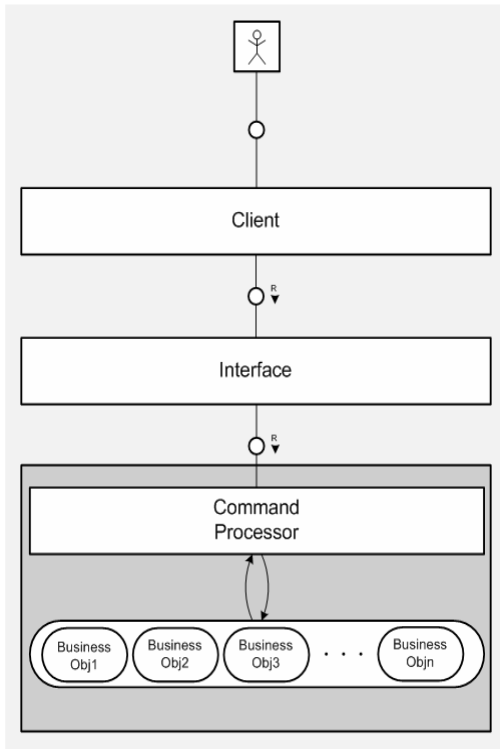


Fig. 7 Command Pattern Composite Structure

### 5.4 Abstract Factory Pattern

The pattern in fig. 9 specifies how to use abstract classes to create related objects without specifying the concrete classes. i.e. The abstract factory defines an interface having operations or methods for creating abstract objects. A software example is an order processing system having a client that has concrete classes derived from an abstract financial tools factory class to calculate the appropriate order fees. The factory method is a simplified version of the abstract factory pattern.

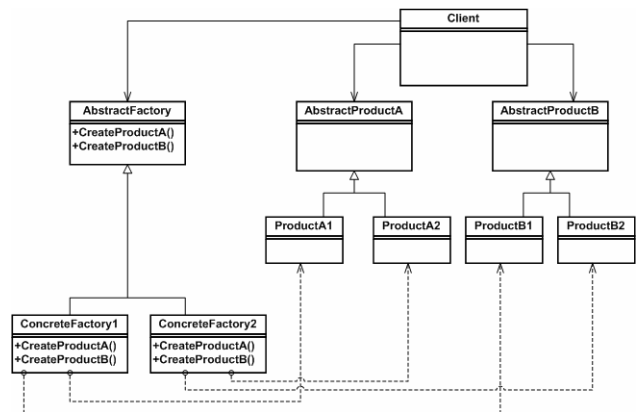


Fig. 9 Abstract Factory Class Diagram

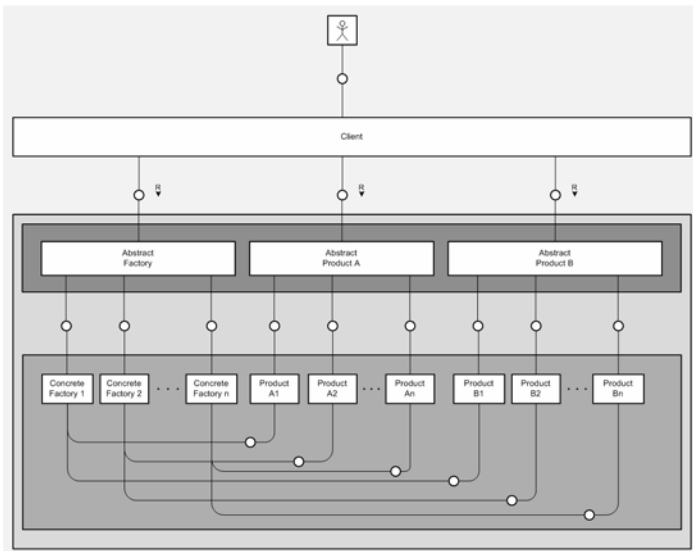


Fig. 10 Abstract Factory Composite Structure

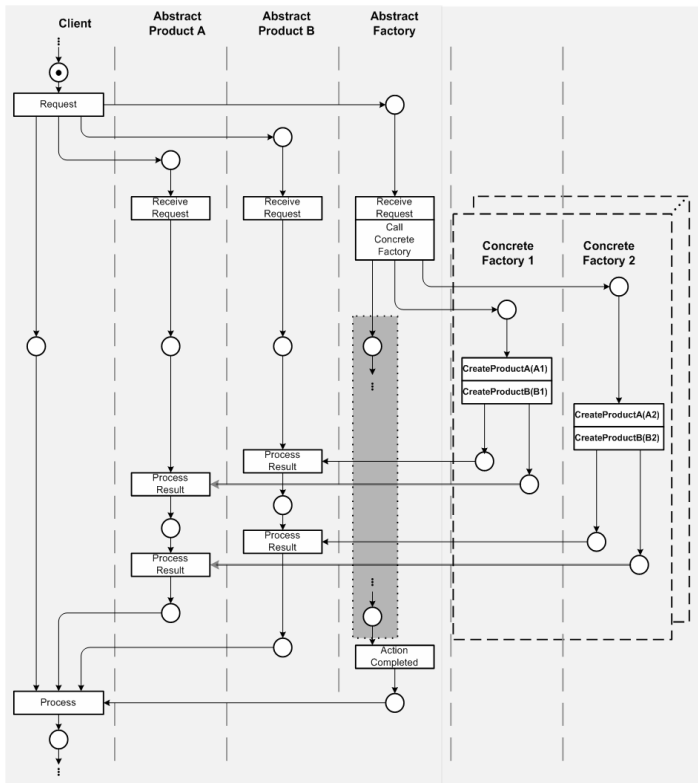


Fig. 11 Abstract Factory Dynamic Structure

Compared to other patterns the abstract factory pattern is complex. The product class depends on the abstract factory class which in turn depends on the concrete factory class for control. The FMC composite diagram in fig. 10 explains the relationships and communication channels between the client, abstract classes and concrete classes. The Petri net in fig. 11 explains that the

client request initially invokes the abstract product A and B and the abstract factory. The abstract factory is responsible for managing the concrete classes. The abstract factory thus uses the concrete factory methods to create concrete products (objects). The reply is sent to the abstract product A and B which act as interfaces to the client. The abstract factory is critical for managing the concrete factory. From the Petri net we identify at least two repeating behavior sub patterns. These are i) concrete factory process and ii) abstract product process. These repeated patterns can serve to develop and refine the model.

The composite structure in fig. 10 indicates that more concrete factories and products can be added to this structure repeatedly.

## 6 Related Work

Most of the work for design patterns is based on UML notations. In [8] the UML is used to create a library for repeating patterns. There is a problem between the actual pattern representation and its application. This is resolvable using parameterized collaborations. Although this is feasible a lot of work using the OCL is required. Many new notations have to be introduced unnecessarily complicating the scenario. In [16] it is suggested that the UML be used to guide design pattern reuse. In [12]-[13] a UML meta modeling language approach is presented. Although this is a valid approach it can become quite complex for large systems.

Design patterns from the Gof can be formalized in the B language [9]. Other valid approaches are using UML-B. These are more valuable for formal verification than visualization. In [10] a better solution is proposed where Le PUS3 / Class-Z are combined to model design patterns. Interesting diagrams which are easily readable are presented but these are still too close to the implementation, hence a fine grained approach. Design patterns are again formalized in [11] using the Disco approach considering the subject vs observer viewpoints.

The FMC notations presented in this paper offer simpler solutions. FMC notations have been used in industry. Some of the techniques discussed above can be used in conjunction with FMC notations.

## 7 Conclusion

This work can be extended to all of the UML design patterns. These patterns are applicable to other scenarios and system structures not necessarily involving software but other systems e.g. production, command and control, network patterns, etc. Simple patterns can be useful to generate more complex patterns.

The patterns described here can be refined or simplified as required. The FMC dynamic structures are based on Petri nets. Petri net theory has rules for reduction, preservation of tokens, liveness, boundness and reversibility. The dynamic structures can be evaluated from a complexity point of view and by introducing the time dimension, i.e. the Petri nets can be converted into time Petri nets and analyzed. Critical and strongly connected components in the diagrams can be identified. This would be useful for creating error recovery mechanisms and fault tolerance.

FMC focus on system related structures implying that it is possible to play about with models according to the user's needs and find those most suitable. FMC focus on fundamental building blocks thus modeling both behavior and structure.

FMC diagrams will serve to capture problem solving experience and knowledge for different domains at a high level. This could easily be shared with different system stakeholders to come up with functional models offering optimized solutions and better system comprehension. FMC offer simple solutions and are practical for use in the software development industry better than more complex approaches.

#### References:

- [1]A. Knöpfel, B. Gröne, P. Tabeling, *Fundamental Modeling Concepts*, Wiley, West Sussex UK, 2005, pp. 1-321.
- [2]B. Gröne, A. Knöpfel, R. Kugel, O. Schmidt, "The Apache Modeling Project. *Technical Report 5*", Hasso-Plattner-Institute, Potsdam, 2004. <http://www.f-m-c.org>.
- [3]B. Gröne, P. Tabeling, "A System of Conceptual Architecture Patterns for Concurrent Request Processing Servers", *Proc. of 2nd Nordic Conference on Pattern Languages of Programs*, 2003, Bergen, Norway, 2003.
- [4]B. Gröne, "Conceptual Patterns", *Proc. of 13th IEEE International Symposium and Workshop on Engineering of Computer Based Systems, ECBS*, Potsdam, Germany, Mar 2006, pp. 241-246.
- [5]P. Tabeling, "Architectural Description with Integrated Data Consistency Models", *Proc. of the 11th IEEE International Symposium and Workshop on Engineering of Computer Based Systems, ECBS*, May 2004, pp. 178- 185.
- [6]P. Tabeling, B. Gröne, "Integrative Architectural Elicitation for Large Scale Computer Based Systems", *Proc. of the 12th IEEE International Symposium and Workshop on Engineering of Computer Based Systems, ECBS*, Apr 2005, pp. 51-61.
- [7]P. Tabeling, "Multi Level Modeling of Concurrent and Distributed Systems." *Proc. of the International Conference on Software Engineering Research and Practice*. CSREA Press, Jun 2002. [http://www.fmc-modeling.org/download/publications/tabeling\\_2002multilevel\\_modeling\\_of\\_concurrent\\_and\\_distributed\\_systems.pdf](http://www.fmc-modeling.org/download/publications/tabeling_2002multilevel_modeling_of_concurrent_and_distributed_systems.pdf)
- [8]G. Sunyč, A. Le Guennec, J.M. Jězequěl, "Design Patterns Application in UML", *Proc. 14th European Conf. Object-Oriented Programming—ECOOP*. Lecture Notes in Computer Science 1850. Berlin: Springer, 2000.
- [9]S. Blazy, F. Gervais, R. Laleau. "Reuse of Specification Patterns with the B Method." *Proc. ZB 2003*. Lecture Notes in Computer Science 2651, pp. 40–57. Berlin: Springer, 2003.
- [10]A. H. Eden, E. Gasparis, J. Nicholson. "The 'Gang of Four' Companion: Formal specification of design patterns in LePUS3 and Class-Z." Department of Computer Science, University of Essex, Tech. Rep. CSM-472, ISSN 1744-8050 (2007)
- [11]T. Mikkonen. "Formalizing Design Patterns". *Proc. 20th Int'l Conf. Software Engineering—ICSE (1998)*, pp. 115–124.
- [12]K. Dae-Kyoo, R. France, S. Ghosh, E. Song, "A UML-Based Metamodeling Language to Specify Design Patterns", *Journal of Visual Languages & Computing – Science Direct, Volume 15, Issues 3-4*, June-August 2004, Pages 265-289.
- [13]R. B. France, K. Dae-kyoo, S. Ghosh, E. Song, "A UML-Based Pattern Specification Technique", *IEEE transactions on Software Engineering*, Vol 30 No. 3, Mar 2004, pp. 193-206.
- [14]Using Design Patterns in UML, <http://www.developer.com/design/article.php/3309461>
- [15]Gang of Four Design Patterns, <http://www.tml.tkk.fi/~pnr/GoF-models/html/>
- [16]N. Bouassida, H. Ben-Abdillah, "Extending UML to Guide Design Pattern Reuse", *Int. Conf. on Computer Systems and Applications*, Mar 2006, pp. 1331-1138.