Pattern Oriented Software Development: Moving Seamlessly from Requirements to Architecture

M S Rajasree , P Jithendra Kumar Reddy, D Janakiram
Distributed & Object Systems Lab
Department of Computer Science and Engineering
Indian Institute of Technology, Madras
Chennai, India
{rajasree, jithendra}@cs.iitm.ernet.in, djram@lotus.iitm.ernet.in

Abstract

Requirements Engineering (RE) deals with the early phases of software engineering namely requirement elicitation, modeling, specification and validation. Architecture of a software system emphasizes the structural constraints imposed on the application. Potential reuse in the form of software patterns are available for software designers to structure their applications. This paper proposes a pattern oriented methodology for software development. Using this approach, the skeleton of the application can be perceived up-front by using knowledge of previously identified patterns. Functional requirements of the application can subsequently be made evolving around this basic structure. The methodology thus bridges the gap between requirements specification and the architecture of the application. This approach not only leads to highly flexible and reusable design solutions, but also provides traceability of requirements in design solutions making maintenance less tedious.

Keywords: *Requirements Engineering (RE), Software Architecture, Design Patterns, Architectural Patterns*

1. Introduction

Architecture gained importance in software development process as a powerful means of software abstraction. To a great extent, architecture is distanced away from the details of the system. Potential reuse in the form of interaction modeling is captured in patterns at varying levels of granularity. Patterns enable designers to capture these interactions as reusable artifacts in software design process. These interactions in turn provide a structure for the entire application. In other words, patterns deal with the architectural aspects of a software system. Commonly occurring patterns in software systems have been categorized. Architectural pattern expresses a fundamental structural organization for the software system by providing a set of predefined subsystems and their responsibilities. It also includes rules and guidelines for organizing the relationships between these subsystems [8]. Object orientation facilitates reuse of classes within and across applications. Generalization and aggregation hierarchies enable this. Design patterns [7] are based on these principles. The fundamental structure of the entire software is not governed by design patterns. But they do influence the architecture of a subsystem.

Software development methodologies practiced today, fail to address the synergy between the requirement engineering process and architectural design. Traditional system development methodologies like waterfall model follow a sequential step. The requirements are captured first and only upon completion of this step, design and subsequent stages in the development process are addressed. Requirement elicitation mainly concentrates on the functional aspects of the system. Unless the collaborations among the entities directly contribute to the functional aspects, they are not adequately captured during this phase. We propose a development methodology wherein the systems structure in terms of the collaborations, is captured at the requirements phase itself by intuitively understanding the interactions among the participants and relating them with the previously known patterns. This gives a skeleton for the application's solution at a higher level, which can further be refined to lower level patterns.

Patterns are available at varying levels of granularity for the above mentioned approach [5, 7, 8]. Architectural patterns guide us in giving a structure for the software. Gang-of-Four (GoF) patterns address issues close to code. The same principles which form the basis of these patterns could as well be applied at an abstract level in the design process.

Choosing an appropriate structure for the application upfront, constrains and bounds the design space. Also, choice of a pattern conveys the semantics of the application. The characterization of application in terms of patterns do not stick to any formal definition of that pattern, but they do convey much more about the structure as well as computing model.

Creating an exhaustive set of patterns for the entire software domain is a never ending process. Also, it is not possible to have a complete pattern language to design a software system. In such cases, the design should be based on the relationship between the entities identified during the requirements phase. Depending on the problem domain involved, a hierarchy of patterns and a relationship between them could be figured out. As this process attains maturity, it could lead towards valuable design guidance in the form of a design handbook for the organization for specific domains and specific concerns [4] similar to the ones available in mature engineering disciplines.

The paper is organized as follows. Section 2 details the important activities in requirements engineering. Section 3 introduces the pattern oriented software development life cycle model. Section 4 explains software design as a pattern composition problem. The approach proposed in Section 3 is explained in Section 5 using a small case study. Section 6 provides a comparative account of related work. Section 7 concludes the paper with a discussion on a few ideas which includes outstanding issues for further work.

2. Requirements engineering

Requirements engineering deals with the early phases of software engineering namely requirements elicitation, modeling, specification and validation [6]. We could employ a variety of techniques for this such as interviewing, use-case modeling, essential prototyping, Class Responsibility Collaborator (CRC) modeling etc. Irrespective of the modeling techniques used, the basis of the activity remains same. Requirements analysis results in domain classes. Domain classes along with framework classes lead to class models.

Extracting information out of problem space itself may not be easy in some cases. Concepts in problem space may not necessarily be translated to concrete objects. This may be due to the fact that realization of requirements may require multiple classes. Ambiguity in problem space needs to be resolved before moving on to solutions.

Requirement specifications should not only aim at solution end from implementation point of view, but should also focus on the long life of designs. Such designs will be resilient to evolving requirements. RE is concerned with the services provided by and the constraints on a large and complex software system [9]. Apart from this, RE is also concerned with the relationship of these factors to precise specifications of systems behavior and their evolution over time and across system families. Thus RE becomes a challenging activity which has effect on the forthcoming phases and the quality of the design. For an application which is intended to be used once, the traceability of requirements is important only during the maintenance phase. However, for the derivation of architectures like productlines, this activity is more crucial. Here RE encompasses activities like planning the baseline architecture, analyzing commonality-variability etc. A pattern oriented approach for the design of frameworks for software productlines is explained in [17].

3. Pattern oriented software development life cycle model

We propose a pattern oriented life cycle model for software development. Figure 1 gives an outline of this approach. The key idea here is to have a global structure for the application based on its overall computation and communication model, guided by the knowledge available in the form of pattern catalogs and pattern languages. An intuitive understanding of the application in terms of the global data flows would suffice for this step. This is an elegant approach since the global concerns of the application are addressed here and it is possible to apply this approach to the system at varying levels of granularity. Architectural patterns [8] can be used as fundamental design decisions for the software system, imposing a structural framework for the application during this step. For example, a system where information flows in a sequential fashion can be perceived as a pipe and filter architectural pattern. Database applications and network protocols could be structured as a layers pattern [8]. The structural framework thus perceived, in turn forms a context for subsequent analysis and realization of requirements.

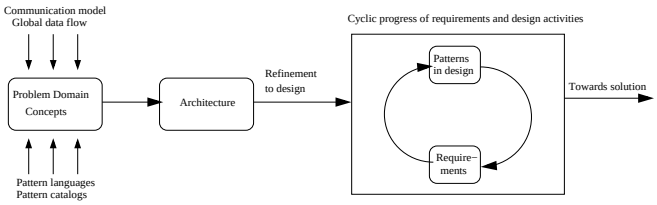

Figure 1. Pattern Oriented Life Cycle Model

Next step in the life cycle is the refinement of this architecture to design. During this phase, requirements could further be analyzed in detail, to identify lower level patterns in the systems and subsystems. By lower level patterns, we mean design patterns which could be product specific like J2EE patterns or general solutions like GoF patterns. Choice of product specific patterns again could be a

requirement driven factor. This is a cyclic activity during which, requirements as well as structure of the application get evolved simultaneously, each activity forming the context for the other.

3.1. Process improvement by patterns

The application of a well-managed, iterative and incremental development life-cycle has been pointed out as one of five characteristics of successful object-oriented projects [10]. Usually in system development process, the requirement models developed early in the development cycle undergo several working compromises during the development cycle. So it is natural that the initially perceived and documented models are not available when the development is complete. Pattern based requirement models solve this problem considerably because the basic design tradeoffs encountered by software designers are well captured in the patterns chosen to fit in the design. Considerable variation from this structure is unlikely when the design elements are filled up in this structure. These models act as powerful communication mechanisms during design and redesign process.

Software design is primarily dictated by the context in which the design activity takes place, and is influenced by enabling techniques like modularization, encapsulation information hiding, separation of interface and implementation etc. Software patterns are solutions, which are based on these enabling techniques. Patterns address the issues in design to a great extent. Requirement models can rightly be transformed to design models by means of these patterns. The domain functionality could then be provided in the design. Since the induction of a pattern is for addressing a specific concern in the system, traceability of requirements in solutions becomes easy.

3.2. Requirements engineering from a new perspective

Requirements engineering should adequately address functional and non-functional requirements of the software. In fact, if functional requirements affect only that part of the software that determines them, they typically have localized effects. On the other hand, requirements which cut across various parts of the system, can be captured from the interactions among these parts. These interactions govern the structure of the system.

While analyzing the requirements in a system it is a good idea to classify the requirements. Certain requirements could be currently existing in the system. The analysis process could stretch itself to foresee certain requirements which the system is likely to accommodate in the future at the same time making provision for incorporating the requirements. Certain requirements may necessitate potential changes in systems design. There may be some, which the system will never be able to handle. This categorization helps the systems designer to come up with an optimum architecture for the system. The designer could also make judgement about the capability of the system that has been designed based on this classification.

Architecture concerns with the structure and is like ”load-bearing walls” [13] of the software. This means that within a particular architectural framework, it is possible for the application to undergo changes, without affecting this structure. The system functionality should be evolvable within this architecture. Pattern oriented approach that we suggest becomes meaningful in this context. Since there are infinite ways of realizing these design solutions in code, it will be possible to add or remove requirements which have localized effects in the future unless they are precluded in advance by the choice of a specific pattern.

3.3. Novel approach for requirements capturing

To ensure long life for designs, they should be adaptable. Software in general and OO systems in particular should be realized as an implementation of an abstraction. At the same time, these abstractions should have the ability to accommodate requirement changes. The modular decomposition of a system should be both open and closed [1]. The designs thus have a stable core on which the resulting applications can rely on, at the same time have open portions which can accommodate context dependent variations or requirement changes. Most of the design patterns address this issue.

Portions of an application that should be kept resilient to changes and extension are often referred to as hot spots [21]. Organization of an application around such hot spots determines how well it is closed for modifications at the same time open for adaptation. Knowledge about the hot spots in a design and how they are accessed by the client software is important for all phases of software development and maintenance, whether it is construction, comprehension or evolution. The ”open-closed” principle and hot spot driven design should be conceived very early in the development life cycle; precisely at requirement capture stages itself. Pattern based models that we suggest essentially do this.

3.4. Patterns in requirements engineering

Any software development methodology has an underlying model supporting the development process. Models and abstractions constitute the basic framework for the development process in a domain. From the requirements point of view, architectural abstractions make trade-off analysis simpler, and provides a model that is easily refinable to code.

The model gets itself evolved as the development proceeds. For example, the domain processes at a coarse level could be expressed by using subsystem or components and their interactions. Further analysis could be aided by use cases for requirement modeling. Use cases lead to a conceptual model where concepts are realized using objects. Subsequently, the collaboration between objects are addressed. Subsystem interactions and relation between various use cases could be seen as requirement patterns, which can be documented. Once these requirement patterns are mapped to corresponding architectural or design patterns, the mapping could as well be used as a reusable artifact.

Interaction diagrams are one of the most important artifacts created during RE. Skillful assignment of responsibilities for the participants in the interaction diagrams is also important irrespective of the granularity of the participants, whether they be subsystems or objects. Proactive and prescriptive use of patterns assist the designer to a great extent in this step. Patterns can aid the RE process in two ways. They can result in single solutions. Secondly, they can aid in the development of reusable frameworks which are customizable design solutions. Patterns play an important role in customizing and designing application frameworks. This has been emphasized in [15].

It is generally observed that successful projects spend considerable amount of time and resources in the RE phase. It would be useful if, this phase can as well come up with requirements patterns and their corresponding mapping for related problems. Requirements patterns could be documented in the form of use cases, combination of use cases or sequence of occurrence of events.

3.5. Patterns as solution to non-functional requirements of software

In software design process, choice of an architecture is more of an activity of giving a structure to the whole application. The realization of the rest of the functionality of the software succeeds this step and ideally the software functionality should be evolvable within this architecture, without compromising its constraints.

Patterns mostly address nonfunctional requirements of software.By structuring a database application using a multiple layered pattern [8], we make the changes isolated. Problems such as lack of flexibility in most OO systems can be solved by reorganizing the design by making use of a strategy pattern [7]. The implication of this is that the design is addressing a non functional requirement of the system called flexibility. Another interesting point is that this reorganization will result in the degradation of system performance because the instance of one class needs to invoke an instance of a strategy class. Thus, pattern based requirement models since they abstract out details, serve the ideal solutions for requirement models. Also, these models enable the point at which certain quality attributes are inhibited. As a result, selecting desired solutions from a set of alternate solutions becomes easier.

In order to make sure that a system is well structured and organized, in addition to exposing global structure of the system, design should obey certain basic design principles which are to be well documented. Well structured requirements and design decisions at several layers of abstraction are crucial for understanding a detailed specification document [20]. The pattern oriented software development method proposed, assists in systematic unfolding of requirements at varying levels of abstraction and provides sound design documentation.

4. Software design as pattern composition problem

Application of patterns in software development is to be seen as a pattern composition problem. Here we provide a design solution, rather than a programming solution that is tunable only at the implementation level. Understanding of how the abstractions in software are to be adapted, extended composed and maintained is equally important as providing knowledge about the locating of the key abstractions in it.Advantages of using patterns as building blocks of architecture and the related issues are explained in [12].

The combination of collective behavior of components need to be explored at the design and architectural levels. This issue is addressed by design patterns.If we use patterns as building blocks of architecture, not only that we address a specific functional aspect, but also the interaction among the various requirements.The objectives met by design elements could be addressed using the roles played by different objects in that pattern. Requirements modeling addresses dynamic nature of the requirements in this case.

When patterns combine to generate solution architectures, the structural and behavioral composition need to be addressed. Behavior composition addresses concerns like the roles played by various objects as elements in patterns. This kind of design is referred to as responsibility-driven design or interaction oriented design in OO literature [16]. Assignment of responsibilities to objects and design of object collaborations is very important. Neglecting the importance of the creation of interaction diagrams and responsibility assignment has been pointed out as a common problem in object technology projects [2]. We believe that requirements analysis emphasizing these steps and patterns as design solutions, will alleviate this problem to a large extent. When patterns are used as compositional units of an architecture, an elegant mechanism for addressing the collaborations among the pattern participants is discussed in [3].

5. Case study

This Section illustrates the methodology we have proposed using an example of component interactions in a feedback control system. The system uses feedback from the output to control a process like any feedback control system available in control literature. Feedback unit takes the output data from the process being controlled and then makes necessary adjustment to be fed to the feedforward unit after comparing the feedback value derived from the output, with a reference value. Then, feedforward unit sends the modified output to the controlled process. Considering the global data flow in the application, the adjustor reads feedback data and reference data, the controlled process reads the modified output and the feedback unit reads the output data from the controlled process.

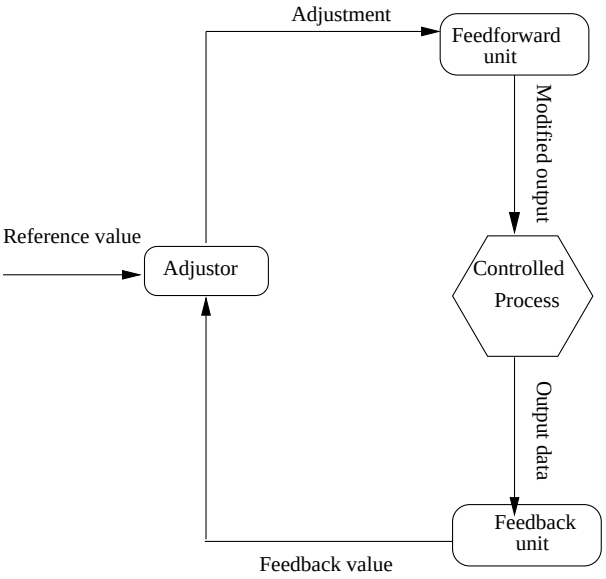

Figure 2. Component Interactions in a Feedback Control System

Even though the global data flows in the application, look like a pipe-and-filter architectural pattern [8], this architecture does not fit in here since pipe-and-filter does not allow any feedback loops. On the other hand, it fits into the blackboard pattern [8], in which several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.

Figure 3 gives one possible design for this problem. Analyzing the requirements further, it can be seen that controlled process acts as a mediator between feedforward and feedback units thus the interactions among them resemble that of a mediator pattern [7]. Controlled process can also be realized as a singleton pattern. Now, having identified these patterns, requirements could further be analyzed in detail to assign responsibilities to the classes in the patterns.

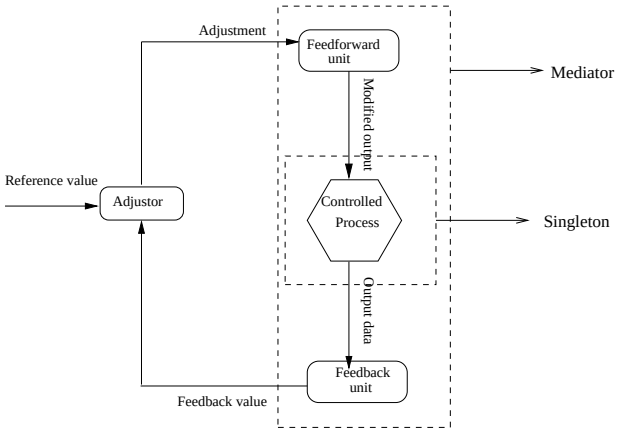

Figure 3. Design 1 for the Feedback Control system

It is natural that more than one design is possible for the same problem. In this context we give an alternative design for the same problem.This design is given in Figure 4. The functionality provided by the feedforward unit to the rest of the units should be the same, irrespective of the different control strategies used by it. A strategy pattern [7] could be used for this. The same interpretation holds with feedback unit also. To reduce the dependency between controlled process and the feedback unit, observer pattern can be used.

When alternate designs exist for the same problem, based on some trade-off analysis, the designer may have to choose the best design. In such situations, a methodology proposed in [4] aids the designer to compare the alternate designs in terms of some metrics like static adaptability, dynamic adaptability, extendibility etc. Details regarding this methodology is available in [4].

From the case study, it is evident that pattern oriented life cycle model allows the developer to systematically arrive at the design, by concentrating on the interactions existing in the domain. It is to be emphasized that trade-off analysis for alternate designs is also possible. The requirements are mapped to corresponding patterns as the design evolves. This makes traceability of requirements in solutions easy. Design solutions thus obtained are reusable since they are composed of patterns which are implementation independent, abstract entities.

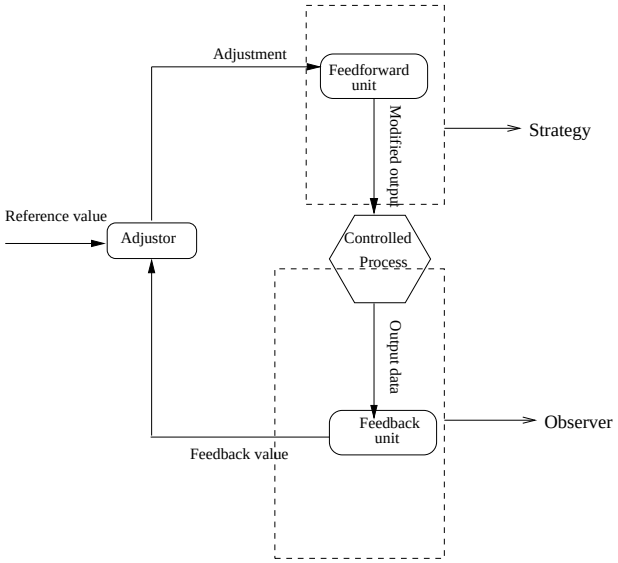

Figure 4. Design 2 for the feedback control System

6. Related work

Software engineering is defined as the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software [11]. Steps available in software development life cycle models explained in [14] do not seem to be addressing all these aspects. Since patterns are identified as distilled experience of expert designers, pattern oriented development life cycle model turns out to be a systematic and disciplined approach.

[19] proposes a mechanism that utilizes UML modeling capabilities to compose design patterns at various levels of abstractions. The approach gives emphasis to the traceability of patterns in designs. A systematic approach which takes into account the global structure of the application and subsequently refining this structure to lower level patterns does not seem to be addressed by research community yet. Our approach also opens up issues that arise when patterns are used as fundamental building blocks of architecture, most important issue being the interactions among patterns.

Pattern composition is addressed by using role diagrams in [18]. The focus here is on deriving a composite pattern, which is a combination of individual patterns. This composite pattern solves a bigger problem in the sense that the synergy of participating patterns makes the composition more than its parts. However, a generalized application development using patterns is not addressed here.

Patterns solving independent problems are documented in [8, 5, 7]. These serve only as independent pattern documentations, explaining the context, forces and solution. Our approach is towards refining and combining these solutions to build reusable application solutions.

7. Conclusions and future work

We have proposed a life cycle model using pattern oriented approach for the development of software. The approach relies on the application of previously known solutions to design problems in the form of patterns. The structure of the application is perceived in the beginning and detailed requirements elicitation follows this step. As the patterns are refined to lower level patterns, requirements also get refined. Through this approach, RE, aids architectural design by mapping the constraints imposed by the requirements to known solutions and facilitates fast trade-off analysis. Architectural modeling is supported by not only the functional and nonfunctional requirements, but also the rationale behind the formation of the pattern. The methodology proposed enables requirements capture in the context of formal architectures. We believe that when complex systems are composed from pre-existing components, the contractual obligations of the participating components also need to be captured as requirements. These contracts may lead to composition patterns, as necessitated by composition context and semantics. As part of our future work, we plan to address these issues. We foresee this as an important problem worth addressing in the context of design reuse in the form of patterns and code reuse in the form of components.

References

[1] B. Meyer. *Object Oriented Software Construction.* Prentice Hall, 1988.

[2] Craig Larman. *Applying UML and Patterns: An Introduction to Object Oriented Analysis and Design.* Prentice Hall, PTR, Upper Saddle River, New Jersey 07458, 1998.

[3] D. Janaki Ram, Jithendra Kumar Reddy, M. S. Rajasree. An Approach to form a Pattern Oriented Design Graph Using PPC Model. In *Proceedings of the SoDA'02, International Workshop on Software Design and Architecture.* Bangalore, India, December 2002.

[4] D. Janaki Ram, K. N. Anantharaman, K. N. Guruprasad, M. Sreekanth, S.V.G.K. Raju and A. Ananda Rao. An Approach for Pattern Oriented Software Development Based on a Design Handbook . *Annals of Software Engineering,* 10:329–358, October 2000.

[5] D. Schdmit, M. Stal, H. Rohnert, F. Buschmann. *Pattern Oriented Software Architecture: A System of Patterns - Vol II* . John Wiley and Sons, 1999.

[6] A. M. Davis. *Software Requirements: Analysis and Specification.* Prentice Hall, Englewood Cliffs, New Jersey, 1990.

[7] E. Gamma, R.Helm, R.Johnson, J.Vlissides. *Design Patterns, Elements of Reusable Object Oriented Software*. Addison Wesley, 1995.

[8] F. Buschman, R. Meunier, H. Rohnert, P. Sommerlab, M. Stal. *Pattern Oriented Software Architecture: A System of Patterns - Vol - I* . John Wiley and Sons, 1996.

[9] A. Finkelstein. The future of Software Engineering: 2000. In *Proceedings of 22 nd International Conference on Software Engineering*, 2000, ACM Press.

[10] Grady Booch. *Object Solution, Managing the Object Oriented Projects*. Addison Wesley, 1996.

[11] IEEE. *IEEE Standards Collection: Software Engineering*. IEEE Standard 610.12-1990, IEEE 1993.

[12] M. S. Rajasree, D. Janaki Ram, P. Jithendra Kumar Reddy. Composing Architectures from patterns. In *Proceedings of the SoDA'02, International Workshop on Software Design and Architecture*. Bangalore, India, December 2002.

[13] Perry, D. E., A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT*, 1992. 17(4).

[14] R. S. Pressman. *Software Engineering A Practitioner's Approach*. Tata McGraw-Hill Companies, Inc., 1997.

[15] R. Johnson, B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[16] R. Wirfs-Brock B. Wilkerson. Object-Oriented Design: A Responsibility-Driven Approach. In *Proceedings of Object-Oriented Programming, Systems, Languages and Applications, OOPSLA'89.*, pages 71–75, 1989.

[17] Rajasree M S, D Janaki Ram, Jithendra Kumar Reddy. Systematic Approach for Design of Framework for Software Productlines. In *Proceedings of the PLEES'02, International Workshop on Product Line Engineering: The Early Steps Planning Modeling and Managing, In Association with OOPSLA 2002*. Fraunhofer IESE, October 28, 2002.

[18] Riehle, D. Composite Design Patterns. In *Proceedings of Object-Oriented Programming, Systems, Languages and Applications, OOPSLA'97, Atlanta, Georgia, USA*, pages 218–228, October 1997.

[19] Sherif M. Yacoub, Hany H. Ammar. UML Support for Designing Software as a Composition of Design Patterns. In *UML 2001 - The Unified Modeling Language - Modeling Languages, Concepts, and Tools, 4th International Conference, Toronto, Canada, October 1-5, 2001, Proceedings*, volume 2185 of *Lecture Notes in Computer Science*, pages 149–165. Springer, 2001.

[20] M. Weber and J. Weisbrod. Requirements Engineering in Automotive Development: Experiences and Challenges. *IEEE Software*, pages 16–24, January 2003.

[21] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Addison Wesley, 1995.