A GRADUATE'S ROLE IN TECHNOLOGY TRANSFER: FROM REQUIREMENTS TO DESIGN WITH UML

Stephen Hallinan
Hewlett Packard,
Ireland
email: stephen.hallinan@hp.com

J Paul Gibson
Department of Computer Science
NUI Maynooth, Ireland.
email: pgibson@cs.may.ie

ABSTRACT

It is a universal challenge to bridge the gap between academia and industry, and between theory and practice. This challenge is particularly critical in the discipline of software engineering and is often categorised under the umbrella of *technology transfer*. Experience suggests that one of the least well understood aspects of software development is in the move from requirements to design. We support the view that software designers fail to treat design as a process rather than a product and as a consequence become experts in representing the products using models/languages but fail to master the design process. In contrast, recent developments in academia have shown that design can be more effectively taught using problem based learning techniques. This appears to produce students who better understand design as a process; but how can we ensure that this academic advancement will have a positive impact when these students move out into the real world? In this paper we analyse the role of a recently qualified student[1] in facilitating technology transfer in the form of introducing engineers to best practices in using UML to move from requirements to design.

KEY WORDS
Software Design and Development, Education, Technology Transfer

1 Introduction

1.1 Software Development's Key Weakness: Requirements and Design

Design is a creative process concerned with decision making. Designers look for solutions to problems. They search a solution space to arrive at a final design. The way in which the search is carried out may be methodical, but never deterministic. To design is to blend the old with the new: designers must use their experience and previous work (the old) to find a solution to their problem (the new). The creative side of design can be categorised as mixing three different modes of work: creating new components which are variations on already existing components and combining these new components in well-accepted ways; finding new ways of using components or combining components; and gaining insight into a problem and building a design (component) to utilise this insight.

A main problem in software engineering is that good software designers commonly have to work in all three modes, and have to have mastered the process of combining these different ways of working.

In purposeful design, the designers have some goal to aim for and this goal is evident throughout the design process. Designers are involved in each design step in an attempt to reach thir goal. Central to purposeful design is the customer requirements. There are two extremes to the way in which designers can develop understanding of the requirements: designers perform their own problem analysis to develop an initial requirements model, or designers accept a requirements specification in which the requirements are completely and consistently recorded.

In practice, design occurs somewhere between these two extremes. It is the role of designers to restructure the requirements model to best use the resources in the target implementation environment. We support the argument that this key movement from requirements to design is very poorly understood[7, 6] - even in the most mature of software development companies/industries/processes.

Through innovative teaching techniques (based on Problem Based Learning[4]), we place emphasis on mastering design as a process, rather than on mastering design as a language (or model). We argue that this produces better software engineers who can transfer these techniques into real industrial projects in order to improve the quality of the systems being developed.

1.2 Technology Transfer

Students act as one of the main conduits for technology transfer between academia and industry. Academics are responsible for ensuring that their students understand the most up-to-date theory and practice, and students are responsible for promoting these new techniques when they move out to an industrial setting. In this paper, we report on the efforts of a single individual in fulfilling their role as a newly qualified graduate in this technology transfer process.

[1]The student is the first author on the paper, and recently graduated from the NUI Maynooth, Ireland on the MSc (Software Engineering) degree programme.

1.3 The industrial case study/placement

The following article is based on the first author's experience as a Machine Control Systems engineer working for Hewlett-Packard (HP) Ireland at the Dublin Ink-Jet Manufacturing Organisation (DIMO) site. We report on the lessons that can be learnt about the UML with respect to its introduction into industry, and the feasibility of relying on recently graduated students to make such a transfer successful.

The primary members of the project team were a developer and a domain expert. The Unified Modelling Language (UML[3, 10]) was to be used to assist requirements elicitation and design. As a student, emphasis is traditionally on learning the UML as a language. Consequently, it is not surprising that software engineers are good at using UML as a static representation of a design (or designs), but are less good at using UML to record the process of design. As a software developer, the emphasis should be on applying the UML to solve real design problems, and to use the models to record and justify design decisions.

The necessary transition from being able to "speak the language" to being able to "think using the language" is analysed as part of this report. This is complementary to a second problem being analysed: that of bringing UML to an industrial setting for the first time, where there are experienced software engineers (who happen to lack experience in UML). We ask: is it possible, or desirable, or reasonable for a newly graduated student — albeit with an MSc — to instigate such technology transfer?

1.4 UML — the scope of our analysis

As part of testing the technology transfer of UML from academia to industry, we decided to focus on the following questions:

- **(i) UML is good as a common language, but does it support a good design process?** — Both the developer and the domain expert were familiar with "speaking UML", and were eager to explore its use in the description of real systems requirements and design; it would enable both parties to "talk the same language" when discussing the requirements of the application, and how functionality would be delivered to the user. However, would it facilitate the recent graduate in transferring their academic experience of moving from requirements to design out into an industrial environment?
- **(ii) UML is wide spectrum, but can it help in the transition from requirements to design?** — It was believed that UML could be used in each phase of the project. Using such a wide-spectrum language would aid in the move from requirements to design, which had been identified as a "difficult step" by the developers.

Sections 2,3 and 4 examine these questions in more detail.

2 Requirements Modelling Using UML – a Technology Transfer Challenge

The requirements elicitation phase of the project began with meetings between the domain expert and the developer. Many of the high level requirements were obtained at the preliminary meetings. It was agreed that there would be three main user groups: Technical users, Super users, and General users. Subsequently, consultations were held every two weeks with the domain expert until the functional requirements specification was initially frozen. The following sub-sections detail the elicitation of these requirements with a focus on the challenges the use of UML presented in this context, how they were overcome, and the significance this may have for future studies of technology transfer of this nature.

2.1 Challenges Using UML

There is a widely documented set of features of the UML that describe the challenges that using it presents. While these challenges are widely known, it can be difficult to determine in what capacity they are being explicitly addressed by the software engineering community and industry. The key area with regard to the challenge of technology transfer is the difference between understanding the syntax and semantics of the UML models and then the successful application of these models as part of a process.

During the construction and refinement of the UML Use case diagrams the goal was to find out how best to represent the requirements. A subset of the aforementioned known features was encountered that can make requirements engineering a challenging task.

2.2 Functional Requirements Elicitation: Working with Use Case Diagrams

The first challenge encountered with the use of Use Case diagrams was correct application so that there was a minimum of unnecessary functionality or elaboration that was non-value-added and hindered the objective that they were intended to assist. In this regard it took a number of iterations of diagrams before the core set of functionality to be delivered to each user type was agreed upon. The early iterations of diagrams contained too much information:

- unnecessary functionality in terms of what was being delivered to the user,
- complexity in terms of inter-relationships of actors and flow of control with regard to the user, and
- system functionality that was not involved in delivering value to the user.

From our teaching experience, this was a problem that would not normally have been explicitly identified when UML was being learned in an academic environment. The

catalyst to overcoming this challenge was the gradual elicitation of core features and a focus on value-added when determining the addition of new features. This foundation work in the project was enabled by the communication between the developers. This highlights the role of inter-personal communication in requirements elicitation and technology transfer that allows the parties involved to speak the same language. It was found that UML was successful as a tool to assist this.

2.3 Ambiguity in Use Case Diagrams

Wherever ambiguity is encountered, a challenge is presented in terms of how to resolve this ambiguity. In the case of Use Case diagrams, it was found to be how to correctly encapsulate system behaviour, so that the minimum of system activity is displayed, and the maximum of system functionality is delivered to the user. This also must be performed in such a manner that neither too much nor too little functionality is attributed per Use Case, as it may result in an unbalanced view of the value delivered.

The UML promotes Use Cases and Use Case diagrams to express system requirements. For modelling the requirements of a system the UML User Guide[3] advises that:

> "Modeling the requirements of a system involves specifying what that system should do (from a point of view of outside the system), independent of how that system should do"

The running theme is that the Use Case diagram should express what functionality is being delivered to the user, but hide how that functionality is delivered. This is consistent with the wider view of how Use Case diagrams should be used:

> "Use case diagrams are used to show the functionality that the system will provide and which users will communicate with the system in some way when it provides that functionality" Bennett et al.[1].

This method works well in theory, but it is easy to see where problems may arise in practice. The first point to notice is that the description of a Use Case, as per the UML User Guide[3], defines the value that is delivered to the actor by the "set of sequences of actions . . . that a system performs". It is then advised that the manner in which the system works should be hidden, or not expressed in the diagram.

This challenge was overcome in the project by focusing on the core value delivered to users, and leaving out any system requirements, as these would be properly addressed only by first concentrating on a solid set of user requirements. In the context of this project it is regarded as an approach improvement as it simplifies the manner in which UML can be applied at this stage. Without the decision to focus on core value delivered to the user, it is doubtful that this approach improvement would have been made. For this reason it is regarded that the use of UML and its support tools and documentation does not assist the beginner in resolving ambiguity in Use Case diagrams. It may be that such challenges will only be overcome with experience, implying that more case study practise would be of benefit to the beginner, and so further assist successful technology transfer.

2.4 External vs. Internal Actors

The challenge in using actors is knowing where and when to use them, and to use them in such a way that does not result in ambiguity. Through investigation it was found that the system delivers value "to an actor" and an actor "represents a role that a human, a hardware device, or even another system plays with a system". This may result in confusion in the Use Case diagram. For example, the image of an actor is that of a stick-man (which may be associated directly with people and hence users); and secondly, if there are also actors representing other systems then the exact context in which that actor is interacting with another system needs to be correctly understood. This type of contextual understanding is difficult to learn from traditional lectures, and comes only through experience of actually doing design as part of the problem solving process.

The tool used to assist the resolution of this ambiguity in the project was the documentation of Use Cases and the flows in Use Case diagrams. In this way the developer is forced to explicitly state the context in which an interaction between the user and the system is happening, and in which an interaction between the system and another system is happening. Then the meaning of the requirements is not so dependent on the graphical representation of the requirements but the meaning attributed to the diagram by the document. The reason that the confusion may only be partially resolved is that, more often than not, the Use Case documentation will be constructed using natural language. This may be a necessary evil during this phase as the requirements are exposed to frequent changes, and maintaining unambiguous Use Case documentation could prove too much of a drain on resources. For this reason Use Case diagrams are probably best used simply as a tool to aid the elicitation of the high level requirements. This is the manner in which they were used in the project. Again this highlights the need for additional resources such as documentation to assist the removal of ambiguity and assist such a technology transfer where the technology itself is proving difficult to work with. In particular the education of the parties involved as to the potential areas where ambiguities may be introduced will assist communication between the parties and enable a fast resolution.

2.5 Functional Requirements Elicitation: Refined Use Case Diagrams

One of the basic challenges presented to the beginner with the move into the detailed requirements stage is the manner in which the increasing number and type of diagram should be managed. In the case of the project the challenge was that there were multiple end-user types with different functionality associated with each. It was decided that it would be of value to the developer if the layout and flow of detail in the UML Use Case diagrams mapped well to the Functional Requirements specification. To facilitate a clear and logical structure in the Functional Requirements specification a number of improvements were made to the structure of the Use Case diagrams:

- packages were used to logically separate the relevant configuration data tables that would manage the database,
- sub-packages were used to logically separate the user groups with permissions on each of the configuration data tables, and
- each Use Case was documented in order to address its relevance to the context in which it was being used.

These actions segregated the Use Case diagrams into logical and manageable sets, which facilitated a focused effort on particular segments, where necessary. It also helped us to pin-point and discard unnecessary functionality, to reveal the key areas that would be the focus of the functional requirements specification. The improvements were aided by the use of UML packages.

In terms of technology transfer, the essence of this improvement was the introduction of logical segregation and simplicity in layout to aid further requirements elicitation, without which the benefit of using the new technology could not be fully realised. This problem, albeit simple, again highlights the fact that lessons learned with experience in modelling are not being fully addressed prior to engaging in real problems resulting in difficulty for the beginner.

Overall the adoption of Use Case diagrams for the purpose of refining the high level functional requirements was found to be fundamental to successful development, as it presented a common framework from which both parties could contribute to the development of the requirements.

3 Moving from Requirements to Design

A major challenge in any software engineering project is the move from requirements to analysis and design[8], the most basic reason being that there is no prescriptive mechanism for achieving a successful transition. There are many different methods of transition and types of analysis available, and this presented a challenge in terms of selecting the best method or combination of methods to suit the project. A number of techniques were investigated prior to deciding which approach to take.

3.1 From Requirements to Design: Choice of Transition Approach

The Use Case driven approach, as advocated by the UML User Guide, was initially investigated. It was found that to use this approach successfully may present a number of challenges to the beginner. Given that the UML had been chosen as the framework for moving from requirements to design, the UML User Guide[3] advises that: “To get the most benefit from the UML, you should consider a process that is: Use case driven, Architecture-centric, Iterative and incremental”

While the User Guide does prescribe how to perform each type of modelling, it does not prescribe an approach for moving from one phase to the next. An interesting contrast exists within this process: the Use case driven approach has already been described as having a “bottom-up” flavour, an architecture-centric approach is clearly “top-down”.

A top-down approach could be described as focusing on abstract views of the system components - for example a Component diagram or a Class diagram. This architectural blueprint serves as a solid basis against which to plan and manage software component-based development. An architecture-centric approach therefore supports such non-functional design features as scale-ability, portability, and re-usability. From our beginner’s viewpoint, for such a process to be successful – where there is a bottom-up drive to a top-down-centric process – an experienced developer is required to recognise how elements from the problem domain translate to a program architecture that accommodates high level non-functional design features. The reason for this is that while Use Case diagrams and Use Case documentation may aid the discovery of classes and relationships, there is no exact transition method for moving from requirements to design. How well the transition is performed may be the result of a combination of two main factors:

- the experience of the developer and familiarity with the problem domain, and
- how well the choice of approach to make the transition is applied.

For these reasons it was decided that the Use Case driven approach would be difficult to apply in this project.

After investigating some other transition approaches and class discovery methods in use, it was decided to use the Class-Responsibility-Collaborator (CRC)[9] approach as a starting point, with the aim of achieving a more complete transition. We were attracted by the option of using documentation to draw its input from, rather than just the UML Use cases by themselves. The nouns and verbs from the problem statement and high level requirements were

highlighted to identify potential classes and potential responsibilities. A CRC session followed. Through "standard manipulation" of cards, a set of classes was chosen that the developer was comfortable with. As the design is not the actual implementation, it is sufficient that the developer is comfortable with the initial set of classes in order to proceed. This reflects the maxim that there is more than one way to model a system. While the CRC approach did not need the UML to be applied, it was possible to translate directly into UML the results of the CRC session. It is therefore seen as a benefit of using the UML that a range of object-oriented techniques can translate directly into the UML. The interoperability of such techniques and tools facilitated a successful technology transfer for this section of the project.

3.2 Initial Analysis: Super Imposition of the BCED Approach

The UML design was now at the stage where analysis of the system requirements could commence. After a number of analysis methods were investigated it was decided that the BCED approach was the most suitable for this project. The Boundary-Control-Entity-Database (BCED) approach is an extension of the Boundary-Control-Entity (BCE) approach, which may be described as an analysis pattern for web application design. It is derived from the Model-View-Controller (MVC) framework[8].

The BCE approach involves a division of classes into three categories. The Boundary classes correspond to the user interface or presentation layer, the Control classes correspond to the middle layer that contains the application logic and the Entity classes represent the data management layer in a three-tier architecture. In this way, each class division maps to a specific tier in the physical domain. In the BCED approach the entity package is separated out into those packages that store database information in program memory and those that contain the database specific connection, read and write information. This separation is a good feature of design as it "provides a level of indirection between the application and the database". The approach was super-imposed upon the project's Class diagrams that were the result of applying the CRC technique. In a similar vein to the application of the CRC approach, the use of the UML facilitated the application of this analysis method.

After the application of the BCED approach the system design now had a clear structure and was ready to be developed and improved using known design techniques and tools. The use of these techniques and tools is detailed in the following sections with regard to the use of UML as part of technology transfer.

3.3 Sequence Diagrams

Several scenarios were developed to test the program design and investigate how the user functions and system functions would operate under certain circumstances. The construction of the Sequence diagrams revealed much about the current design, such as where there were unnecessary classes and methods, and incorrect apportionment of functionality, i.e. methods in inappropriate classes. The use of Sequence diagrams also helped focus on dataflow through the system. This was an aspect of the system that had not been previously examined. The Sequence diagrams show how the system should work dynamically, so it serves as an artefact to refer to when developing how the system passes messages to achieve some goal. The primary function of the sequence diagrams is as an analysis tool[3]: "Modeling a flow of control by time ordering emphasizes the passing of messages as they unfold over time, which is a particularly useful way to visualize dynamic behavior in the context of a use case scenario".

As a result of the Sequence diagrams, some classes were discarded and their functionality was subsumed into one of the main controlling classes. The initial spread of functionality among classes had been too great, so that classes that had only one purpose. Consequently, there had to be more message passing sequences than should be necessary. For these reasons the use of the UML Sequence diagrams is regarded as a useful design tool to test system behaviour and reduce complexity. Refined Sequence diagrams also serve as an artefact to demonstrate how the system works in certain situations. Only a bare understanding of the UML is necessary to understand the purpose these diagrams serve and are therefore a goodexample of straightforward technology that can be expected to be successfully transferred by recently qualified students.

3.4 Aggregation and Generalisation

It was decided to examine the class model with respect to enhancing the relationships from an object-oriented perspective. Two examples of this are aggregation and generalisation.

The model was first examined for opportunities to use aggregation. Consequently, many of the disadvantages that are associated with inheritance could be avoided. One of the main sources of these disadvantages is the fact that as functionality is inherited down the chain of sub-classes, if the super-classes or base class changes it can have far reaching effects for the sub-classes that were not initially accounted for. This is described in Requirements Analysis and System Design as: "Changes to the implementation of a super-class will have a largely unpredictable effect on the subclasses in the application system. This is true even if the super-class interface remains unchanged"[8].

The model was then examined for opportunities to use generalisation (or inheritance). The basis for using this type of relationship in a design is that it enables re-use of functionality. Therefore to see if this type of relationship may be of benefit in a design, the functional requirements and the current design may be examined with the aim of identifying common functionality. A common way in which

generalisation is implemented in designs is to identify naturally occurring hierarchies that may translate into a superclass/sub-class hierarchy in the program design. The most obvious case for this in the project was the fact that there were three user groups assigned. It followed that the Super user is a type of General user, and the Technical user is a type of Super user. This was then explicitly specified in the class model, where a purely abstract class was also introduced.

The application of these two design techniques to the system demonstrate a successful case of transferring knowledge expressed previously only in theory, to real-world application. Again the use of UML was an important factor as it was the medium through which these techniques could be expressed and referenced.

3.5 Collaboration Diagrams

Collaboration diagrams are a kind of Interaction diagram similar in nature to the Sequence diagrams previously described. The aim of a Collaboration diagram is to illustrate a certain degree of user or system functionality. They indicate where branching or iteration may occur within the program design. A Collaboration diagram was developed for each scenario as per the Sequence diagrams. For a number of scenarios it was found that the same Collaboration diagram was sufficient. This is what is known in modelling as a mechanism, which aids the reduction of complexity within the program. As the diagrams were developed, the Class diagram was correspondingly modified to reflect any new additions or changes. It was decided at this point that putting further effort and time into the development of the design at this level would not add much more value as it was quite possible that the design would undergo modification once the implementation commenced.

For the reasons described above the use of Collaboration diagrams was beneficial, but for the level of complexity involved in the project and the experience of the developer in applying such design tools, it was reaching the stage where it was becoming more time consuming and less value-added to perform further analysis and design. This may indicate an area that the people involved in the technology transfer should be careful to avoid, which is one where it is becoming more difficult to realise value from the tools and techniques to be transferred. This may lead to a de-valuing of the tools and techniques resulting in a discontinuation of their use.

4 Effect of Evolving Requirements on System Design – Capability of the UML to Cope with Change

It is well accepted that evolving requirements are one of the main causes of the failure of software projects[2]. A most important example of changing requirements is the case where the requirements were poorly engineered in the first place, where either key requirements were left out or misunderstood, or the main users or stakeholders were not involved in the requirements process. This is well described by Martin Fowler[5].

The project requirements may be classified as “fairly static” — that is to say, from the outset of the project it was known that once a requirement had been identified and described in detail then it was thought unlikely to change. However, the first major change came during the period that the user requirements were being initially “closed-out”. It was found that an extra layer of complexity had to be introduced into the way that the users could select and view data. This change was incorporated immediately into the functional specification. As little analysis or design work had been performed, the change had no impact for system design. The next change that occurred was a few weeks after most of the analysis had been performed. The database that the application had been intended for — Oracle — was changed to Microsoft SQL. The physical change of the database was not going to happen for six to eight months. At the time of the change no implementation logic or SQL database query commands had been generated. While this was a major change, with respect to some of the supporting systems, it had little impact for the application. This is an example of the usefulness of the BCED approach, which had been facilitated in the design by the use of UML. It shows the importance of the object-oriented capability that UML gives designers, along with the fact that design and analysis patterns such as the BCED approach may translate directly into a UML design. For the purposes of the problem domain that the project resided in, it was found that UML was rich enough to allow a system design to be implemented that was robust to small changes such as those described.

It is accepted that a system with more dynamic requirements would be needed to properly assess the capability of UML to cope with change. Given that there was still change present in a project with requirements that may be classified as “static” provides anecdotal evidence of the types of problems that may arise and how improved attention at the design stage can save time and effort. With regard to the process of technology transfer, we found it difficult to draw any reasonable conclusions about the succesful transfer, into an industrial environment, of the student’s knowledge of, and experience with, evolving requirements.

5 Conclusions and Recommendations

5.1 UML

In the experience of the developer the most positive aspect of using UML in this manner was that it provided a common framework for communication of requirements. The process of modelling helps to focus the discussion on particular aspects of the system and ensures everyone involved is “together”. A positive side of using the Use Case dia-

grams was that they enabled a clear set of high level functional requirements to be agreed upon.

The ambiguous aspects of the Use Case diagrams contributed to the developer seeking an alternative approach to the Use Case driven approach to assist the move from requirements to analysis and design. It was found that there was no general method for performing this transition. The key to making a successful transition may be the experience of the developer. For analysis it was found useful to look at currently existing solutions (such as the BCED) approach rather than "re-inventing the wheel". The use of Sequence and Collaboration diagrams helped us to explore how the system would actually behave, as well as how the program design may work from a high level.

Given that the requirements of the project were generally static, it is a point of interest that there was still a degree of change to the requirements as discussed in section five. This is in line with the wider view that change to requirements is generally underestimated. In the experience of the developer this notion of requirements evolution and the need for an adaptive process is not limited to software development and may be found in all manners of projects right throughout engineering. As previously stated the project in question was not a stringent test of the capability of the UML to cope with change.

As an overall evaluation from the point of view of a beginner responsible for technology transfer, UML was found to be difficult to apply in a manner that best suited the context of the problem. A common language (like the UML) is a necessary but not sufficient requirement for good design process involving teams of developers. UML does not provide any special assistance in moving across the spectrum of development. Consequently, we argue that the step from theory to practice — in how the wide-spectrum nature of the language can be more rigorously managed and exploited — would be better taken with the guidance of an expert.

5.2 Design and Technology Transfer

Academic courses dealing with software design need to be more problem-oriented and less language-oriented. Learning the UML is not the same as learning how to design software (using the UML). Recent graduates have a responsibility to transfer their knowledge of the new theory underpinning design as a process, rather than just to demonstrate their mastery of a specific design language, model or tools. Our small project demonstrates that such transfer can take place: however, it is not easy and it is unrealistic to expect the "average graduate" to play this important role.

Acknowledgements

Special thanks to Bill Andrews for his role as the domain expert.

References

[1] Bennet. *Object Oriented Systems Analysis and Design using UML*. McGraw Hill, 1999.

[2] Barry W. Boehm. Software risk management: Principles and practices. *IEEE Software*, 8(1):32–41, January 1991.

[3] Grady Booch. *The UML User Guide*. Addison Wesley, 1999.

[4] G.E. Feletti D. Boud. *The challenge of problem-based learning*. Routledge Falmer, 1998.

[5] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley, 2003.

[6] J.Paul Gibson. *Formal Object Oriented Development of Software Systems Using LOTOS*. Thesis csm-114, Stirling University, August 1993.

[7] Paul Gibson. Formal requirements engineering: Learning from the students. In *Australian Software Engineering Conference 2000 (ASWEC00)*, Canberra, Australia, April 2000. IEEE.

[8] L. A. Maciazek. *Requirements Analysis and System Design — Developing Information Systems with UML*. Addison Wesley, 2001.

[9] Ewan Tempero Robert Biddle, James Noble. Reflections on crc cards and oo design. In *Proceedings of the Fortieth International Confernece on Tools Pacific: Objects for internet, mobile and embedded applications*, volume 10. ACM, 2002.

[10] J. Rumbaugh. *The UML Reference Manual*. Addison Wesley, 1999.