

2006

Engineering Enterprise Software Systems with Interactive UML Models and Aspect-Oriented Middleware

Paul Nguyen
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_projects



Part of the [Computer Sciences Commons](#)

Recommended Citation

Nguyen, Paul, "Engineering Enterprise Software Systems with Interactive UML Models and Aspect-Oriented Middleware" (2006). *Master's Projects*. 124.

DOI: <https://doi.org/10.31979/etd.w5h3-m7b2>

https://scholarworks.sjsu.edu/etd_projects/124

This Master's Project is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Projects by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

Engineering Enterprise Software Systems with Interactive
UML Models and Aspect-Oriented Middleware

A Writing Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Paul H. Nguyen

Copyright © 2006

Paul H. Nguyen

All Rights Reserved

Abstract

Large scale enterprise software systems are inherently complex and hard to maintain. To deal with this complexity, current mainstream software engineering practices aim at raising the level of abstraction to visual models described in OMG's UML modeling language. Current UML tools, however, produce static design diagrams for documentation which quickly become out-of-sync with the software, and thus obsolete. To address this issue, current model-driven software development approaches aim at software automation using generators that translate models into code. However, these solutions don't have a good answer for dealing with legacy source code and the evolution of existing enterprise software systems.

This research investigates an alternative solution by making the process of modeling more interactive with a simulator and integrating simulation with the live software system. Such an approach supports model-driven development at a higher-level of abstraction with models without sacrificing the need to drop into a lower-level with code. Additionally, simulation also supports better evolution since the impact of a change to a particular area of existing software can be better understood using simulated "what-if" scenarios. This project proposes such a solution by developing a web-based UML simulator for modeling use cases and sequence diagrams and integrating the simulator with existing applications using aspect-oriented middleware technology.

Acknowledgements

I would to thank my research advisor, Dr. Robert Chun, for encouraging me to follow my dreams and providing the support and coaching to help me get there. In 1990, I graduated from San Jose State from the Computer Science department with the knowledge I needed to carry my career in the industry for ten years. In 2000, I came back to SJSU, seeking a Masters to prepare for the next ten. Standing here now, six years since the start of that journey, and looking back; I am confident that this work will help me reach that goal. But, to my pleasant surprise, I ended up with more than I could have imagined. Dr. Chun, you have taught me how to make the best of my talents, how to focus my research and let creativity drive the work, and most importantly, how to make contributions back to into the field. This, I expect, will live with me beyond that ten year goal I had originally projected. It will live with me for the rest of my life. And for this, I am forever grateful.

I would also like to thank Dr. Lee Chang and Dr. Suneuy Kim for reviewing my work and providing valuable feedback. In addition, I am grateful to Dr. Chang for the intense but worthwhile semester with UML, Patterns, and Refactoring in the CMPE 221 course – the result of which formed some of the motivation for my research.

And last but not least, no person can make a journey such as this without strong family support. To my beautiful wife, Mai-Tram, for your sacrifices those last six years; and, to my wonderful two children, Audrey and Pascal, who have been wondering why their dad always stays up late at night...

To you, I dedicate this work.

This journey was a family effort, and I could not have made it here without your love and support.

To my advisors and family, many thanks again for your support.

--Paul

Table of Contents

1.	INTRODUCTION.....	6
1.1	TOWARDS MODEL-DRIVEN SOFTWARE DEVELOPMENT	6
1.2	CHALLENGES IN THE SOFTWARE INDUSTRY	7
1.3	THE EVOLUTION OF ABSTRACTION: A BRIEF HISTORY.....	9
2.	CURRENT STATE-OF-THE ART	16
2.1	MODEL-DRIVEN DEVELOPMENT & SOFTWARE FACTORIES	16
2.2	NAKED OBJECTS & BABYUML.....	18
2.3	ASPECT-ORIENTED PROGRAMMING.....	20
3.	MOTIVATION.....	22
3.1	ENTERPRISE SOFTWARE	22
3.2	KEY PROBLEMS	23
3.3	EXTRACTING SYSTEM MODELS: A PRACTICAL APPROACH TO DEALING WITH TODAY’S PROBLEMS.....	25
3.4	ACTIVE MODELS: HOPE FOR THE FUTURE.....	29
4.	TOWARDS INTEGRATED ACTIVE SYSTEMS.....	33
4.1	TRADITIONAL SOFTWARE DEVELOPMENT & MODEL-DRIVEN ARCHITECTURE (MDA).....	33
4.2	DOMAIN-SPECIFIC SOFTWARE DEVELOPMENT	34
4.3	METATOOLS FOR DOMAIN-SPECIFIC LANGUAGES.....	35
4.4	SOFTWARE DEVELOPMENT WITH ACTIVE SYSTEMS	36
5.	THE EXPERIMENT	37
5.1	SCOPE OF EXPERIMENT & INVESTIGATION	38
5.2	THE METAMODEL	39
5.3	A DOMAIN SPECIFIC LANGUAGE FOR DESCRIBING USE CASES.....	40
5.4	JBOSS AOP	40
5.5	WEB INTERFACE FOR INTERACTIVE UML DIAGRAMS.....	40
6.	EXPERIMENTAL PLATFORM AND RESEARCH PROTOTYPE	41
6.1	OVERVIEW OF ARCHITECTURE	41
6.2	DSL LANGUAGE DESIGN & IMPLEMENTATION	42
6.3	DIAGRAMS AND DOCUMENT GENERATION	52
6.4	MODELING & SIMULATION.....	56
7.	CASE STUDIES	60
7.1	OVERVIEW OF CASE STUDIES	60
7.2	CASE STUDY A: ACTIVE USE CASE DOCUMENTS.....	61
7.3	CASE STUDY B: BLACK BOX SYSTEMS INTEGRATION VIA WEB SERVICES.....	71
7.4	CASE STUDY C: REFACTORING DATABASE ACCESS CODE TO THE HIBERNATE FRAMEWORK	75
8.	ANALYSIS	89
9.	CONCLUSION.....	93
10.	RELATED AND FUTURE WORK.....	95
11.	APPENDICES	97
11.1	SAMPLE UI SCREENS FROM THE RESEARCH PROTOTYPE.....	97
11.2	METAMODEL	103
11.3	DSL SYNTAX AND EXAMPLES.....	104
11.4	SOFTWARE TOOLS AND DEVELOPMENT FRAMEWORKS USED.....	107
12.	TABLES AND FIGURES.....	108
13.	REFERENCES.....	110

1. Introduction

1.1 Towards Model-Driven Software Development

Over the past two decades, the software industry's strive towards the goal of engineering software based on reusable components has been met with many challenges. Although much progress has been made with the introduction of C++ in the late 80's and the mainstream adoption of object-orientation with Java during the 90's, software reuse today still falls short of our expectations. An evolutionary new technology has been gaining popularity in recent years bringing new promise to software reuse. This new paradigm known as Aspect-Oriented Programming (AOP) addresses the modularization of "cross-cutting concerns" which have eluded solutions by current object-oriented techniques. Initial work in AOP research has been focused on programming language features lead by Xerox PARC's AspectJ programming language. But recently, AOP's impact has spread into other areas of software with some notable developments which include: integrating aspects in middleware, applying aspects in analysis and design methods, and leveraging aspects in generative application frameworks.

Perhaps, one of the most promising areas of synergy for AOP is in model-driven development. Two competing approaches aim to take the future of software development on different paths. Model-Driven Architecture (MDA™) from the OMG takes a top-down approach focusing on evolving UML to a full-fledged general purpose programming language supported by tools that generate code from UML. On the opposite end is Software Factories, which combines a number of best practices including software product lines and domain-specific models. In stark contrast to MDA, Software Factories focuses on building reusable domain-specific frameworks from the ground up and providing meta-tools

to help create specialized modeling environments and domain-specific programming languages which target this framework.

1.2 Challenges in the Software Industry

There is no single development, in either technology or management technique, which by itself promises even one order-of-magnitude improvement within a decade in productivity, in reliability, in simplicity.

-- Frederick Brooks

In 1986, Frederick Brooks published an IFIPS paper titled “No Silver Bullet”, which was later republished in IEEE Computer Magazine in 1987 [18]. In the article, he posed a challenge to the software engineering industry to disprove his prediction that in 10 years, no new programming paradigm or technique could bring even one order of magnitude of improvement in productivity. The main driving force behind’s Brook’s predication is the realization that software is complex by nature and that improvements can only be made by stepwise and persistent progress through evolution. Brooks further explains that there exists a promising body of work which “attack the conceptual essence” of software complexity. These include: software reuse via buy vs. build, requirements refinement via rapid prototyping, organically growing software via incremental development, and a focus on “people” by the cultivation of great conceptual designers. Much like “Moore’s Law” in hardware engineering that has thus far stood ground with the test of time, Brooks’ prediction held true during that decade. But, what about the decade that followed? And, how is software engineering today in 2006?

In the 1995 reprint of “The Mythical Man-Month”, Brooks included “No Silver Bullet” as an addendum and additional chapters on selected opinions and responses to his original paper [19]. The report reconfirmed Brooks predictions citing major problems with software reuse due in part to business organizational issues, lack of incentive discouraging investment in reuse of object-oriented components,

poor documentation, and the advent of generic system software (such as the database management system) minimizing the need to reuse in the application code.

As for the question of where Software is today, Jason Bloomberg's article on Web Services and Service-Oriented Architecture titled "Software's Dirty Little Secret" [20] summarizes the current status quo in three points. First, compared to the "high-tech" hardware industry where general purpose computers are built on the "meta-requirement" of programmability, software is very "low-tech". That is, software is currently built for a very specific purpose and remains very much a craft-based industry. Second, Bloomberg further points out that commercial off-the-shelf (COTS) software packages are inflexible requiring users to adapt their behavior and work to the limitations of the software. Third, Bloomberg argues for a redefinition of "Software Quality" different from those put forth by Six Sigma and ISO 9001. Bloomberg writes:

We must take a step back, so that we can judge software quality based upon its flexibility and agility, rather than how few defects it has, or how well-documented the process of creating it might be.

Thus, like "Moore's Law", Brooks' prediction remains unchallenged, even a decade beyond his original deadline. The software industry, as a whole, is then left then to deal with complexity in progressive steps, resolving with the conclusion that there will never be a "Silver Bullet".¹

If each technology or technique alone can not deliver at least a 10 fold improvement within 10 years, perhaps as Brooks suggests, the answer lies in a multi-disciplinary, multi-paradigm approach. What then, are the promising emerging technologies of the current decade that holds promise to propel the software industry forward in productivity? Is there synergy? What are some of the integration

¹ Every once in a while, a new technology always comes along claiming to be the next "Silver Bullet".

challenges? In section 3, this work explores this further in three promising research areas: Model-Driven Development, Software Product Lines, and Aspect-Oriented Programming. But, before looking ahead, the next section discusses the progress made to date and reflect on past challenges and lessons learned.

1.3 The Evolution of Abstraction: A Brief History

Since the beginning of computing, programming languages have been an indispensable tool in the battle against complexity. As the problem space presents itself through experience, language designers built abstractions into languages from lessons learned, steadily marching closer to the problem domain, and creating the tools to aid in the solution to ever harder problems. The general trend has been focused on abstractions close to the hardware and computing environment fueled by the exponential grow of computing power, faster networks, and global reach to end users. The following discussion presents a summary of findings from Schorsch & Cook's report in the *Journal of Defense Software Engineering* titled "Evolutionary Trends of Programming Languages" [21] in the context of other notable developments and progress in the software industry.

The major trends identified in [21] will be discussed in the context of developments in computing hardware, and progress in system software and end-user interaction (i.e. GUI's and the Internet). The points of discussion are along the lines of:

1. What problem was solved?
2. What was the level of abstraction introduced?
3. How did those abstractions relate to other developments?
4. What were some examples?

1.3.1 Evolutionary Trends in Programming Languages

1.3.1.1 *Machine-Independent Programming*

The first generation languages were expressed in machine language in a form that can be directly executed. This forced the programmer to work in the language of the hardware close to the instruction set of the Central Processing Unit (CPU). As the hardware evolved, however, programs must be rewritten in the new CPU instruction set. Later evolutions to second and third generation languages progressed further from the hardware to free software from the confines of the computing device. The second generation high-level assembly languages raised the bar to symbolic machine instructions which was later followed by progress in third generation languages which abstracted away the CPU instruction set.

Table 1. Summary of Contributions from Machine-Independent Programming

Challenge Problem	Contribution	Examples	Context
Invent a programmable computer.	<ul style="list-style-type: none"> • General purpose computing machine. • Turing Complete • Von Neumann Architecture 	ENIAC, EDSAC, IBM 701	First generation low-level machine code for machines of the 1940's and 1950's.
Program in "Symbols", rather than bits.	<ul style="list-style-type: none"> • High-Level "Symbolic" Machine Language. 	IBM S/360	Second generation assembler code of 1960's.
Make programs portable across hardware.	<ul style="list-style-type: none"> • Evolution of control structures: Fortran "Goto", structured programming (Algol), case statements, generalized loops, tasks and co-routines, exception handling, parallel programming • Data structures: floating-point, logical data types (i.e. chars, strings, booleans, arrays, records, abstract data types, etc... 	Fortran, COBOL, Algol, Ada, Pascal, C, etc...	Third generation, high-level languages from 1950's to current time.

1.3.1.2 *The Rise of Virtual Machines*

Before the advent of the modern virtual machines as exemplified in the Java JVM and Microsoft Dot-Net CLR, high-level languages developed along two parallel paths of the “interpreter” vs. the “compiler”. The primary reasons for this split were due to the problems the languages were designed to solve which influenced the trade-off between expressiveness of abstractions vs. performance and complexity of compiler design. Early versions of interpreted languages focused on different computational models. For example, Lisp explored the Functional, and Prolog explored the Logic computational models. It is interesting to note, that these two paths have essentially converged with the creation the modern virtual machine and just-in-time compilers such as the Java JVM/JIT. The invention of the Java JVM, was a huge leap in machine independence, raising the bar not just above hardware, but also the operating system software.

Table 2. Summary of Contributions of Virtual Machines

Challenge Problem	Contribution	Examples	Context
Abstract away the operating system, programming language, and/or computational model.	<ul style="list-style-type: none"> • Intermediate machine language • Machine and OS Independence (Java) • Language Independence (Dot-Net CLR) • Alternative Computational Models (Lisp, Prolog) 	Lisp, Prolog, Pascal P-Code, Java JVM, Dot-Net CLR	A long history from the 1960’s to date.

1.3.1.3 *Programming Language Interoperability & Domain-Specific Languages*

High-Level programming languages have emerged (as previously discussed) along two evolutionary paths: those that are typically interpreted and closer to a problem domain, and those that are typically compiled to native assembly code. In general, the languages closer to the problem domain

are also known as Domain-Specific Languages (DSL), while those that are more optimized for hardware and focus on solving general problems are also classified as General-Purpose Languages (GPL). This gives rise to the issue of interoperability amongst the languages. In the ideal world, a developer should be able to pick the best tool for the problem. To make this possible, solutions tackle the problem from two angles: 1). Interoperate amongst GPL's, and 2). Integrate DSL's with GPL's.

Table 3. Summary of Contributions from Programming Language Interoperability & Domain-Specific Languages

Challenge Problem	Contribution	Examples	Context
Interoperability amongst GPL's.	<ol style="list-style-type: none"> 1. Calls to external libraries in a different language 2. External data exchange 3. Share code libraries across languages 4. Shared Classes and Objects 	<ol style="list-style-type: none"> 1. Most have some capability, but data exchange problematic. Ada does a better job. 2. Machine-Independent data standards (EDI, XML) 3. DLL, COM, CORBA, Web Services (i.e. SOAP/SOA) 4. Dot-Net CLR 	The age of 3GL's and beyond.
Integration of DSL's with GPL's.	<ol style="list-style-type: none"> 1. "Glue" Scripting Languages 2. DSL Embedded in GPL 3. Scripting Languages used by software or frameworks developed in a GPL 	<ol style="list-style-type: none"> 1. Perl incorporating features of sh, sed, & awk. 2. TCL in C, Embedded SQL, JavaScript in Java 3. JavaScript, HTML, XML, etc... for web frameworks, GUI components, configuration, data exchange, etc... 	Significant developments during the 1980's and 1990's.

1.3.1.4 Increasing Modularity

Breaking up a complex problem into smaller easier to solve problems is a common trait across engineering fields. The initial modular units in programming languages were functional and data groupings. This later evolved into encapsulation and information hiding of object-oriented languages, then to object-oriented frameworks. Current active research in aspect-oriented programming introduces

an additional modular unit orthogonal to objects called Aspects. And, work in software architecture focuses on coarse-grain modules that make up whole systems or platforms.

Table 4. Summary of Contributions from Increasing Modularity

Challenge Problem	Contribution	Examples	Context
Decompose the problem into smaller parts.	<ol style="list-style-type: none"> 1. Procedures, Functions, User defined data types 2. Objects 3. Frameworks 4. Platforms 	<ol style="list-style-type: none"> 1. starting with pre-OO languages like Pascal, and Ada, C, etc... 2. Simula, Smalltalk, C++, Java, C#, etc... 3. GUI Frameworks. I.E. MFC, Swing. 4. J2EE and the Java Application Server 	1970's to date, with leaps during the OO era and currently with AOP.

1.3.2 Other Developments

1.3.2.1 Modeling Languages

The Unified Modeling Language (UML) from the Object Management Group (OMG) is a standard modeling language with its roots in data modeling, object-oriented languages, and a number of other modeling disciplines including business process and real-time event driven modeling. A major contribution of the language is a visual notation that has proven to be a common language amongst designers. It is difficult today to pick up a software design book without coming across a UML diagram. This gave rise to the documentation and knowledge distribution of reusable object-oriented designs in the form of patterns popularized by the “Design Patterns” book [22]. However, UML is not without its drawbacks. Although it is a standard modeling notation, UML is a general purpose design language and requires extensions to support concepts of a specific problem domain. Furthermore, although UML incorporates data modeling concepts, the UML tools in the marketplace have poor support for database modeling. This is due primarily to the maturity of the data modeling tools and the

impedance mismatch between the concepts of objects and that of the relational model. As such, the general use of UML is focused on a core subset with a complementary set of additional modeling practices. Some, however, avoid UML completely. This is understandable, since, not all developers find visual notations useful preferring instead to use other techniques to capture and understand the requirements [23].

1.3.2.2 Unix, Linux, and Open Source

The invention of Unix during the 1970's which coincided with the development of the C language brought great improvements in productivity to programming in many ways, namely, hardware portability with good performance and shell scripting with many reusable "little languages"². But, perhaps, Unix's great contribution is yet to come in its later variant in "Linux". Created in the early 1990's, Linux targeted the emerging low-cost personal IBM computers. With the introduction of the web browser in the mid 1990's and the wide availability of the internet, Linux enabled a new collaborative network centric software development movement known as "Open Source". The significant contribution of this phenomenon is a disruptive form of software reuse and a different business model for selling software. With the ERP application space largely a failure, new efforts in Open Source Applications in the ERP space (SurgarCRM) hold great promise for the rise of commodity software, and hence a potential for mass reuse. This model seems to have already started taking hold, for example, in software development frameworks and integrated development tools, such as those from the Apache and Eclipse Foundations.

² Unix's shell environment incorporates many small DSL's which can easily be composed together into many variety of shell programs.

Some of these, for example, include text processing, line editors, and search tools.

1.3.2.3 Co-Evolution of the Application Server and Database Technology

The creation of the Relational Database Management System (RDBMS) based on a research paper in 1970 by an IBM researcher named Ted Codd liberated programmers from the details of data access and management. A key contribution was in the specification of a declarative query language (SQL) that was simple for both end users and developers alike. The declarative nature of SQL abstracted away the navigational details of data retrieval and allocated the responsibility of data management and query optimization to the database management system. SQL, as a DSL, used in both GUI query tools and embedded in GPL's (like C, C++, and Java) is perhaps one of the most successful DSL/GPL integration to date. The RDBMS was founded on strong mathematical set theory and relational algebra, a model that has stood the test of time for business applications to this day. During the 1990's, however, other models challenged the relational model. The need to store and manage objects in a convenient way from within the popular object-oriented language of time (C++) gave rise to the creation of the ODBMS. There were many heated debates during this time on all fronts from vendors to practitioners. While the ODBMS vendors were hard at work on standards, the RDBMS vendors pushed a hybrid-approach and eventually won out over the ODBMS [24]. As a result, in 2001, the ODBMS standards group disbanded. On the practitioners' front, a huge cultural divide in design methodologies caused difficulties on projects and contributed to poor team dynamics and project delays [25]. In the end, both technologies co-evolved and influenced each other in possible ways. With the rise of the Java Application server and the J2EE Frameworks, a number of innovative object-relational mapping approaches were invented to deal with the relational impedance mismatch. In the RDBMS, the SQL standard evolved to support objects which provided query language support and integrated both the object and relational models. A new challenger came into the picture in the late 1990's to early 2000's; the native XML database. Unlike the ODBMS vendors, however, XML had a strong standards body

which was quick to adopt XPath and then later XQuery and other related specifications. Furthermore, the adoption of XML in applications focused on the web presentation, data exchange, and framework configuration. Thus, XML was not at all a real threat to the RDBMS space. Nevertheless, recent releases of major RDBMS solutions have incorporated the XML model and specifications (included query languages) into their products.

1.3.3 Summary and Current Challenges

Programming languages exist to deal with complexity. Over the past 20 years, the main focus has been to raise the level of abstraction, starting with building blocks close to the hardware in the 50's to objects and frameworks of the 90's and the current times. However, as object-oriented frameworks have evolved, so has their size in number of modules and classes. Today's Java J2EE Framework, for example, handle a gamut of services, from web presentation, to persistence, email, security, messaging, management, XML processing, and emerging web services technology – to name a few. Likewise, the Microsoft Dot-Net Framework mirrors the feature set from J2EE, but is still very young and rapidly changing. As a consequence, application developers are faced with a huge library of components that are constantly evolving. The advent of integrated software development tools (the IDE) have helped to a limited degree with context sensitive online help, assistance with coding, and framework code generation; but, there are too many tools, concern over tool-framework lock-in, and ease-of-use issues.

2. Current State-of-the Art

2.1 Model-Driven Development & Software Factories

The industry is currently seeking simplification by forging yet another attack on complexity on many fronts. Building on its widely popular modeling language (UML), the OMG's Model-Driven

Architecture (MDA) proposes to abstract away from current programming languages and application platform specifics into UML based models. In [26]:

The MDA separates certain key models of a system, and brings a consistent structure to these models. ...models of different systems are structured explicitly into Platform Independent Models (PIMs), and Platform Specific Models (PSMs). How the functionality specified in a PIM is realized is specified in a platform-specific way in the PSM, which is derived from the PIM via some transformation.

This automatic transformation of UML models into executable code is assisted by a tool that can be considered a high-level compiler. In a nutshell, OMG aims to evolve UML's documentation centric usage to a full-fledged programming language. This vision implies the end of programming languages as we know it! As to be expected, the announcement is seen as a claim to be the new "Silver Bullet", while at the same time evoking bad memories of the CASE tools genre in the 80's. There are strong arguments against MDA. Martin Fowler and Scott Cook cites fragmentation in the MDA community itself with three approaches to MDA [27], furthermore, Fowler writes that there is a more pragmatic Model Driven Development approach (MDD) often confused with MDA. The MDD folks shun UML but adopt a number of current best practices in software development; namely, software product lines, frameworks, patterns, and agile practices. One such approach receiving much attention recently is Microsoft's "Software Factories". Rather than use UML, Software Factories [16] aims to create meta-tools that generate domain-specific languages (graphical or textual) to capture high-level concepts. These DSL are then supported by specialized editing, debugging, and build-by-assembly tools reminiscent of software product line approaches. Software Factories also emphasize the creation of a domain-specific application framework to which the DSL and tools are targeted for code generation. Both approaches recognized the need to capture domain knowledge in a machine re-usable form; however, they differ in philosophy. MDA sees the need to model up-front and abstract the process of

transformation to code into standard specifications for tools vendors. Software Factories starts from existing applications in the domain, extracts out a common framework, and builds custom tools to support this framework. This is essentially the classic top-down vs. bottom-up design debate. Figure 1. below depicts the evolution and influences on these two techniques.

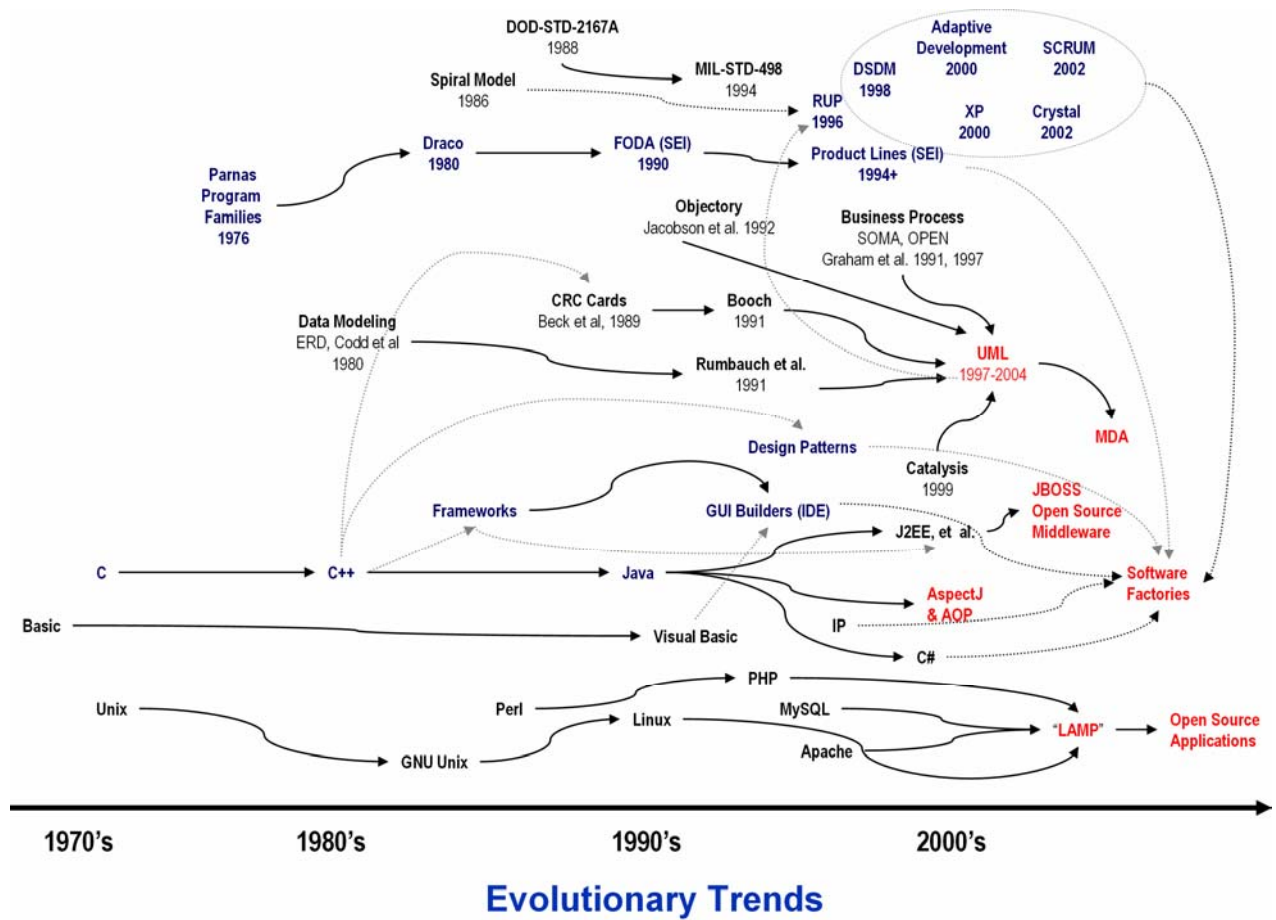


Figure 1. Influences on MDA and Software Factories

2.2 Naked Objects & BabyUML

In academia, two notable ongoing researches take a refreshing retro look back to the roots of object-orientation. Richard Pawson’s PhD Thesis on “Naked Objects” [28] emphasizes domain objects as behaviorally complete entities interacting directly with users through standardized automatically

generated user-interfaces. This is a contrast to the current practice of class-centric object-oriented designs. Pawson's Naked Object system modernizes the original object-oriented design principles from Simula and Smalltalk. In [28], Pawson writes:

The inventors of Simula had the idea of building systems out of 'objects'. Each software object not only knows the properties or attributes of the real-world entity that it represents, but also knows how to model the behaviour of that entity...

In the original work, each object was seen as being self-contained... - the attributes of an object were encapsulated with all the necessary behaviours.

Pawson makes a strong argument against the current popular practice of use-case driven approaches and the model-view-controller pattern. This objects-first thinking is apparent in the Naked Objects Framework and User Interface where the "Object" is the center of attention and the "Class" takes a back seat. Although Pawson does not emphasize the use of UML, his work has been shown to be complementary with the use of current round-trip UML tools, such as Together Control Center [29].

Trygve Reenskaug takes the Naked Objects approach one step further by specifically integrating UML and Web Services into the architecture and focuses on higher-level of abstractions at the level of components in his ongoing work with "BabyUML" [30][31]. In Reenskaug's ECOOP 2004 presentation, he discusses the background and inspiration behind BabyUML [30]. These inspirations included: Engelbart's "Augmenting the Human Intellect", Pawson's Naked Objects, Shaw's Basic English, and current industrial technologies such as distributed components, web services, and UML. In essence, Reenskaug is attempting to blend the "old" with the "new". Figure 2. below shows the influences on BabyUML.

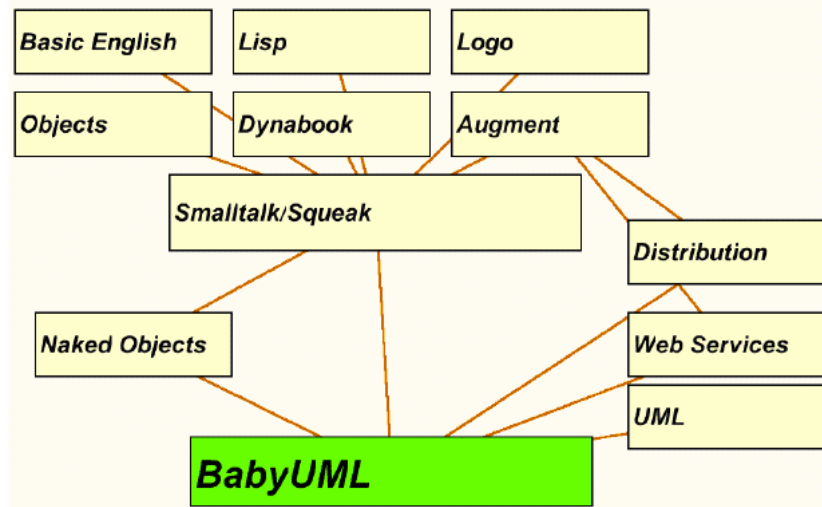


Figure 2. Influences on BabyUML (reproduced from [30])

OMG’s MDA, Microsoft’s Software Factories, Naked Objects, and BabyUML all share one common theme – the focus on “models” as an important artifact in software development. OMG makes models the dominant abstraction, Software Factories blends domain modeling with other current best practices, and leading edge research in Naked Objects and BabyUML takes us back to OOP origins putting the user at the center of control of objects.

2.3 Aspect-Oriented Programming

In other developments, evolving separately from model driven development is Aspect-Oriented Programming (AOP) [33]. AOP is based on the principle of separation of concerns [32], and introduces a new form of modularity (called “aspects”) orthogonal to and complementary with objects. Aspects can encapsulate cross-cutting concerns that are currently redundant in object-oriented systems. Since objects focus on encapsulation and modeling behavior of real world entities they represent, system level concerns such as persistence, transactions, security, and concurrency tend to be intertwined with objects making reuse difficult. Mik Kersten’s presentation at OOPSLA 2002, shows a diagram highlighting “tangling of logging code” in red code in the Tomcat servlet engine (Figure 3. below).

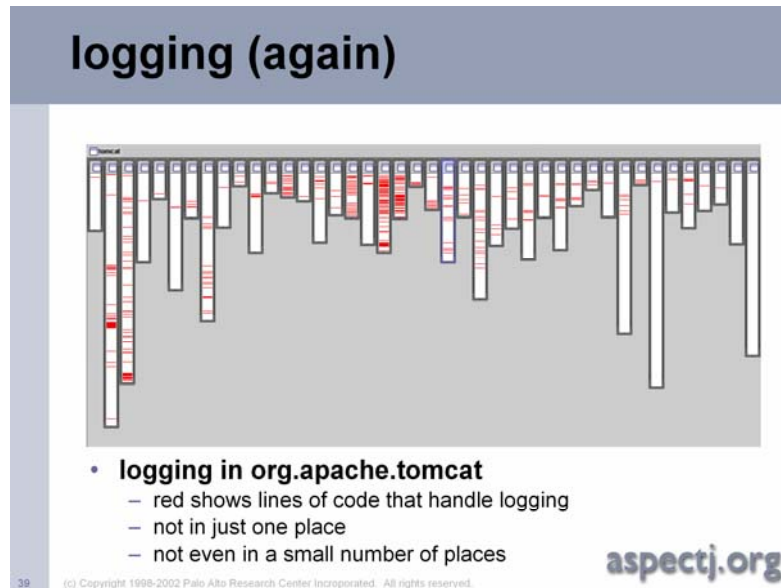


Figure 3. Logging not modularized in Tomcat (reproduced from [34])

The main reason for this phenomenon is due to the lack of expressiveness in current object-oriented programming languages. Cristina V. Lopes, one of the inventors of AspectJ, writes [33]:

Programming languages support a very small set of referential relations. In particular, reflective references, groups and temporal references are, practically, inexistent. They can be simulated by combinations of computation and new nouns. And that's exactly one of the things that make programs much more complex than they should be: programmers have to express a rich set of referencing forms using a very small set of referencing forms. In the process, intentions get diluted and tangled.

Since late 1990's, there has been steadily rising interest in aspects both in academia and in industry. A body of work is mounting, expanding the concept of "Aspects" across the landscape of software engineering. Areas of research include: architecture [36][35], requirements analysis and design [36][37][38], cross pollination with use cases [39][3][41][40], integration with software product lines [42], and others. Interest in the open source community is also widespread. There are numerous projects focused on extending current languages with aspects features [76], and aspects have also made

their way into frameworks, databases, and tools [45][47][43][44][46][75]. The full impact of the aspect movement has yet to be realized, since they have not taken hold in large scale enterprise software. However, with new developments in middleware (JBoss 4.0 & JBoss AOP), the first step has been taken.

3. Motivation

3.1 Enterprise Software

Integration has been the holy grail of MIS since the early days of computing in organizations. As early as 1969 Blumenthal proposed an integrated architecture and a framework for organizational information systems. However, due to the high level of organizational and technical complexity associated with their development and implementation, integrated enterprise-wide systems have been difficult to achieve in practice.

-- Kumar & Hillegersberg, "ERP experiences and evolution" [48]

Large scale business software systems as embodied in Enterprise Resource Planning (ERP) software are amongst the most complex software systems currently in use. Enterprise software is typically delivered to large businesses today from a handful of vendors often with a multitude of options and configurations in multi-module application suites. Businesses are diverse and constantly evolving to stay competitive; often, making bold moves which include: acquisitions, internal reorganizations, or inventing new business models to drive growth. Information systems supported by Enterprise Software have failed to keep up with the pace of business changes --a condition widely acknowledged in industry evoking a "cry for change" with new Business/IT alignment initiatives. Although there are business and organizational issues at play within user organizations, the vendor-customer model is also at fault. ERP's are supplied by only a few vendors and the law of "supply-demand" behind the production of software necessitates creating a highly customizable "generic" product. That is, software is made for "mass customizations", not "mass production". In [48], Kumar and Hillergersberg points out that:

A key premise of ERP systems is the underlying, sometimes unstated, but often implicitly promoted notion that the reference models in ERP systems embody best business practices...

While at the abstract level the idea of “universal” best practices may be seductive, at the detailed process level these mismatches create considerable implementation and adaptation problems.

“The Reference Models in ERP Systems” -- That is, the “domain model” embedded in the software -- is too abstract! The solution to enterprise systems calls for a component based industry where more “specialized” domain-specific models can be bought or built and integrated through standard interfaces. This has been the focus in recent years with the development of horizontal application “super platforms” and loosely-coupled integration standards, such as web services. However, to date, there are still many challenges.

3.2 Key Problems

My own experience integrating and customizing enterprise systems, and the numerous issues (as cited below) form the motivation for this research:

1. **Evolution:** The evolution of enterprise software solutions implemented with a mix of packaged software, home-grown systems, and legacy interfaces poses a challenge for upgrades. The full impact of such changes is hard to determine because they are not well isolated. As a result, solutions implemented often require a large migration effort or become obsolete by replacement.
2. **Incompatibility:** While the technology interoperability issues are well understood and can be easily studied and corrected, the incompatibility in the information models amongst various applications developed by different vendors have not been well addressed. Generic data exchange standards exist, but domain-specific issues are still unresolved. One of the most difficult problems to solve is the problem of the “dynamic domain model”. A dynamic domain

model is the evolving concepts and business rules represented in the business software itself. Such models evolved due to internal changes, new software releases from multiple vendors, and/or high customization within the user organization.

3. **Requirements Mismatch:** Due to the inflexibility of most software packages, typical implementations are often done by forcing the business and users to adapt to the software. This, effectively forces the vendor's view of the domain model on the business. Unfortunately, users find other ways to work around the inflexibility by reverting to personal productivity tools. This is a major problem because this makes portions of the "dynamic domain model" of the business inaccessible. The problem compounds with each new software release and/or package added to the mix.
4. **Documentation Centric:** In an effort to capture and understand the "dynamic model" of the business, enterprise software implementation projects produce a massive amount of documentation. Even for model driven teams that fully embrace RUP and UML, the difficulty and cost in time and effort in updating the documents are often too high. IT resources are limited and often reallocated to new projects once a solution has been deployed. Furthermore, there are physical limits in the medium. Comprehensive UML models just don't fit on a standard page. At best, if the documents are maintained diligently, they only provide a partial view of the system.
5. **Lack of Isolation between Logical and Physical:** Configuration management is challenging in today's enterprise systems. The packaging and deployment model of Java Application Servers is the most troublesome area. Such deployments are typically packaged in a single file, which actually contains other packages nested within. This makes the package tightly bounded to the environment configuration, and as a consequence, any change in the environment or the package

can cause instability. There needs to be a seamless separation between the “logical” (i.e the application) and the “physical” (i.e. the hardware and network configuration).

6. **The Asynchronous Nature of Distributed Teams:** The nature of today’s large scale software development teams imposes yet another difficult dimension to the problem. Best practice prescribes continuous integration and automated tools, but sometimes this is not possible. Package solutions implementation often requires vendors to make changes and distribute them as patches and minor releases to the customer during implementation. Unfortunately, this is often done in isolation.

3.3 Extracting System Models: A Practical Approach to dealing with today’s problems.

To mitigate the risks and address some of the key problems discussed, it follows from agile practices to focus on the working software and strive to raise the level of understanding amongst the team. This can be done in part using reverse engineering tools to extract models. However, the tools that exist today extract only the “static” model. For example, Erwin can generate physical database models from metadata within the database, and Together can parse Java code and archives to generate class and static sequence diagrams. Extracting the “dynamic model”; however, is difficult. And, in the case of J2EE non-existent!

For example, it is possible to extract some dynamic behavior from the system by capturing the operational contacts of transaction data. The diagram below shows the typical artifacts created in a forward engineering process [49].

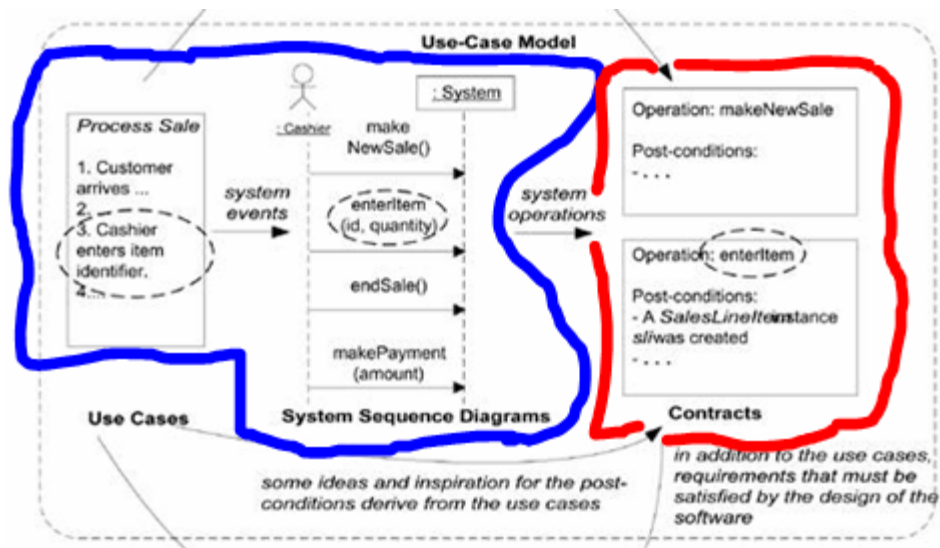


Figure 4. Extracting Operational Contracts (diagram adapted from [49])

Using the system, the actions and flow through a user interface can be performed in a controlled environment (such as a test system). The focus here is to replay a transaction whose side-effects and change to the data store is not well understood. This activity is represented by the boundaries in blue above. Most ERP systems use a RDBMS as its data store. One feature of modern RDBMS is the presence of “Active Elements”. Active elements like stored procedures and database triggers are often used to implement complex database integrity constraints. For the purpose of extracting committed transaction details, database triggers can also be used to audit changes to tables and record the information needed for the operational contracts in the use case model (in Figure 4). This is essentially a “black box” reverse engineering approach focused on a subset of the system one use case at a time. However, the triggers can not be hand written since the number of tables and structure of the columns in the database is not known in advance. The triggers have to be dynamically generated using the metadata facilities available in the RDBMS. Fortunately, modern RDBMS, such as Oracle, have this available. For example, to inspect the tables and columns in the database, one only needs to query against the USER_TABLES and USER_TAB_COLUMNS dictionary tables.

```

SQL> desc user_tables
Name                                         Null?    Type
-----
TABLE_NAME                                  NOT NULL VARCHAR2(30)
TABLESPACE_NAME                             VARCHAR2(30)
etc...

SQL> desc user_tab_columns
Name                                         Null?    Type
-----
TABLE_NAME                                  NOT NULL VARCHAR2(30)
COLUMN_NAME                                 NOT NULL VARCHAR2(30)
DATA_TYPE                                    VARCHAR2(106)

```

Figure 5. Oracle Dictionary Tables For user “tables” and “columns”

With the knowledge of the table names and their columns, database trigger code can be generated. A template for such a trigger is shown below in Figure 6.

```

CREATE OR REPLACE TRIGGER DEBUG_TRIGGER_XX
AFTER INSERT OR UPDATE OR DELETE ON TABLE_NAME FOR EACH ROW
DECLARE
IF INSERTING THEN
    // generate trace in a log table
END IF;
IF DELETING THEN
    // generate trace in a log table
END IF;
IF UPDATING THEN
    // generate trace in a log table
END IF;
END ;

```

Figure 6. Sample Database Trigger Template

In Figure 6. The template takes the table name and generates trigger code for the portions highlighted in bold. An example of a transaction captured by the triggers is shown below in Figure 7.

Database Transaction Log			
No.	Timestamp	Operation	Table Name
23160	12/12/2005 11:42:27	UPDATE	FGT_RP_REPORT_ENGINE
23161	12/12/2005 12:42:4	UPDATE	TPT_PERSON_LOG

Database Transaction Log Detail		No. 23160	Operation: UPDATE	Table: FGT_RP_REPORT_ENGINE	Time: 12/12/2005 11:42:27
Column Name	Old Value			New Value	
CI_NAME	crystal mterprise 9 ras edition			crystal enterprise 9 ras edition	
DESCRIPTION	Crystal Enterprise 9 RAS Edition			Crystal Enterprise 9 RAS Edition	
FLAGS	000000000			000000000	
ID	rpteg000000000000001			rpteg000000000000001	
JAVA_CLASS_NAME	com.saba.report.crystal.Crystal9ReportA dapter			com.saba.report.crystal.Crystal9ReportA dapter	
NAME	Crystal Enterprise 9 RAS Edition			Crystal Enterprise 9 RAS Edition	
PROPERTIES	reportFileLocation=;rasServerName=;			reportFileLocation=C:\SabaWeb\web\CrWeb;rasServerName=ussclmsrep01t.hds.com;	
TIME_STAMP	ts			1240464613	

Database Transaction Log Detail		No. 23161	Operation: UPDATE	Table: TPT_PERSON_LOG	Time: 12/12/2005 12:42:4
Column Name	Old Value			New Value	
BAD_LOGIN_CTR	0			0	
ID	emplo000000000001000			emplo000000000001000	
LOGGED_ON	11-28-05 03:51:14			11-28-05 03:51:14	
LOGIN_CTR	0			0	

Figure 7. An Example of a committed transaction captured by database triggers.

This information gathering technique helps to validate requirements analysis and captures existing system behavior into UML diagrams for discussions. An example UML sequence diagram with operational contracts captured using database triggers is shown in Figure 8. The focus of the diagram is on the entity object “Password” and the state changes it undergoes based on the sequence of events. The UML model is also decorated with the “screenshots” of the user interface to emphasize the use case actions.

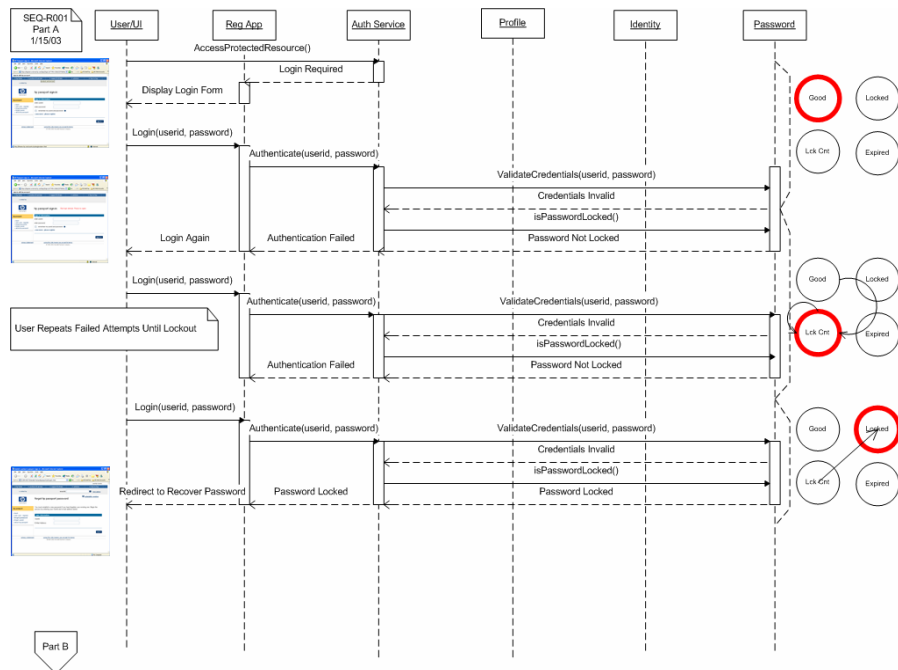


Figure 8. Sequence Diagram Showing Operational Contracts Extracted Using Database Triggers.

This is possible due to the power of the Oracle data dictionary, the SQL query language that provides access to the metadata, and the availability of extensible active elements in the database.

Java also has a Metadata facility and a reflection API. Thus, at least from a conceptual perspective, it should be possible to reverse engineer a running application from the reflective facility in the Java Virtual Machine. Unfortunately, the existing reflection API is too limited. For one, there is no declarative query language (like SQL) for selecting all the classes and objects available. And, second, there are no active elements comparable to database triggers. Well, at least not until the advent of AOP!

3.4 Active Models: Hope for the Future

An Active Model is a model that is derived from metadata and exists within the software itself. In the case of the Oracle RDBMS, the metadata used to derive the model is the same data the RDBMS uses for optimization and enforcement of database structure and constraints. Thus, to be an “Active

Model”, the model describes the structure and behavior of the software it models as well as enforces that same structure and behavior. Thus, the model lives with the system and is always in sync.

With AOP, we have taken one step forward towards this possibility for Java. Consider for instance, the popular example of the tracing aspect in AspectJ. Figure 9. shows the code for a tracing aspect that weaves in tracing behavior for a selected set of classes. This code is conceptually equivalent to the database triggers template in Figure 6.

```
import java.io.FileOutputStream;
import java.io.PrintStream;
import java.io.FileNotFoundException;
import org.aspectj.lang.JoinPoint;

public aspect TracingAspect {

    pointcut classes(): within( cmpe221.* ) ||
                       within( edu.sjsu.engr.cmpe221.presentation.* ) ||
                       within( edu.sjsu.engr.cmpe221.action.* ) ||
                       within( edu.sjsu.engr.cmpe221.form.* ) ;

    pointcut constructors(): execution(new()) ;

    pointcut methods(): execution(* *(..)) ;

    before(Exception e): handler(Exception) && args(e) {
        System.out.println( "Exception Thrown: " + e.toString() ) ;
        System.out.println( thisJoinPoint.toLongString() ) ;
        System.out.println( thisJoinPointStaticPart.getSignature().getDeclaringType().getName() ) ;
    }
    before(): classes() && constructors() {
        doTraceEntry(thisJoinPoint, true);
    }
    after(): classes() && constructors() {
        doTraceExit(thisJoinPoint, true);
    }
    before(): classes() && methods() {
        doTraceEntry(thisJoinPoint, false);
    }
    after(): classes() && methods() {
        doTraceExit(thisJoinPoint, false);
    }
}
```

Figure 9. Tracing Aspect in AspectJ with before and after advice on constructors and method calls

An example of applying the tracing aspect to an existing Java code base is shown below. In the example, the login page in a web application is presented, and the user provides a user id and password submitting the authentication request to the server. In the back-end, a servlet engine processes the request and the AOP enriched code logs the sequence of messages to the console.

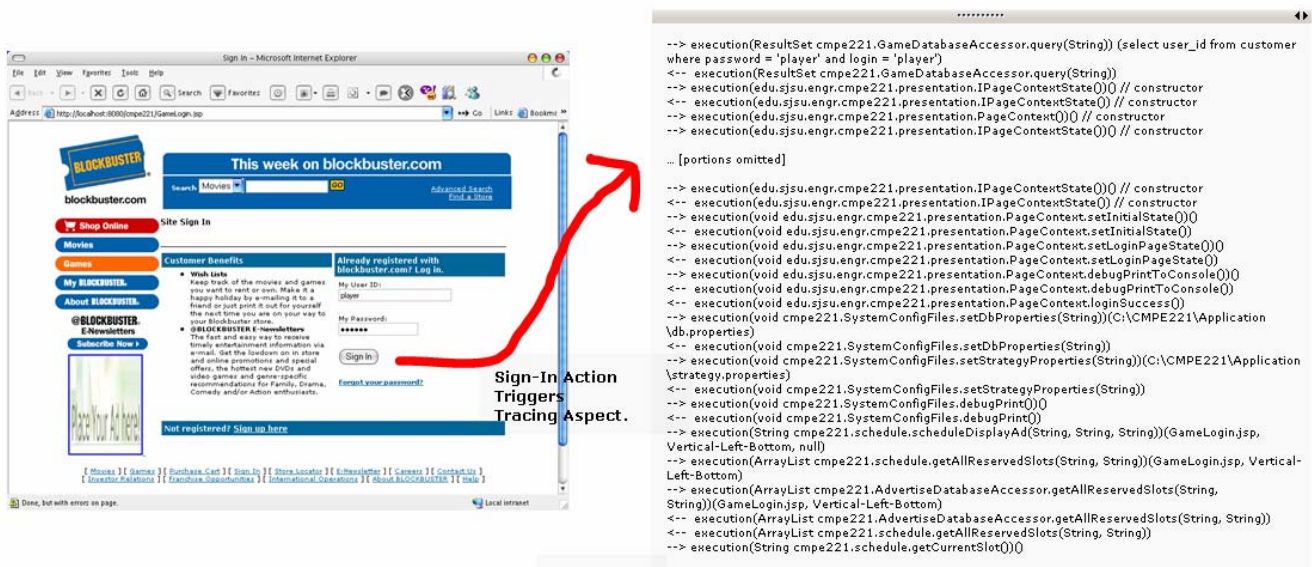


Figure 10. Login Scenario Demonstrating Tracing Aspect

The database trigger example demonstrates “black box” reverse engineering, while the Java tracing aspect shows the power of AOP can be used in a “white box” reverse engineering scenario. Both techniques can be used concurrently to build UML models that represent the system behavior. However, this process is currently a labor intensive undertaking and is typically done on a small subset of the system and only when needed.

The work on Naked Objects [28] and BabyUML [31] has shown us the value of active “domain models” by relieving the developer from the user interface work and providing the framework for automatically generating it. The agile practice says, work at the programming language level and evolve the system with the user – in effect, treating the software system and the model itself as one entity. UML and MDA proposes to focus only on the higher levels of abstraction and let the tools do all the transformation work and mapping to programming languages. The software product line approach as embodied in its current form in Software Factories suggest creating customized domain specific frameworks and custom domain-specific languages and tools to automate software development. But,

do these approaches help with current chronic problems in current enterprise software development and integration? In the context of the issues facing the enterprise today, how do these approaches solve these problems? And, what is the first step towards that goal? Table 5. Summaries the current state-of-the-art methods against the problems previously discussed in enterprise software.

Table 5. Innovations and Problems that Motivate Active Models

Problems in Enterprise Software: Solution Evolution, Incompatibility, Requirements Mismatch, Documentation Centric, Isolation of, Logical and Physical Layers in Deployment, and Collaboration amongst Distributed Teams.	
Naked Objects and BabyUML	<ul style="list-style-type: none"> • Main strength is in active domain model (Naked Objects) and the focus on building a UML Virtual Machine (BabyUML). • Doesn't address evolution directly, but the roadmap towards a virtual machine using UML MOF and the focus on large scale components could provide the framework for work in isolation.
Agile Software Development	<ul style="list-style-type: none"> • Focuses on collaboration in small teams and evolution on a smaller scale (i.e a single system and not entire integrated solutions) • Doesn't address the need for maintaining documentation or technical issues directly. Emphasizes continuous integration and testing, but doesn't prescribe specific tools technology required.
UML and MDA	<ul style="list-style-type: none"> • Main strengths in standard modeling notation, but models to date are static • Proposes a large scale automation roadmap with MDA, but current focus is on standards for tools development.
MDD and Software Factories	<ul style="list-style-type: none"> • Expands on agile software development and attempts to scale up to larger teams • Addresses incompatibility with domain-specific models, but advises custom notations and tools. Not sure how multiple domains will be integrated. • Addresses documentation maintenance with model driven generation. • Not clear how evolution will be solved in the context of enterprise packages. Tools currently focus on single product family. Not sure how multiple families and integration with legacy systems will be addressed. • Suggests new component market approach with meta-tools and domain-specific frameworks. But, it is currently unclear if any standards will be defined.

What might an active model look like? To answer this question, we need to redefine the notion of a “model”. In today’s documentation centric world, models in UML are stored in static documents. These are passive models! But there is a deeper issue behind the analogy we use for them. That is to say the word “model” doesn’t seem to fit in the context of software. For software, which aims to model

the real world, it seems more appropriate that a model of software should also be software. As such, today's UML model is just a "snapshot" --- i.e. a view at a point-in-time.

The notion of a "Virtual Machine" for UML seems promising for Active Models [14][50][17]. It suggests looking at the current state of meta-models and integrating them with the operating environment. We have done this before -- in the areas of databases with the relational model and active database elements. Sometime back object-oriented databases fought to take over the database market, but lost. Since then, the work in object-oriented application frameworks with respect to databases has been focused on abstracting the database away and not integrating with it. Today's RDBMS have embraced the object-oriented models and more (i.e. supporting multimedia and XML). Perhaps it is time to unify the application framework with the database.

4. Towards Integrated Active Systems

4.1 Traditional Software Development & Model-Driven Architecture (MDA)

In the traditional software development methods currently in practice, the process of developing a single system from high-requirements and abstractions to a running executable is done via manual transformation. As shown in Figure 11, a typical Unified Process, the software development artifacts include documentation, UML models, and code. As noted, the current practice uses a general-purpose platform (such as a Java Application Server and Relational Database), a general-purpose modeling language (UML), and general-purpose programming languages (such as Java or C#). Due to this emphasis, there exists a large semantic gap between the solution space and the problem space. As a result, traceability from requirements to solution is often difficult, if not impossible. MDA attempts to address this by automating most of this manual transformation from the problem space to the solution space, effectively removing the need to develop in lower-level 3GL languages or to maintain

traceability. In essence, MDA’s main focus is strictly in the problem space and defers all translation work to automated tools.

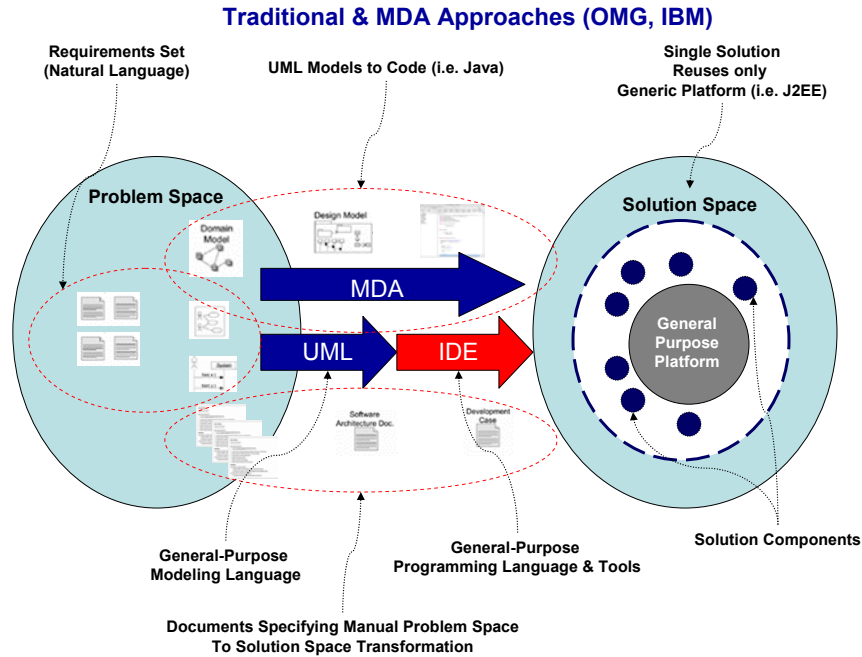


Figure 11. Traditional Software Development & MDA approach

4.2 Domain-Specific Software Development

In contrast to single system development and MDA, domain-specific software development focuses on partial solutions for a family of systems. A high-level language, called the domain-specific language, attempts to capture core concepts from the problem space and is supported by a generator and domain-specific platform. These three key elements: the DSL, Generator, and Domain-Specific platforms work together and provide a total evolving solution for problem space to solution space mapping and automation. As shown in Figure 12, the “grey” area in the solution space represents the domain specific “partial solution” which evolves with the generator and DSL as more concepts from the

problem space are understood and generalized. One difficulty with this approach is the complexity and high-cost of developing the DSL and generator. As a result, this approach is not widely practiced.

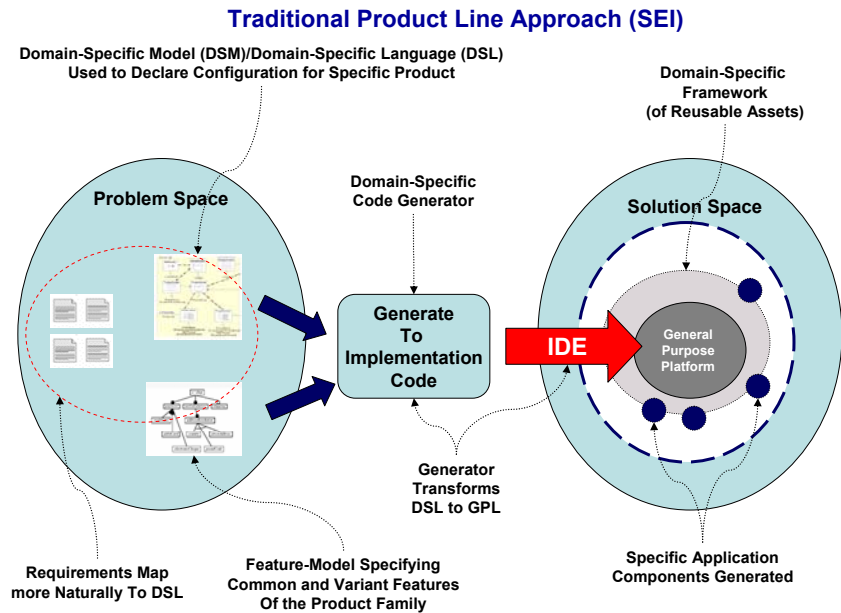


Figure 12. Software Product Lines & Generative Software Development

4.3 Metatools for Domain-Specific Languages

To address the complexity and high-costs of developing DSL tools and languages, a new market for Metatools (tools that generate tools) is growing. For example, Microsoft’s Software Factories approach introduces a Software Factory DSM/DSL Tool Generator, a graphical meta modeling environment for creating custom modeling languages and their editors, debuggers, and generators. Figure 13 shows how this integrates and extends the work from software product lines and generative software development.

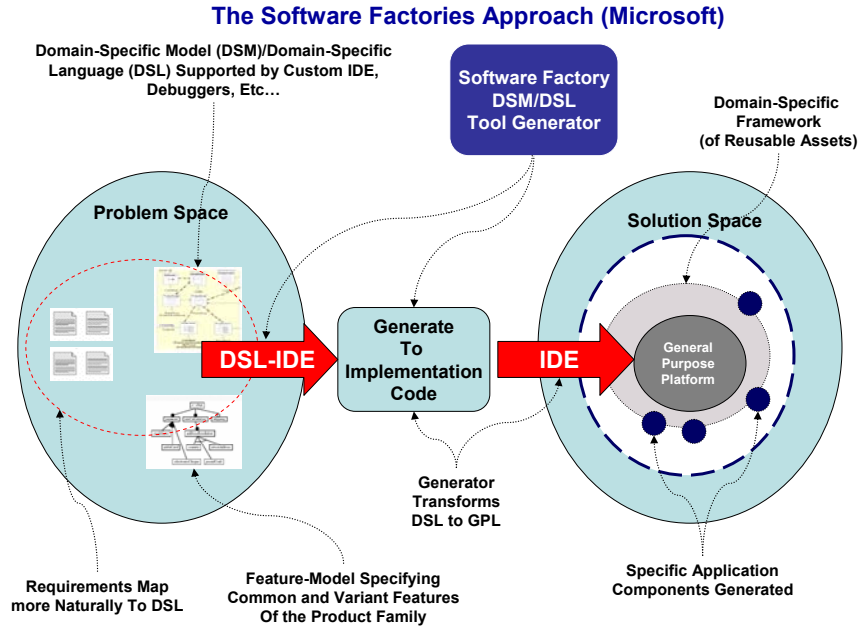


Figure 13. Software Factories Tools for Product Lines & Generative Software Development

4.4 Software Development with Active Systems

Active Systems, introduced in this research, represents a convergence of the problem space and solution space into a highly interactive and dynamic environment. As shown in Figure 14, Active Systems contain three main sub-systems: Active Documents, Active Models, and Active Database. At the foundation of this approach is a integrated metamodel managed by the Active Database which supports declarative programming, model diagram generation, documentation generation, and model simulation. Inputs to the system are described as “manipulation” and outputs as “projections”. Traditional approaches and tools can add assets (Active Objects) into the Active Database and new tools and languages can be used to “manipulate” – i.e. associate and map assets to Active Models and Active Documentation. The primary benefit to this approach is that it emphasizes system evolution – via asset “manipulation” and not code “compilation”.

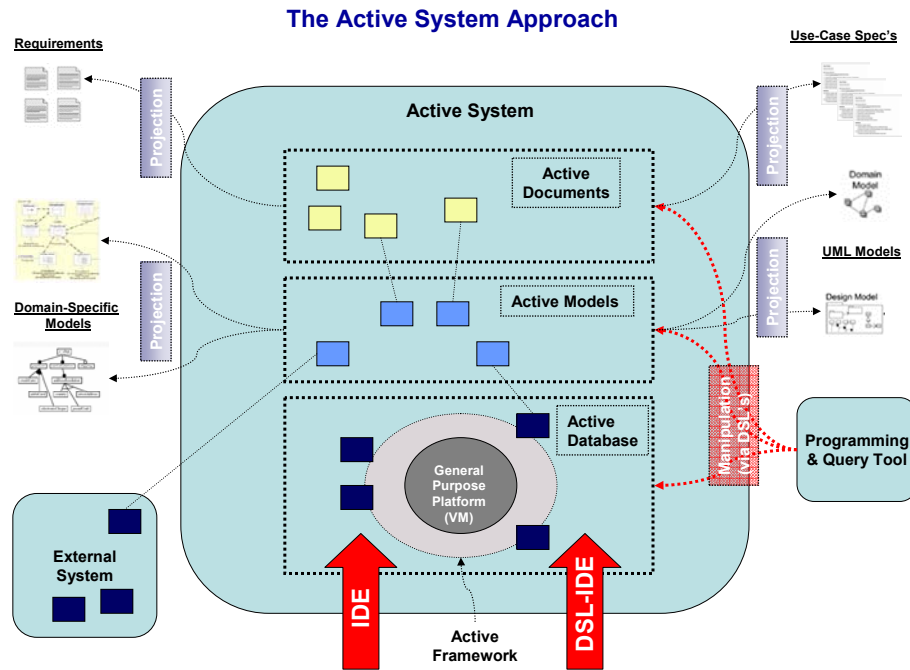


Figure 14. Active Systems Integrate Domain-Specific and General-Purpose Development Methods

5. The Experiment

To explore first steps towards the goal of Active Models, let us focus on a few UML diagrams – the subset most often used in analysis and design. Granted there are other graphical notations, as suggested in Software Factories, but using UML has some benefits. UML is the most widely used in enterprise software since it has incorporated the main modeling concepts of the prior generations. Thus, UML is a good starting point, and the experiment will focus on using three core models: the domain model, the use case model, and the design model as highlighted by Larman [49] in Figure 15.

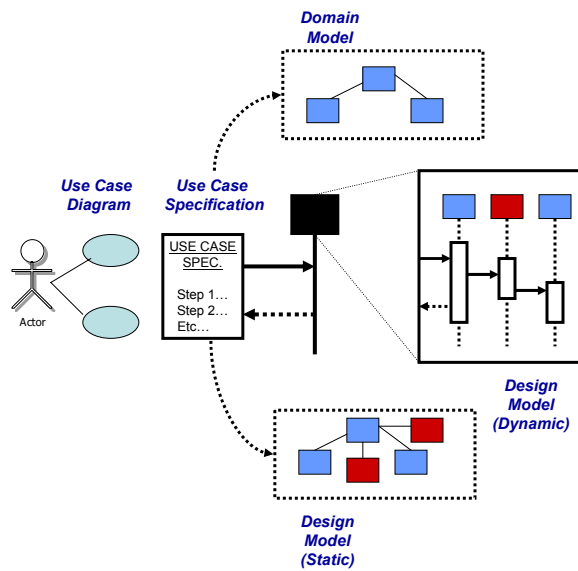


Figure 15. UML Models and Their Relationships

The main focus of the experiment will be on the use case and design models. Active Domain models have been proposed in Naked Objects and will not be highlighted in this work. However, a discussion on how this work relates with Active Domain Models will be presented in the conclusion.

5.1 Scope of Experiment & Investigation

The scope of the experiment includes the following activities:

- Develop a metamodel for a subset of UML and add additional support for use cases as a basis for the simulation of active models and generation of HTML documents and Java code.
- Develop a domain-specific language for describing use case specifications and sequence diagrams to support capturing of requirements
- Leverage the work from JBoss AOP to develop an execution framework for the simulator

- Develop a web-based interface for interacting with the framework, for viewing use case specifications, and displaying interactive UML diagrams.
- Study the practical application of Active Systems using case studies

5.2 The Metamodel

A relational database schema for the metamodel was developed to support capturing the metadata for UML Use Cases and Sequence Diagrams. A portion of this metamodel is shown in Figure 16. The main concepts of steps, use case flows, and scenarios are represented in the model. Using a relational database also provides some added benefits to the research work. Mainly, this benefit comes from leveraging SQL's ability to query, join, filter, and transform data from the metadata tables which provides a good foundation for developing document and code generation facilities.

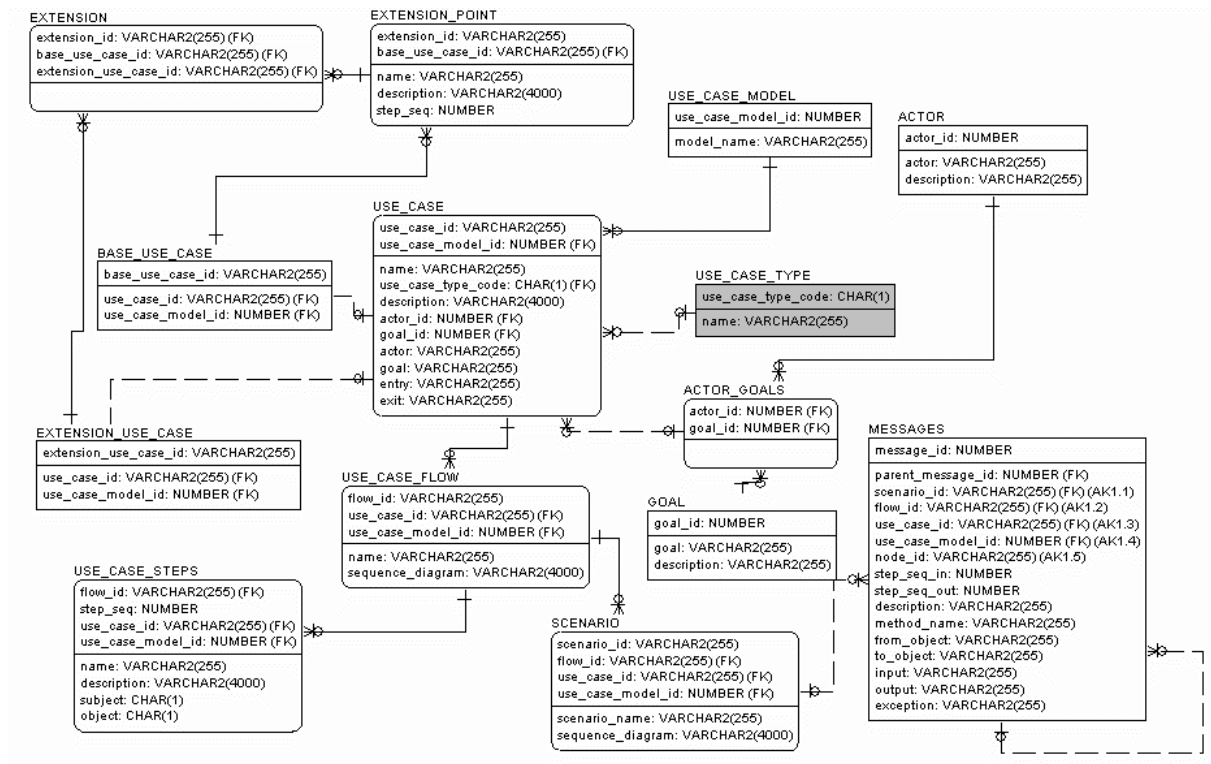


Figure 16. Metamodel for Use Cases

5.3 A Domain Specific Language for Describing Use Cases

To demonstrate the connection of documentation with software, a dynamically generated use case specification will be used. HTML and the Web provide an ideal interface medium for this demonstration. Since there is a growing trend for enterprise software to reach out beyond the boundaries of the enterprise to partners and customers, enterprise applications are increasingly supporting the web browser as an interface. HTML, with its roots in documentation generation also makes this an ideal choice. A SQL-Like DSL will be used to describe the use case and a DSL interpreter created to generate the proper elements in the use case metamodel.

5.4 JBoss AOP

JBoss AOP is amongst one of the recent entries and next generation dynamic framework in the AOP arena. The JBoss AOP framework's "hot deploy" capability is especially useful for this work in the area of model simulation. In addition, JBoss's approach of "Pure Java" using reflection and interception compared to a language extension approach, like AspectJ, makes the immediate practical value of the framework apparent. The combination of the JBoss AOP framework and the relational database used in this research effectively approximates an Active System with currently available technology.

5.5 Web Interface for Interactive UML Diagrams

To investigate the relationship of active models and simulation, the techniques for integration of AOP and Use Cases [3] will be explored. AOP at the level of Use Cases breaks concepts apart by defining variation points for their composition as Use case fragments, called "slices". The approach for demonstration will use a UML sequence diagram and an interactive session with a modeler for selecting

different variation points. The end result is a sequence diagram generated interactively by selecting use case extensions.

6. Experimental Platform and Research Prototype

6.1 Overview of Architecture

This section describes the high-level architecture and major open source, research, and off-the-self components used to build the experimental platform. These components are organized into three main layers: the database tier, the Java JVM “middleware” tier, and the presentation tier. In the database tier, an Oracle Database (version 10g) was used as the foundation for managing tables that make up the metamodel repository and simulation state. Additionally, Oracle was chosen due to the extensive functionality of active elements in the form of Oracle stored procedures. A few PL/SQL packages were developed to manage updates to the metamodel, maintain execution state of a session, and generate diagram code. These diagram code generators create input code for the graph generation frameworks in the presentation tier. The middle tier contains the Java JVM, JBoss AOP, the interpreter for the Use Case/UML textual “DSL” language, and a set of components hosted by Apache Tomcat. These components manage interaction with the end user and include: the main Console Servlet which takes commands from the end user, and the Oracle XDK components that deal with generating XML from SQL and transforming them into dynamic HTML documents via XSLT transformation templates. In the presentation tier, which is hosted by a standard internet browser, the web user interfaces renders dynamic HTML documents and references generated graphical diagrams. For input from the end user, there are two main areas of the web user interface: the DSL console and the Simulation console. The DSL console primarily takes commands and passes this on to the DSL interpreter. The simulation console allows the end user to single step through a sequence diagram with “next” and “previous”

buttons and renders the resulting UML sequence diagram for each time step. An illustration of this high-level architecture is shown in Figure 17. Notice, in addition to the toolset developed as part of this research, standard tools were also used for development. For example, the Eclipse Java IDE was used to compile Java code, and Oracle SQL*Plus utility used to create tables and compile database stored procedure packages. The rest of this section discusses the implementation of the research prototype and the main collaborations amongst the major system components in the experimental platform.

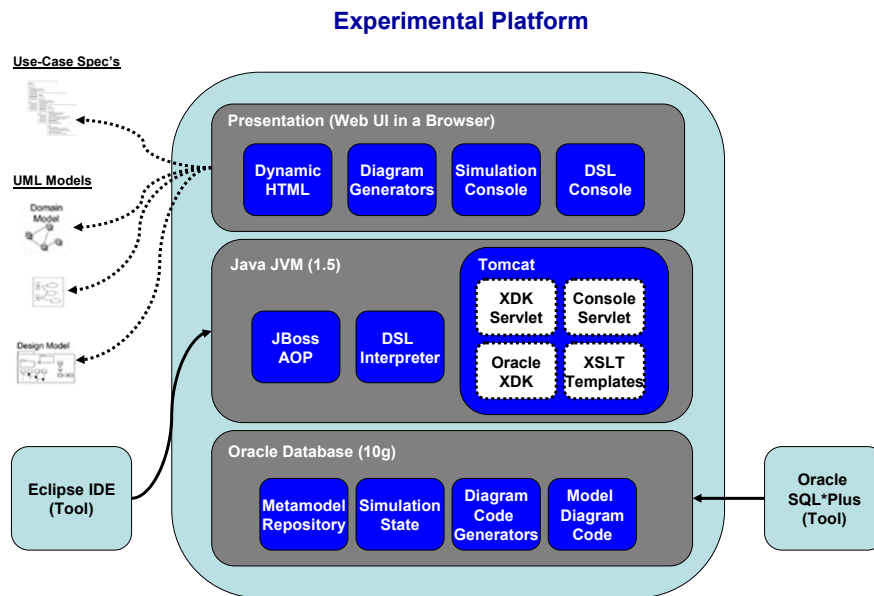


Figure 17. Architecture of Experimental Platform

6.2 DSL Language Design & Implementation

For the domain specific language, the JavaCC parser generator was used. JavaCC takes a grammar described in JavaCC's BNF form and generates Java code for a parser. Inside the JavaCC grammar file are also customized Java code which invoke a set of classes that make up the DSL interpreter. For example the code below (Figure 18) shows the starting node in the grammar for the create use case command. The example code defines a non-terminal *CreateUseCaseCommand()* which

matches the token `<CREATE>` followed by the `UseCaseExpression()` non-terminal, and optionally (zero-or-more) non-terminals `TheSteps()` and `WithStatements()`, consecutively.

```
void CreateUseCaseCommand() :
{
    theCommand = new CreateUseCaseCommand() ;
}
{
    (
        <CREATE> UseCaseExpression() (TheSteps())? (WithStatements())?
    )
    {
        // add code here
    }
}
```

Figure 18. DSL Non-Terminal For Create Use Case Command

Also shown, is custom Java code enclosed in curly braces, which instantiates the `CreateUseCaseCommand` object. This object, referenced by `theCommand` in subsequent evaluations of non-terminals, is used to store the parsed data – the details regarding the command itself. An example of how this is done is shown below (Figure 19) where the actor variable is added to the command object during the evaluation of the `WithStatement` branch of `CreateUseCaseCommand`.

```
void WithStatements() : {}
{
    (
        With() ( ActorVariable() | GoalVariable() | EntryVariable()
                | ExitVariable() | ExtensionVariable() )+
    )
}

void ActorVariable() :
{
    Node n = jjtThis ;
}
{
    (<ACTOR> ("=")? TextLiteral())
    {
        ASTTextLiteral txt = (ASTTextLiteral) n.jjtGetChild(0) ;
        ActorWithVariable actor = new ActorWithVariable( txt.getText() ) ;
        ((UseCaseCommand) (theCommand)).addVariable( actor ) ;
    }
}
```

Figure 19. Example of JavaCC Integration with Java DSL Interpreter Objects

The interpreter for the DSL contains five commands which implement the `UseCaseCommand` interface (Figure 20). Once the parser (generated by JavaCC) validates the correctness of a command, the parser instantiates and completes the set of required and optional properties selected for the command. Each

command, as specified by the parent interface of *UseCaseCommand*, the *Command* interface, implements the *execute()* method which will be invoked by the Console Servlet.

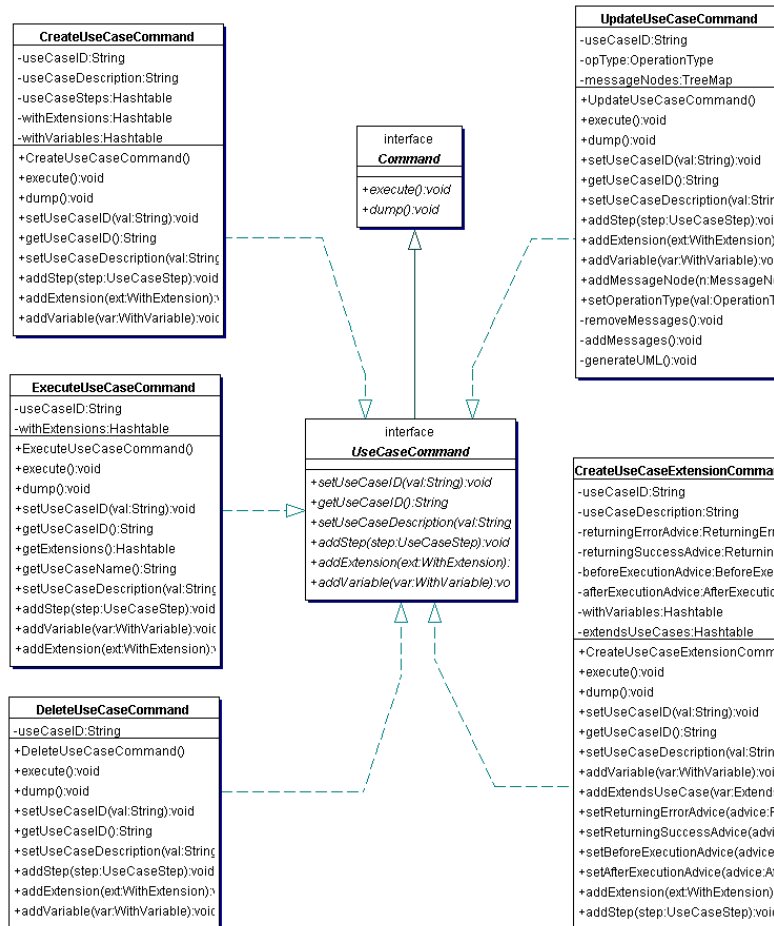


Figure 20. DSL Interpreter Command Class Diagram

Upon execution, the command object creates the appropriate metadata in the metamodel repository for the command and invokes the diagram code generator. A sequence diagram that illustrates this collaboration is shown in Figure 21. The parser, interpreter, metamodel repository, and diagram code generators work together to capture the use case specification into a repository for use in document and diagram generation. The following sections will cover the implementation specifics for each of the commands in the prototype. Most of the commands were not implemented to their fullest extent as specified in the design, but enough of their features were implemented to support the research

work. For example, while the parser is complete, the interpreter will ignore parts of the command in some cases.

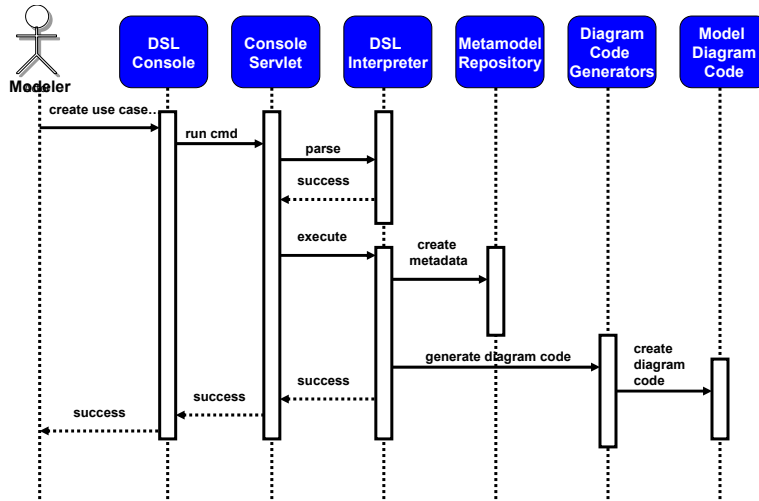


Figure 21. UML Sequence Diagram for DSL Interpretation

6.2.1 Create Use Case

The main purpose of the *Create Use Case* command is to allow a modeler to express and create a typical use case specification. Recall, from the metamodel, a use case contains actors, goal, flows, steps, and many other elements. This is captured in a database schema diagram (as shown in Figure 16. Metamodel for Use Cases). The DSL interpreter classes that support this command is shown in Figure 22. This is basically an object-oriented equivalent to the relational metamodel. The *CreateUseCaseCommand* is the main workhorse of the group and is responsible for managing collections of different types of command options (Steps, Extensions, and Variables). This class also has the *execute()* method, which when invoked, effectively “interprets” the command and creates the appropriate metadata.

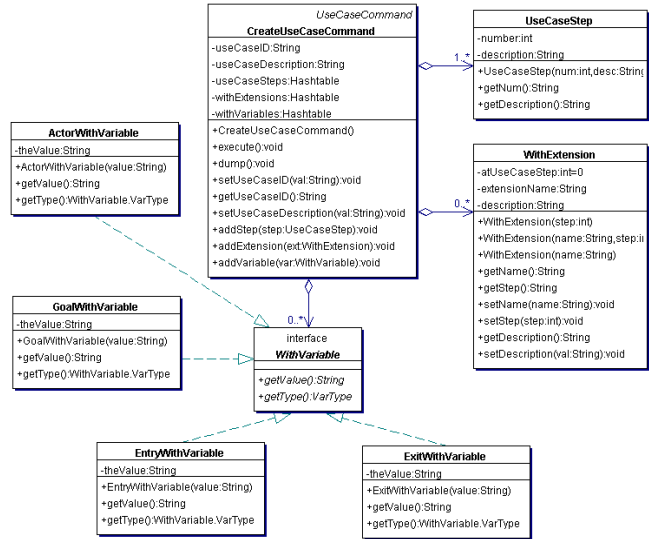


Figure 22. Object Model For Create Use Case Command (Class Diagram)

Another important method that all *UseCaseCommand* classes must implement is the *dump()* method. This method is used to dump the state of a command to the console to aid in debugging and validation of parser instantiation of command parameters. An example of this method is shown in Figure 23.

```

public void dump() {
    System.out.println( "====" );
    System.out.println( "Class: " + this.getClass().getName() );
    System.out.println( "Use Case ID: " + this.useCaseID );
    System.out.println( "Use Case Description: " + this.useCaseDescription );
    System.out.println( "Steps: " );
    if ( this.useCaseSteps != null )
    for ( Enumeration k=this.useCaseSteps.keys(); k.hasMoreElements(); )
    {
        String key = (String) k.nextElement();
        String value = ((UseCaseStep)this.useCaseSteps.get(key)).getDescription();
        System.out.println( "    ["+key+"] = " + value );
    }
    System.out.println( "With Variables: " );
    if ( this.withVariables != null )
    for ( Enumeration k=this.withVariables.keys(); k.hasMoreElements(); )
    {
        String key = (String) k.nextElement();
        String value = ((WithVariable)this.withVariables.get(key)).getValue();
        System.out.println( "    ["+key+"] = " + value );
    }
    System.out.println( "Extensions: " );
    if ( this.withExtensions != null )
    for ( Enumeration k=this.withExtensions.keys(); k.hasMoreElements(); )
    {
        String key = (String) k.nextElement();
        String value = ((WithExtension)this.withExtensions.get(key)).getStep();
        System.out.println( "    ["+key+"] = " + value );
    }
    System.out.println( "Database Configuration: " );
    FrameworkTools.Database.dumpConfig();
    System.out.println( "====" );
}

```

Figure 23. Debug *dump()* method in Create Use Case Command

```

create use case buy_product described as "Customer wants to buy a product from the online store."
steps
  step 1 "Customer browses catalog to select items to buy"
  step 2 "System displays catalog"
  step 3 "Customer selects items to buy"
  step 4 "System acknowledges selection"
  step 5 "Customer goes to check out"
  step 6 "System displays check out screen asking for shipping information"
  step 7 "Customer fills in shipping information (address, next-day or 3-day delivery, etc...)"
  step 8 "System calculates total cost (including shipping) and displays payment options"
  step 9 "Customer fills in credit card information and places order"
  step 10 "System authorizes purchase and displays confirmation"
with
  actor "Customer"
  goal "Buy Product(s)"
  extension check_out at step 5
    described as "System displays check out screen with shipping information"
  extension authorize_purchase at step 9
    described as "System authorizes purchase"

```

**Results
From
Parsing
Confirmed
via call
to dump()**

```

Class: CreateUseCaseCommand
Use Case ID: buy_product
Use Case Description: Customer wants to buy a product from the online store.
Steps:
  [9] = Customer fills in credit card information and places order
  [8] = System calculates total cost (including shipping) and displays payment options
  [7] = Customer fills in shipping information (address, next-day or 3-day delivery, etc.)
  [6] = System displays check out screen asking for shipping information
  [5] = Customer goes to check out
  [4] = System acknowledges selection
  [3] = Customer selects items to buy
  [2] = System displays catalog
  [10] = System authorizes purchase and displays confirmation
  [1] = Customer browses catalog to select items to buy
With Variables:
  [ACTOR] = Customer
  [GOAL] = Buy Product(s)
Extensions:
  [check_out] = 5
  [authorize_purchase] = 9

```

Figure 24. Example Create Use Case Command & Parser Results

Figure 24 shows an example create use case command and the result from parsing shown by calling the *dump()* method. Once the command is properly instantiated, as shown, by the properties set in the command object, the command can be invoked via a call to the *execute()* method. As shown previously in the sequence diagram (Figure 21), the execute call does two things. First, the metadata for the command is created in the metamodel repository; and, second, the diagram code (i.e. code for generating images) is updated to represent the command. In this case, the diagram code for a use case command is a “black box” sequence diagram. In some cases, the command object (depending on its type) may also invoke the diagram generator to create the image. Or, alternatively, can defer the generation of images to a later time when a modeler wishes to “view” the diagram. As shown in the code (Figure 25), the create use case command invokes the diagram generator via a system call. More details with regards to the diagram generator and the document/diagram generation collaboration are covered in Section 6.3.


```

public void execute() throws Exception {
    String v_usecaseid = this.useCaseID ;
    String v_usecasedesc = this.useCaseDescription ;
    String v_actor = ( ((WithVariable)withVariables.get("ACTOR")) == null ? "" : ((WithVariable)withVariables.get("GOAL")) == null ? "" : ((WithVariable)withVariables.get("ENTRY")) == null ? "" : ((WithVariable)withVariables.get("EXIT")) == null ? "" : ((WithVariable)withVariables.get("ACTOR")) == null ? "" : ((WithVariable)withVariables.get("GOAL")) == null ? "" : ((WithVariable)withVariables.get("ENTRY")) == null ? "" : ((WithVariable)withVariables.get("EXIT")) == null ? "" : "" );
    String cmd = "{ call pkg.create_use_case ( ?, ?, ?, ?, ? ) }";
    FrameworkTools.Database.execute_procedure( cmd, v_usecaseid, v_usecasedesc, v_actor );
    if ( this.useCaseSteps == null )
    for ( Enumeration k=this.useCaseSteps.keys() ; k.hasMoreElements() ; )
    {
        String key = (String) k.nextElement() ;
        String value = ((UseCaseStep)this.useCaseSteps.get(key)).getDescription() ;
        System.out.println( " [" +key+" ] = " + value );
        cmd = "{ call pkg.create_use_case_step ( ?, ?, ? ) }";
        FrameworkTools.Database.execute_procedure( cmd, v_usecaseid, key, value );
    }
    if ( this.withExtensions != null )
    for ( Enumeration k=this.withExtensions.keys() ; k.hasMoreElements() ; )
    {
        String key = (String) k.nextElement() ;
        String value = ((WithExtension)this.withExtensions.get(key)).getStep() ;
        String desc = ((WithExtension)this.withExtensions.get(key)).getDescription() ;
        System.out.println( " [" +key+" ] = " + value );
        cmd = "{ call pkg.create_extension_point ( ?, ?, ?, ? ) }";
        FrameworkTools.Database.execute_procedure( cmd, key, v_usecaseid, value );
    }
}
/* generate blackbox sequence diagram */

```

```

// generate graphic
String gencmd = "Z:/WORKSPACE/Eclipse/Prototype/metamodel/GenUmlSeq.cmd " +
System.out.println ( "Running UML Sequence Generator..." );
System.out.println ( " Command: " + gencmd );
Process p = Runtime.getRuntime().exec( gencmd );
p.waitFor();
if ( p.exitValue() == 0 )
{
    System.out.println( "UML Sequence Diagram Generation Succeeded!" );
}
else
{
    InputStream errStream = p.getErrorStream();
    for ( int j = errStream.available(); j>0; j++ )
        System.out.write( errStream.read() );
}
}
catch (Exception e){
    System.out.println( e );
}
}

```

Calls to create metadata

Call to generate diagram

Figure 25. Portions of the execute() command for Create Use Case Command

The remaining commands implemented follow the same basic design. The following section will present their design and implementation details specific to each command. However, the *Delete Use Case* command will be skipped in the discussion because the implementation is trivial.

6.2.2 Update Use Case

The *Update Use Case* command shares much in common with the *Create Use Case* command. That is, this command can add/remove actors, use case steps, with variables, and all the optional parameters; but, the implementation of this command currently in the research prototype does not support this feature³. Rather, the emphasis of the implementation is on the additional “special” responsibility this command has – the ability to describe the dynamic design model as sequence diagrams for a use case. As such, the example below (Figure 26) demonstrates the command with a single sequence diagram message.

³ Alternatively, the “Delete Use Case” command used in conjunction with the “Create Use Case” command can be used instead of update.

```

update use case buy_product
add main scenario messages
7.1.1
from OnlineStore to ShippingService
requesting checkPrice with "products", "fromAddress", "toAddress", "deliveryType"
returning "total cost and estimated delivery date"

```

Figure 26. Example Update Use Case Command

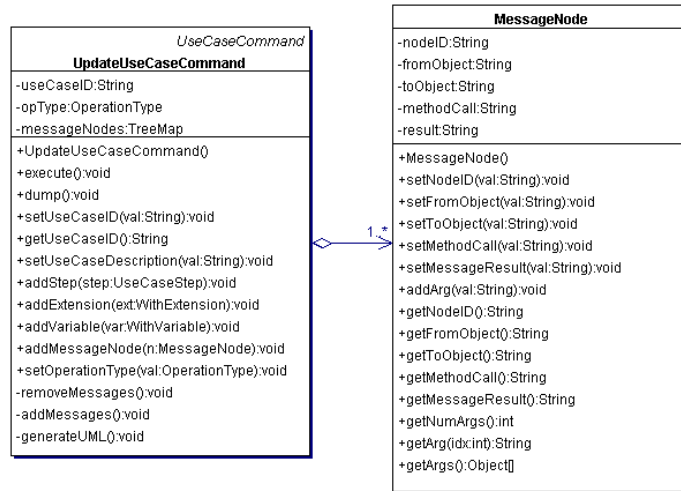


Figure 27. Object Model for Update Use Case Command (Class Diagram)

The object model in Figure 27 implements the command with a collection of *MessageNode* objects. Upon the invocation of *execute()*, the message nodes are scanned and inserted into the metamodel repository. In addition, this command also supports removing messages from the sequence diagram. This is represented by the *opType* internal state which is set by the parser when a user types in “... remove main scenario message” instead of “... add main scenario message”.

6.2.3 Create Use Case Extension

Figure 28 shows an example of a *Create Use Case Extension* command. Use Case Extensions in this research differs from the traditional use case extension. One main difference is the explicit representation of aspect-oriented concepts. In the example, the notion of AOP “joinpoints”, “pointcuts”, and “advice” are represented. This command also combines the capability to define external use case steps (i.e. similar to the *Create Use Case* command) with the definition of internal sequence diagram

messages (i.e. similar to the *Update Use Case* command). These messages represent “execution fragments” that can be inserted into the base execution flow of some other use case. As shown, the example extends the *buy_product* use case at the *check_out* extension point before the execution of the message at that point.

```

create use case extension check_out_extension
described as "Get current shipping options from service provider"
with
  goal = "Ensure that the latest shipping options are available at check out"
before execution
  advice steps
    step 1 = "Get current shipping options available"
    step 2 = "List of shipping options and prices (delivery type, current rates, etc...)"
  advice scenario messages
    1.1 from Extension to ShippingService
      requesting getShippingOptions
      returning "Shipping options and rates"
extending
  buy_product at check_out before execution
  
```

Figure 28. Example Create Use Case Extension Command

The object model in Figure 29 shows how the Create Use Case Extension command is implemented by the interpreter. A key difference with the *Update Use Case* command is where the *MessageNodes* are associated. In the *Update Use Case* command, there is only “one” collection of *MessageNodes*. For *Create Use Case Extension*, there is one collection for each advice. In addition, there is also a collection of *ExtendsUseCase* objects which represent the logical pointcut mappings to extension points in base use cases.

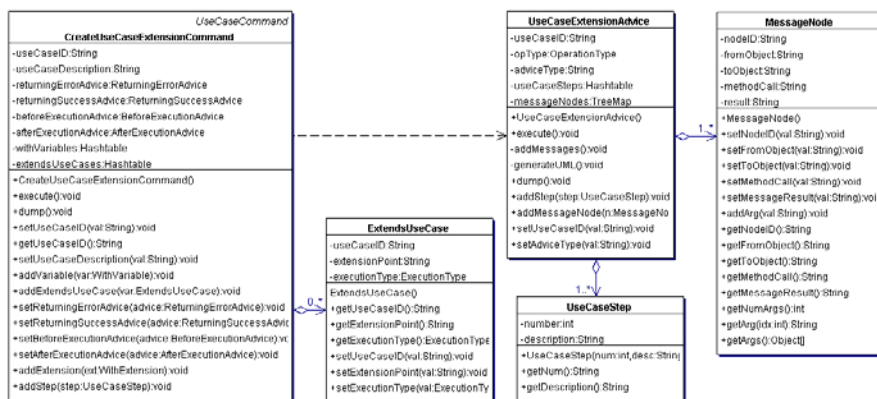


Figure 29. Object Model for Create Use Case Extension Command (Class Diagram)

6.2.4 Execute Use Case

Execute Use Case commands have an optional list of extensions as parameters to the command. When supplied, the semantics for the command says that the simulator should enable these extensions and include the message fragments define by their advice definition. An example of this command is shown in (Figure 30). The command requests an execution of the *buy_product* use case with two extensions.

```
execute use case buy_product
with extensions check_out_extension,
audit_authorize_purchase_extension
```

Figure 30. Example Execute Use Case Command

The object model for this command is shown in (Figure 31). The implementation is very simple. There is only a single collection of extensions that are used to create data in the metamodel repository. However, unlike all the other commands, there is much more to the implementation of this command than the parsing and interpretation phases. This command initiates a modeling session, switches from the DSL console to the Simulation Console, prepares an execution plan, and waits for additional simulation requests. The collaboration model is much more complicated.

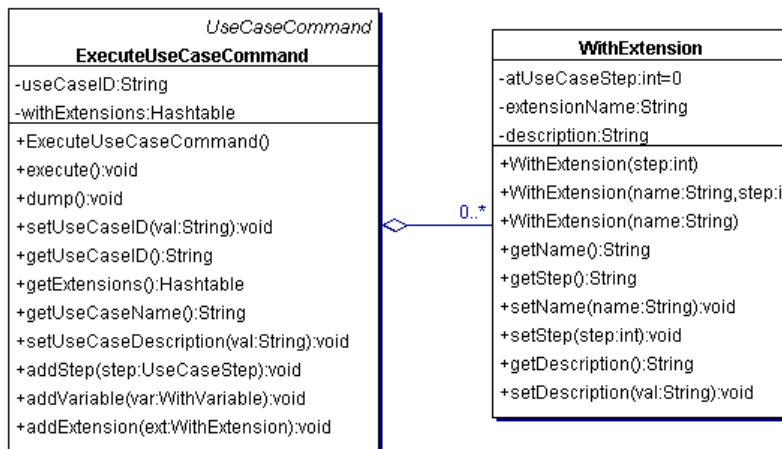


Figure 31. Object Model for Execute Use Case Command

The collaboration for simulation initiated by the Execute Use Case command is shown in Figure 37. The details on the implementation of the simulator will be covered in Section 6.4. For now, it is worthwhile noting that this collaboration includes the use of the Java JVM (via reflection) and JBoss AOP – two key components in the design of the simulation environment.

6.3 Diagrams and Document Generation

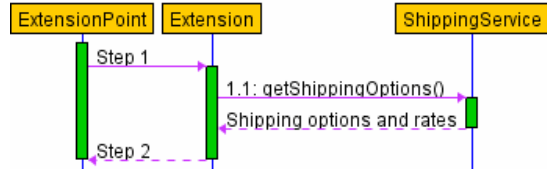
All the diagram generators used in this research follow a basic usage pattern. They take as input some ASCII text and generate as output an image in JPG or PNG format. The ASCII text is basically a DSL (i.e. a language) the diagram generator understands. Various diagram generators were integrated into the modeling and simulation environment. For use case document sequence diagrams, the *sequence* [77] toolkit was used. For use case diagrams, *GraphViz* [78] was used. For class diagram and sequence diagrams inside the simulator, *UMLGraph* [71] was used. The metamodel repository contains all the information about the different diagram types and can be queried via standard SQL and transformed with store procedure logic (PL/SQL). This is essentially the function of the diagram code generators which create the DSL code for diagrams and stores them in the database. With the exception of the simulation sequence diagrams, the calling mechanism to diagram generators is a system call to a shell command from within the Java JVM. For the simulation sequence diagrams, the system call approach did not work and a workaround was implemented. This workaround wrapped the generator with a CGI script hosted under Apache running on Cygwin. As a result, the generation of simulation sequence diagrams is very slow, but this can easily be corrected by moving the CGI script to another machine. Some examples of the diagram DSL code and the generated image are shown below in Figure 32.

```

ExtensionPoint {
  Extension."Step 1"->"Step2"{
    ShippingService."1.1:getShippingOptions()"
    ->"Shipping options and rates"{}
  }
}

```

Sequence Diagram in Sequence Language



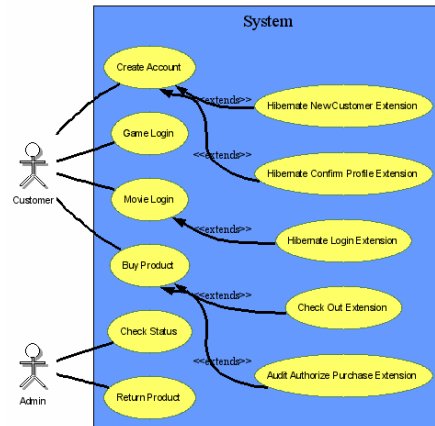
Generated Sequence Diagram

```

usecase.dot - WordPad
File Edit View Insert Format Help
audit_authorize_purchase_extension [label="Audit Authorize Purchase Extension"
Return_Product [label="Return Product"
)
// Uses
edge [arrowhead=none, color=black, style=bold, fontsize=9] ;
Customer -> create_account
Customer -> game_login
Customer -> movie_login
Admin -> check_status
Customer -> buy_product
Admin -> Return_Product
// Extensions
(
edge [arrowtail=normal, label="<<extends>>", dir=back]
movie_login -> hibernate_login_extension
buy_product -> audit_authorize_purchase_extension
buy_product -> check_out_extension
create_account -> hibernate_new_customer_extension
create_account -> hibernate_confirm_profile_extension
)
)

```

Use Case Diagram in GraphViz Dot Language



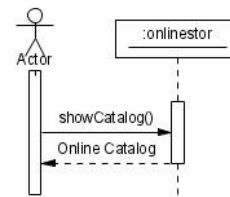
Generated Use Case Diagram

```

umlseq.pic - WordPad
File Edit View Insert Format Help
#header
.PS
copy "umlgraph.pic";
boxwid=1.0;
maxpswid=22;
maxpsht=22;
# define objects
actor(A,"Actor" )
object(Z1,":onlinestor" )
step()
#message body
active(A)
step()
active(Z1)
message(A,Z1,"showCatalog()")
rmessage(Z1,A,"Online Catalog")
inactive(Z1)
step()
#footer
complete(A)
complete(Z1|
.PE

```

Sequence Diagram in UMLGraph Language



Generated Sequence Diagram

Figure 32. Examples of DSL Code for Diagram Generators

To generate the diagram, a two phase generation process is used. First, the metadata is queried and transformed into the appropriate language for the diagram type. This code, the DSL for diagram generators, is stored into the database. On the second phase, the DSL code is queried from the database and written into a file on the file system for input into the diagram generators. This makes the process of generating diagrams generic. To add new diagram types, a generator has to support some textual diagramming language so a transformation program can be created to transform the metamodel data into this language. For example, Figure 33, shows a generator for use case diagrams.

```

-----
procedure gen_use_case_diagram
-----
is
    v_output system.use_case_diagram%TYPE ;
    v_interfaces varchar2(2000) ;
    v_actor_id number ;
begin
    /etc...
    -- add the header
    v_output := 'digraph example {' || crlf ;

    /etc...
    -- generate list of actors
    v_output := v_output || '// Actors' || crlf ;
    v_output := v_output || '{' || crlf ;
    v_output := v_output || 'node [shape=custom, shapefile="Actor.png",' || crlf ;
    v_output := v_output || '    width=.5, height=0.77, fixedsize=true, ' || crlf ;
    v_output := v_output || '    color="#ffffff", label="\n\n\n\n\n\n\n\n\n\n"' || crlf ;
    for gen_actors in ( select actor from actor where actor_id > 0 )
    loop
        v_output := v_output || gen_actors.actor || crlf ;
    end loop ;
    v_output := v_output || '}' || crlf ;
    v_output := v_output || '' || crlf ;

    -- generate list of use cases
    v_output := v_output || '// System Use Cases' || crlf ;
    v_output := v_output || 'subgraph clusterSystem' || crlf ;
    v_output := v_output || '{' || crlf ;
    v_output := v_output || 'label="System"; // Name your system here ' || crlf ;

    /etc...
    for gen_use_cases in ( select use_case_id, name from use_case )
    loop
        v_output := v_output || gen_use_cases.use_case_id
            || ' [label="' || gen_use_cases.name || '"]' || crlf ;
    end loop ;
    v_output := v_output || '}' || crlf ;
    v_output := v_output || '' || crlf ;
    /etc...
end ;

```

Figure 33. Example DSL Code Generator for Use Case Diagrams (partial view)

Once the code for the diagram is ready, it will be used when a user requests to view the diagram. At that point, the code queried from the database and passed on to the diagram generator to create the diagram (dynamically) for delivery to the web browser. This collaboration is shown below in Figure 34.

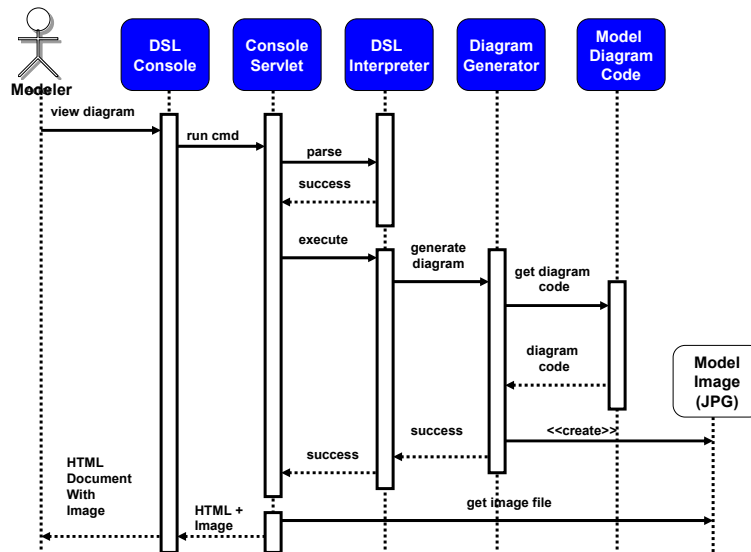


Figure 34. View Diagram Collaboration

Document generation is done using standard XML to HTML techniques via the XSLT language. The main framework used to enable this is the Oracle XDK toolkit which includes a Java library and an XDK Servlet. The Oracle XDK library converts SQL queries into XML documents and passes this data to the XDK Servlet which applies a presentation Stylesheet (in XSLT). This process is illustrated in Figure 35 and example code (XDK page and XSLT) for viewing use case documents is shown below in Figure 36.

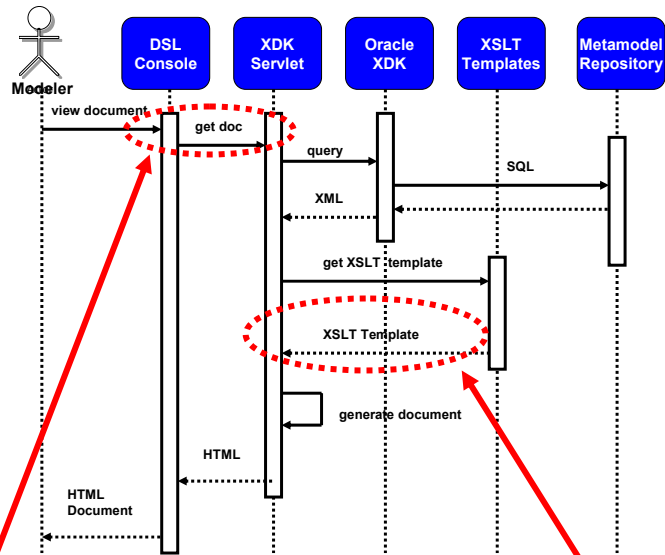


Figure 35. View Document Collaboration

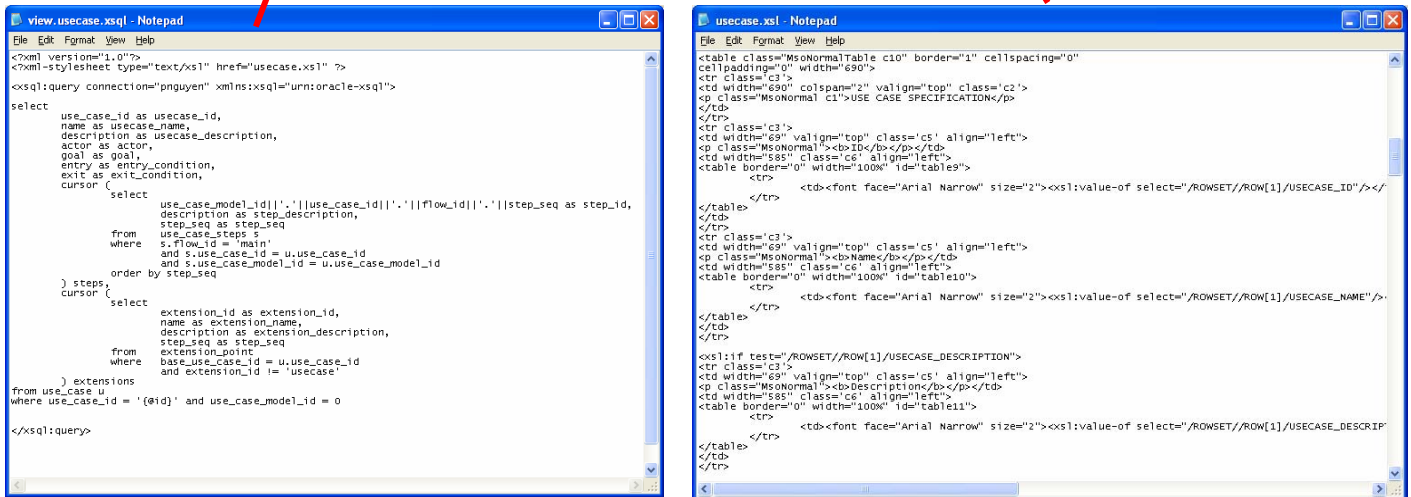


Figure 36. View Use Case XDK Page and XSLT Template

6.4 Modeling & Simulation

Since all diagrams and documentation (use cases) are dynamically generated each time the model is changed by the modeler, the diagrams and documents are updated automatically. Beyond modeling and diagram generation, the research prototype also supports simulation. The general

collaboration is shown in Figure 37. A sequence diagram is generated one time step at a time as the modeler interactively clicks the “next” message button. As messages are evaluated, the simulator also checks to see if a Class in the JVM exists with the call signature of the requested message. If so, the simulator calls the method and renders the result of the call in the next diagram. Messages are in the format of *UMLGraph* sequence diagrams stored in the simulation state area of the database. The data model of the simulation state tables is shown in Figure 38.

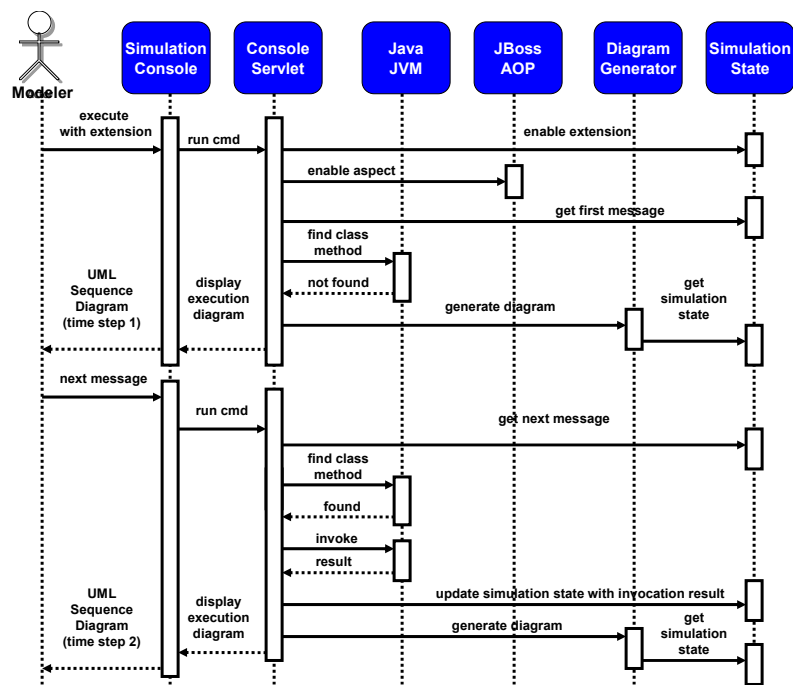


Figure 37. Simulation Collaboration (Sequence Diagram)

At the start of a simulation session, the simulator queries the metamodel repository for the use case model and sequence diagrams for base use cases and extension use cases (if chosen). This information is used to form an “execution plan” and stored as a series of messages in the *Execution_Messages* table. These messages are used to generate the *UMLGraph* code for the sequence diagram.

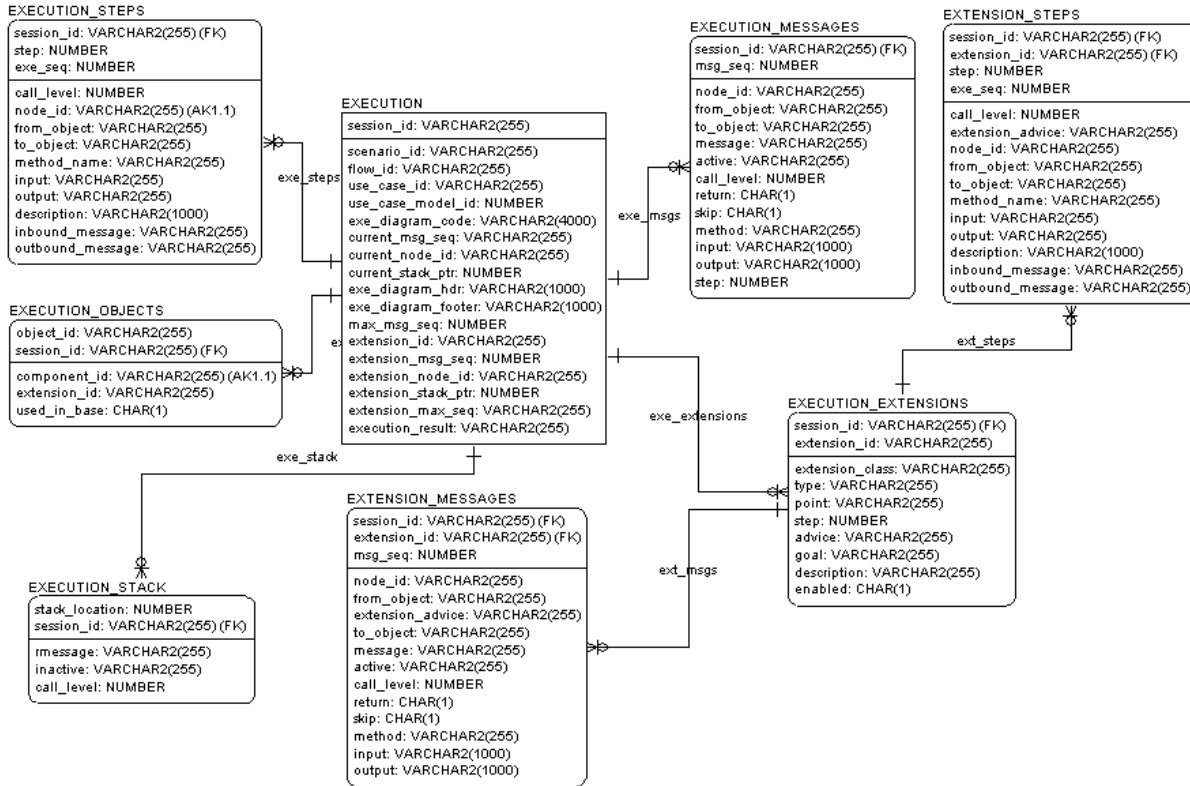


Figure 38. Data Model for Execution State Tables

For example, executing the following command:

execute use case buy_product with extension check_out_extension

Generates the following execution plan (partial view):

MSG_SEQ	NODE_ID	FROM_OBJECT	TO_OBJECT	MESSAGE	ACTIVE	CALL_LEVEL	RETURN	SKIP	METHOD
1	1.1	Actor	OnlineStore	message (A,Z1,"showCatalog()")	active(Z1)	1	N	N	showCatalog
2				rmessage(Z1,A,"Online Catalog")	inactive(Z1)	1	Y	N	
3	3.1	Actor	OnlineStore	message (A,Z1,"selectProducts (productList)")	active(Z1)	1	N	N	selectProducts
4				rmessage(Z1,A,"void")	inactive(Z1)	1	Y	N	
5	5.1	Actor	OnlineStore	message(A,Z1,"checkOut ()")	active(Z1)	1	N	N	checkOut
6				rmessage(Z1,A,"Check Out Screen with Shipping Information")	inactive(Z1)	1	Y	N	
7	7.1	Actor	OnlineStore	message (A,Z1,"enterShipping (address, deliveryType)")	active(Z1)	1	N	N	enterShipping
8	7.1.1	OnlineStore	ShippingService	message (Z1,Z2,"checkPrice (products, fromAddress, toAddress, deliveryType)")	active(Z2)	2	N	N	checkPrice

The *Execution* table tracks the current message (i.e. time step) of the sequence diagram. It also holds fragments of the *UMLGraph* code for the diagram. For example, after a couple of time steps, the *Execution* table contains the following:

USE_CASE_ID	USE_CASE_MODEL_ID	EXE_DIAGRAM_CODE	CURRENT_MSG_SEQ
buy_product	0	#message body active(A) step() active(Z1) message (A,Z1,"showCatalog()") rmessage (Z1,A,"Online Catalog") inactive(Z1) step()	2

The *Extension_Messages* table holds messages for use case extensions which will be inserted into the *Execution_Messages* table when appropriate. That is, the simulator understands aspect-oriented joinpoints, pointcuts, and advice as declared by the *Create Use Case Extension* command. The simulator also updates output messages in the *Execution_Messages* table after the invoking of a real Java method. The simulator blends model generation with Java Execution. One feature the simulator implemented is the handling of errors. If an error occurs, the simulator knows to bypass all subsequent messages and render the error back to the calling object all the way up the chain. It is possible to catch these errors and handle them in the simulation, but the research prototype did not implement this feature.

Java Classes can be hand coded or a boilerplate code for Classes from the model generated by the research prototype. If the code is generated, the prototype adds a *NotImplemented* Java annotation to each method to signal to the simulator that this method has not been implemented. If removed from the code, the method will be included in the call signature searches during a simulation session. For example, the code generated for the *AuditLog* class is shown below.

```
package app.java ;
public class AuditLog
{
    @NotImplemented
    public String createLogEntry( String creditCard, String purchaseAmount )
    {
        return null ;
    }
}
```

Figure 39. Sample Code Generated by Research Prototype

7. Case Studies

7.1 Overview of Case Studies

Three Case Studies were co-developed along with the use case language and model simulators to both guide the direction of the research as well as validate the main ideas behind interactive modeling and simulation. The first case study (*Case Study A: Active Use Case Documents*) uses a simple use case and follows the process of creating and modeling the use case as active documents and models. The second case study (*Case Study B: Black Box Systems Integration via Web Services*) focuses on active modeling and simulation, and attempts to create a simulation component that acts a client to a live web service on the internet. For this case study, the Cybersource Credit Card Payment processing system was used and the use case from *Case Study A* was further enhanced with this payment service component. In the third case study (*Case Study C: Refactoring Database Access Code to the Hibernate Framework*), an existing body of source code from a multi-semester-multi-team student project was used. This case study exhibits real life code maintenance and evolution issues and was tackled as a coarse-grain refactoring problem. A new database persistent framework (Hibernate) was introduced into the existing architecture and the gradual migration to this framework explored with the existing source code. Support for the migration was explored using active models, simulation, and the dynamic “hot deployment” feature of JBoss AOP.

7.2 Case Study A: Active Use Case Documents

7.2.1 Use Cases in Current UML Tools

Six UML tools were surveyed for their support of use case specifications. All of the tools provided some support via one or more text fields that are later used for document generation. For example, in Figure 40, MagicDraw provides numerous fields for specific elements of a use case specification (i.e. pre condition, multiple flows), while the others (Together, Omondo plugin for Eclipse, and Rational Software Modeler) all provide only one or two text fields to document the entire use case specification. The only link to UML models these tools support is an association with the use case in the Use Case Diagram.

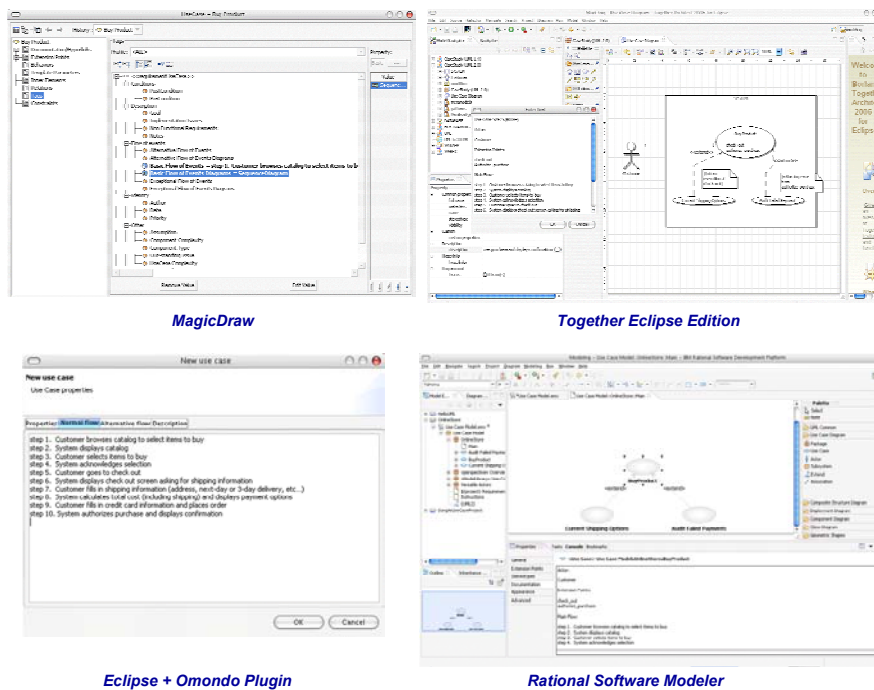


Figure 40. Basic Support for Documenting Use Cases

Two of the six tools, Poseidon and Oracle Developer 10g as shown in Figure 41, provide a rich HTML editing environment. Surprisingly, Oracle Developer 10g, which is more a programming tool than a

UML modeling tool, provides the best support with automatic inclusion of hyperlinks to actors and related use case extensions.

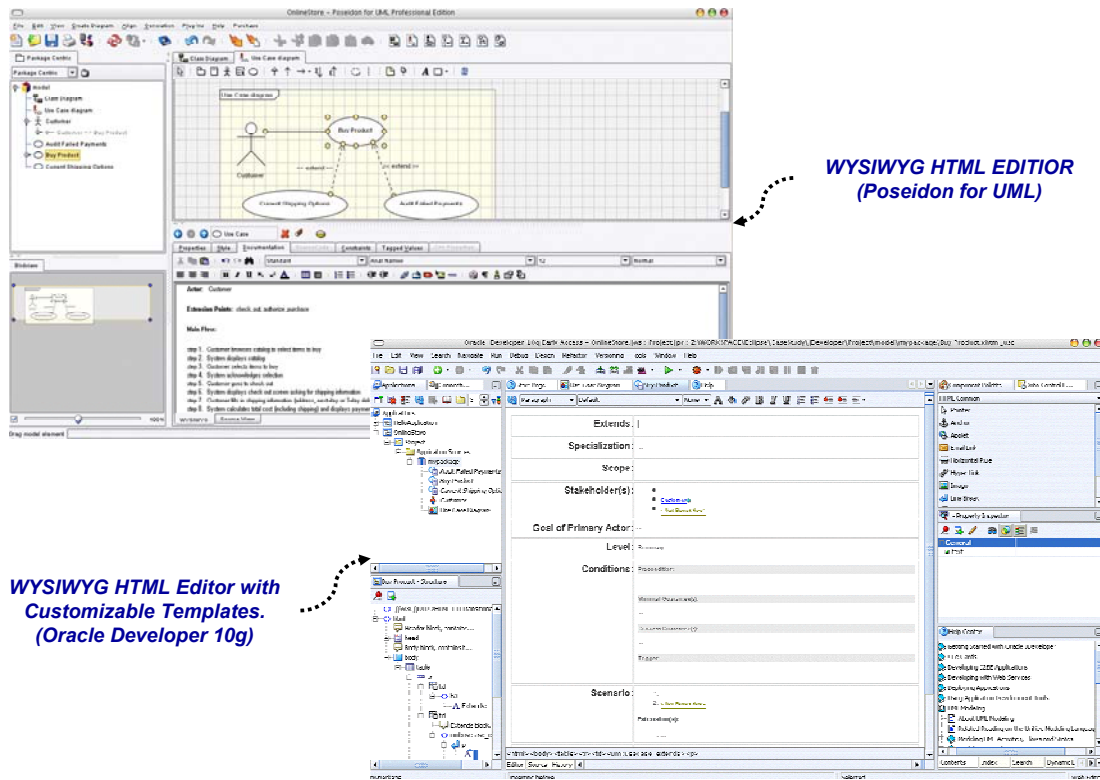


Figure 41. WYSIWYG HTML Editor Support For Use Cases

This short survey of UML tools shows their documentation centric nature which attempts to encourage writing use case specifications closer to the UML models for document generation with the UML diagrams. While this helps keep documentation in sync with the models, the tools currently provide little assistance in managing the associations with UML models and therefore provide little support for requirements traceability.

7.2.2 Programming Use Cases

Programming UML models with a textual language has not been a popular approach in the industry currently dominated by graphical modeling tools. While visual diagrams appeal to the

cognitive and pattern matching abilities of the human mind, they also have their drawbacks [52]. With large scale models, for example, where it is common to see thousands of modeling elements on a canvas, the drawing medium on a computer screen quickly becomes unusable and the model incomprehensible. As a matter of fact, UML was not designed only for graphical tools [65]. It is possible to implement a textual programming language that could be parsed into metadata stored in a repository. The UML standard, defines such a meta-model – the Meta Object Facility (MOF) [66]. Using a small example taken from the *UML Distilled* book [67] (Figure 42).

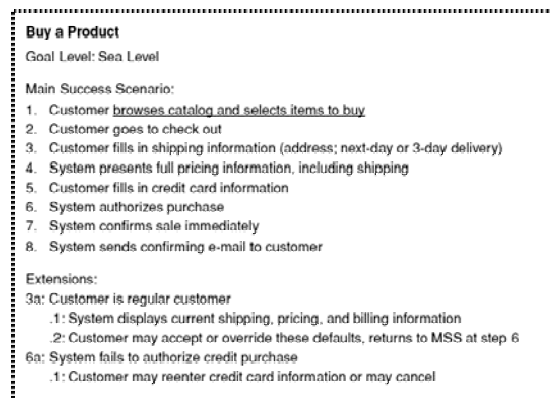


Figure 42. Buy Product Use Case Specification Document (from [67])

The example code below is the equivalent specification for the Buy Product Use Case described in Figure 42:

```

create use case buy_product
  step 1 "Customer browses catalog to..."
  step 2 "System displays catalog"
  step 3 "Customer selects items to buy"
  step 4 "System acknowledges selection"
  step 5 "Customer goes to check out"
  step 6 "System displays check out screen ..."
  step 7 "Customer fills in shipping ..."
  step 8 "System calculates total ..."
  step 9 "Customer fills in credit ..."
  step 10 "System authorizes purchase ..."

with
  actor "Customer"
  goal "Buy Product(s)"
  extension check_out at step 5
  extension authorize_purchase at step 9
  
```

Figure 43. Buy Product Use Case Specification using a Declarative Language

At first glance, with the exception of a few key words, the sample code does not differ a great deal with the use case specification document in Figure 42. Behind the scenes, however, this code was parsed into metadata stored into a repository. As a result, the metadata can be used to automatically produce HTML documentation and a UML system sequence diagram. Figure 44, below, shows the generated HTML document and diagram.

The screenshot shows a Microsoft Internet Explorer browser window displaying a web page titled "Customer wants to buy a product from the online store". The page content is structured as follows:

Actor	Customer																						
Goal	Buy Product(s)																						
Main Flow	<table border="1"> <thead> <tr> <th>Flow Step</th> <th>Description</th> </tr> </thead> <tbody> <tr><td>Step 1</td><td>Customer browses catalog to select items to buy</td></tr> <tr><td>Step 2</td><td>System displays catalog</td></tr> <tr><td>Step 3</td><td>Customer selects items to buy</td></tr> <tr><td>Step 4</td><td>System acknowledges selection</td></tr> <tr><td>Step 5</td><td>Customer goes to check out</td></tr> <tr><td>Step 6</td><td>System displays check out screen asking for shipping information</td></tr> <tr><td>Step 7</td><td>Customer fills in shipping information (address, next-day or 3-day delivery, etc?)</td></tr> <tr><td>Step 8</td><td>System calculates total cost (including shipping) and displays payment options</td></tr> <tr><td>Step 9</td><td>Customer fills in credit card information and places order</td></tr> <tr><td>Step 10</td><td>System authorizes purchase and displays confirmation</td></tr> </tbody> </table>	Flow Step	Description	Step 1	Customer browses catalog to select items to buy	Step 2	System displays catalog	Step 3	Customer selects items to buy	Step 4	System acknowledges selection	Step 5	Customer goes to check out	Step 6	System displays check out screen asking for shipping information	Step 7	Customer fills in shipping information (address, next-day or 3-day delivery, etc?)	Step 8	System calculates total cost (including shipping) and displays payment options	Step 9	Customer fills in credit card information and places order	Step 10	System authorizes purchase and displays confirmation
Flow Step	Description																						
Step 1	Customer browses catalog to select items to buy																						
Step 2	System displays catalog																						
Step 3	Customer selects items to buy																						
Step 4	System acknowledges selection																						
Step 5	Customer goes to check out																						
Step 6	System displays check out screen asking for shipping information																						
Step 7	Customer fills in shipping information (address, next-day or 3-day delivery, etc?)																						
Step 8	System calculates total cost (including shipping) and displays payment options																						
Step 9	Customer fills in credit card information and places order																						
Step 10	System authorizes purchase and displays confirmation																						
Extensions	<table border="1"> <thead> <tr> <th>Extension Point</th> <th>Description</th> </tr> </thead> <tbody> <tr><td>check_out</td><td>System displays check out screen with shipping information (<i>Extension point defined at step 5</i>)</td></tr> <tr><td>authorize_purchase</td><td>System authorizes purchase (<i>Extension point defined at step 9</i>)</td></tr> </tbody> </table>	Extension Point	Description	check_out	System displays check out screen with shipping information (<i>Extension point defined at step 5</i>)	authorize_purchase	System authorizes purchase (<i>Extension point defined at step 9</i>)																
Extension Point	Description																						
check_out	System displays check out screen with shipping information (<i>Extension point defined at step 5</i>)																						
authorize_purchase	System authorizes purchase (<i>Extension point defined at step 9</i>)																						

Below the table is a UML System Sequence Diagram showing the interaction between the Customer and the System:

- Customer:** Initiates Step 1 (browses catalog), Step 3 (selects items), and Step 5 (goes to check out).
- System:** Responds to Step 2 (displays catalog), Step 4 (acknowledges selection), Step 6 (displays check out screen), Step 8 (calculates total cost), and Step 10 (authorizes purchase).

Figure 44. Generated HTML Use Case Document

Furthermore, update commands for the use case also support adding and/or removing individual use case steps, extension points, or other items such as pre-conditions and post-conditions. If the use case was elaborated into design models, then the tool can warn the modeler of potential problems or prevent the change.

7.2.3 A Joinpoint Model for Use Cases

The Use Case Specification language was also designed to express aspects as use case extensions with a defined join-point model. For example, the example code below (Figure 45) declares semantics for a credit card authorization failure extension that will only fire when a customer enters an invalid credit card.

```
create use case extension audit_authorize_purchase_extension
with
  goal = "Audit authorization failures"
returning error
advice steps
  step 1 = "Get authorization context and create a log..."
  step 2 = "void return (don't change base behavior)"
advice scenario messages
  1.1 from Extension to AuditLog
    requesting createLogEntry with "creditCard",
      "purchaseAmount"
returning "void"
extending
  buy_product at authorize_purchase returning error
```

Figure 45. Example of a Use Case Extension Declaration

A sequence diagram fragment (UML interaction frame) is automatically generated from the metadata produced from the create extension command. This is shown below in Figure 46, which demonstrates the “returning error” join point.

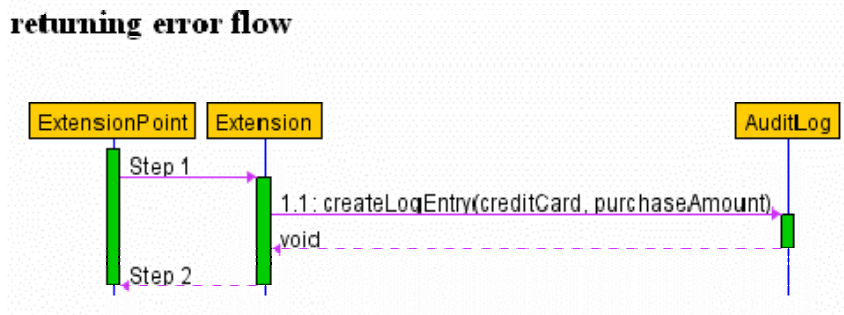


Figure 46. Example of Use Case Extension Sequence Diagram Interaction Frame

The Use Case Specification language supports the following join point model for use case extensions.

```

(<extension point>)? around|after|before execution
(<extension point>)? returning error
(<extension point>)? returning success

```

Figure 47. A Jointpoint Model for Use Cases

The partial grammar above (Figure 47) defines join points for use case step execution, return with success, and return with error. If the extension point is omitted, then the join point is defined at the use case level rather than at the use case step associated with the extension point. Note, as recommended in [3], the execution step for the extension point is defined in the base use case rather than the extension use case. This provides more flexibility as changes in the base use case can be done without impacting the definition of extension use cases.

7.2.4 Interactive UML Diagrams

Sequence diagrams can be modeled interactively, one message at a time as shown below in Figure 48, which defines three messages for the *Buy Product* base use case. While this may seem an overkill to code these messages compared to a point-and-click approach of graphical UML tools, the payoff comes with the simulator where a sequence diagram can be single-stepped through its timeline and rendered interactively.

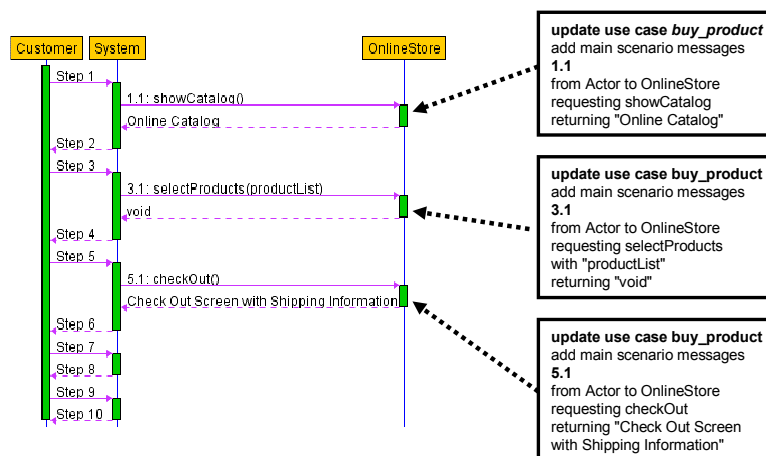


Figure 48. Adding Sequence Diagrams to Use Cases

While modeling sequence diagrams, the tool can also track the relationship of the use case with the domain objects by associating the messages and objects with the use case. As a result, class diagrams can be generated to show this relationship as shown in Figure 49. The class *OnlineStore* is shown to implement a set of methods as dictated by its participation in the *buy_product* use case collaboration.

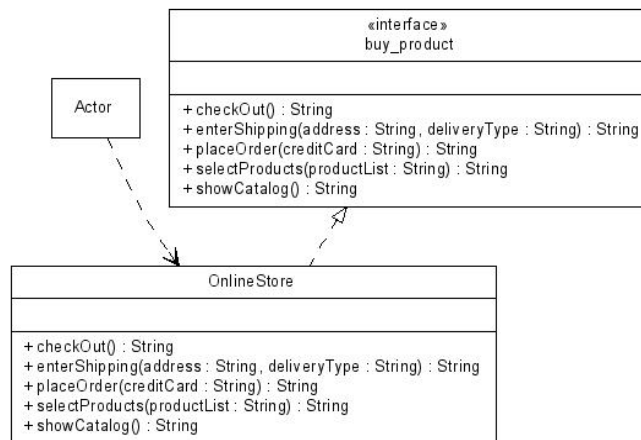


Figure 49. Generated Class Diagram From Use Case Metadata

7.2.5 Execution Paths in Aspect Based Models

Figure 51 shows the simulation screen of the execution of a sequence diagram with an extension enabled. A modeler can walk forwards or backwards in time and observe the effects of use case extensions. To enable an extension, a modeler specifies the extension as an option to execution in a command as follows:

```
execute use case buy_product with extension check_out_extension
```

Figure 50. Example Use Case Execution Command with an Extension

If the extension is statically bound (i.e. defined without any guard conditions) then the simulator will incorporate the interaction frame defined for the extension into the base interaction and display this to the modeler. On the other hand, if the extension is dynamically bound, such as with a “returning error”

type, then the simulator will only incorporate the extension’s interaction frame if the condition is detected. Currently, the only way to trigger such a condition is to implement the behavior of the class in Groovy [70] or Java and have the simulator invoke the code which causes the error.

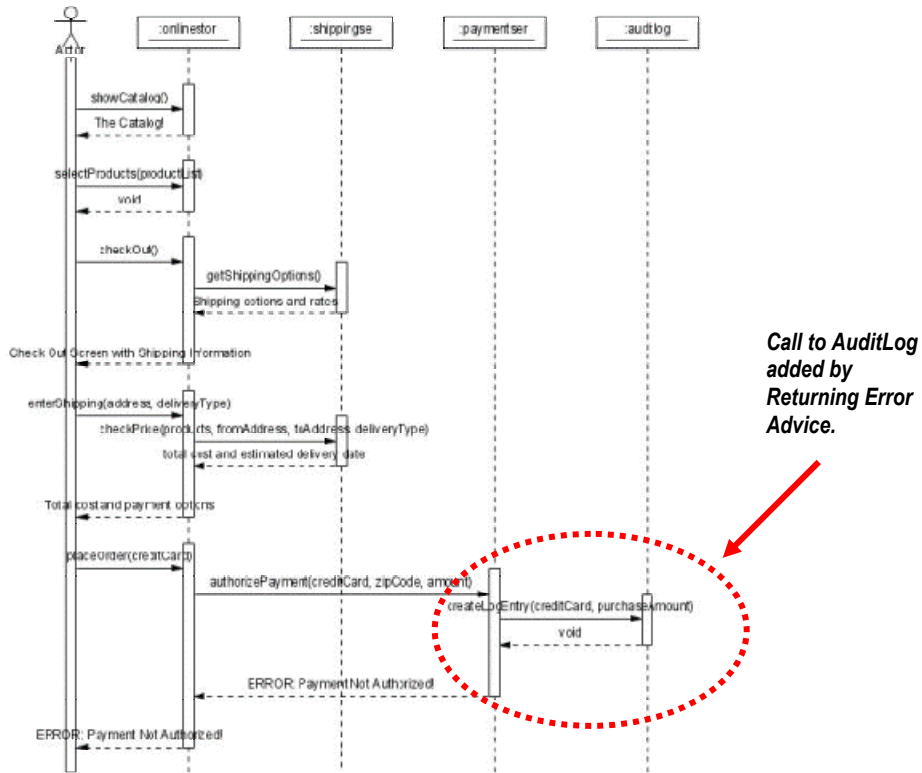


Figure 51. A Simulation Session with a Returning Error Advise Enabled

7.2.6 Adding Behavior with Groovy Scripts

Using the Groovy scripting language [70], which is integrated with the simulation environment, a modeler can add behavior to classes interactively and invoke the methods during a simulation session. This enables the modeler to explore exceptional conditions and alternative flows without having to first model all the possible paths. That is, a modeler can just start with the main success path and then add arbitrary error conditions later with code. Figure 52 below shows the Groovy code editing screen for the *AuditLog* class.

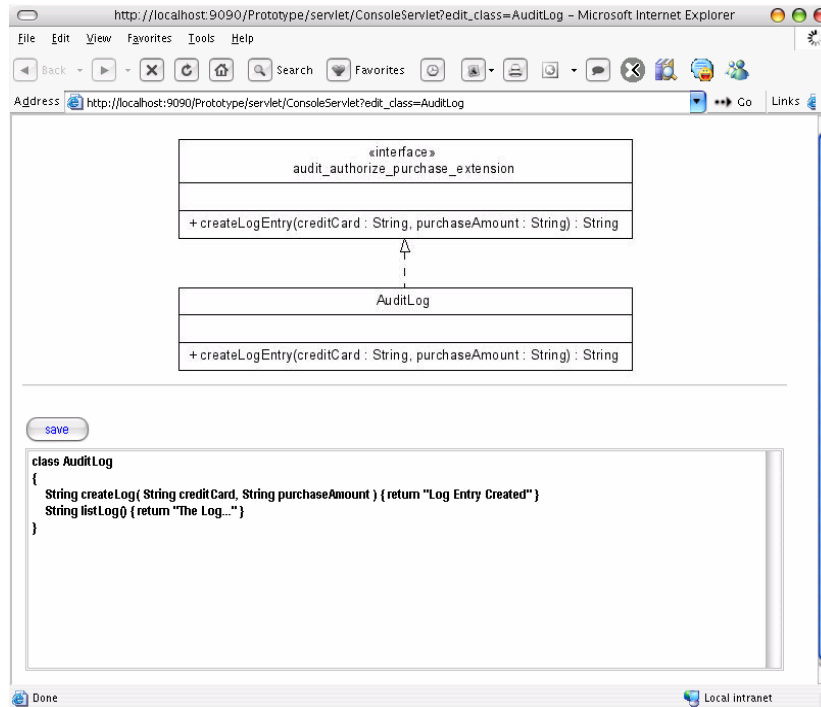


Figure 52. Groovy Code Editor

7.2.7 Integrating Java Code

The simulation system can also detect the existence of a Java class using reflection and invoke the methods if there is a match in call signature. With this ability, legacy code can be wrapped with Java classes, or if the application is already in Java, then the simulator can work with the code directly. In the current research prototype, however, the simulator only supports String types. The main reason for this is due to the user interface's limited ability to render different object types for input data to a Java method invocation. This input request currently is in the form of a single input field which expects a comma separated list of strings. For example, with the following Java code below which implements the `authorizePayment` method for the `PaymentService` class, the simulator will display the screen as shown in Figure 53.

```

public class PaymentService {
String authorizePayment( String creditCard,
String zipCode, String amount )
{
if ( "1234567890".equals( creditCard) &&
"11111".equals(zipCode) )
return "Payment Authorized!" ;
else
return "ERROR: Payment Not Authorized!" ;
}
}

```

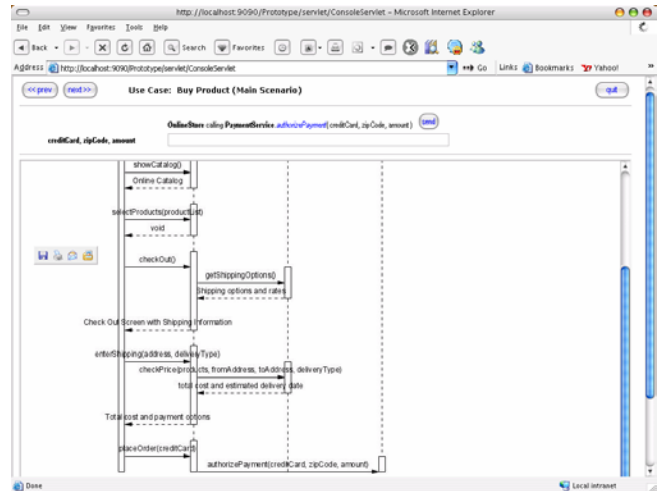


Figure 53. Simple Input Dialog for a Java Method

7.2.8 Case Study Summary

The results from this case study reveals that Active Documents as explored using a declarative language for *Use Case Specification* provides better support for requirements analysis and change management. This is primarily achieved using dynamic documentation generation and a model simulation environment which puts Active Models at the center of the analysis work. Compared to existing documentation centric *Use Case Specifications* methods as supported by existing techniques and UML tools, an Active Document and Active Model provides a more direct association to the working software and can be more easily maintained and leveraged for system evolution. Furthermore, incorporation of *Early Aspects* into the *Use Case Specification* language enhances the value of Active Documents and Active Models by rendering the effects of applying aspects to a base scenario in the sequence diagram.

7.3 Case Study B: Black Box Systems Integration via Web Services

In the past several years, web services and Service-Oriented Architecture (SOA) have been gaining in popularity and adoption. Many online business systems integrate with external web services as “black boxes”, incorporating their features seamlessly into internal business processes. Often, during the requirements analysis phase of a new integration project, these services are readily available for use. In essence, parts of the “solution space”, as represented by the web services are already built, but understanding how to use them and incorporating their features into a UML model is difficult. This case study looks at one such web service, the Cybersource Credit Card Payment service, and attempts to integrate the service into an interactive UML model (i.e. the sequence diagram for the *Buy Product* use case).

7.3.1 The API

Integration often starts with an Application Programmer Interface (API). Understanding the API requires code level exploration with test client programs. Many service providers make available sample client code which demonstrates how to call the web service to assist developers in quickly coming up-to-speed with integration efforts. For example, in the Cybersource SDK for Java [73] details are available on setup, usage, sample code, and test transactions. In addition, Cybersource makes available numerous API choices and simulated transaction processors for testing prior to go live. Figure 54 shows the API’s available from the Cybersource website as of January, 2006.

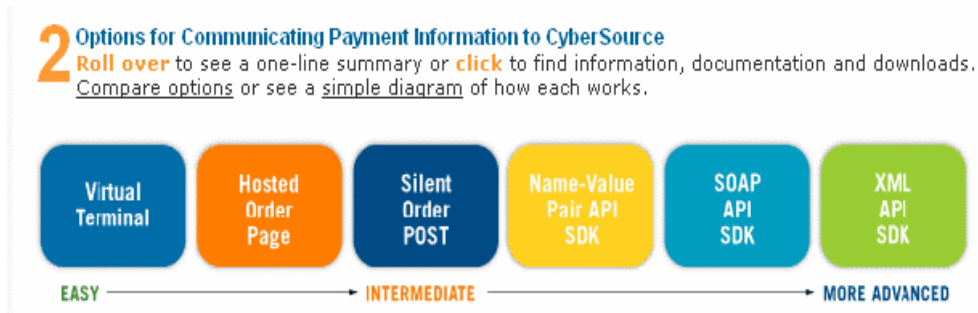


Figure 54. Options and API for Integrating with Cybersource [74]

7.3.2 The Java Code

In the previous case study (Section 7.2.7), a sample Java class, the *PaymentService*, was demonstrated with “hard-coded” logic to respond with a “success” or “failure” from incoming messages on the *authorizePayment* method. In this case study, the class is re-written to act as a client module to the Cybersource web service. The new code for this class is shown below in Figure 55.

```
public class PaymentService {
    public String authorizePayment(String creditCard, String zipCode,
        String amount) throws Exception {
        ICSCClient client = new ICSCClient() ;
        ICSCClientOffer offer = new ICSCClientOffer();
        ICSCClientRequest request = new ICSCClientRequest();
        request.setField("ics_applications", "ics_auth");
        request.setField("merchant_ref_number", "007");
        request.setField("merchant_id", client.getMerchantID());
        /* portions omitted ... */
        request.setField("customer_cc_number", creditCard);
        request.setField("bill_zip", zipCode);
        offer.setField("amount", amount);
        request.addOffer(offer);
        ICSReply reply = client.send(request);
        if (reply.getReplyCode() <= 0) {
            return "ERROR: " + ICSEException(reply.getErrorMessage());
        } else {
            return "Transaction succeeded";
        }
    }
}
```

Figure 55. Payment Service Java Code for Calling Cybersource

7.3.3 Results

Two simulation runs were done with different transaction inputs for the authorize payment message. As documented on the Cybersource website a transaction amount of \$1500 will simulate a transaction error. For the two simulation runs, the following inputs were used:

- For the success test: **Credit Card** = 4111111111111111, **Zip code** = 95130, **Transaction Amount** = 125.00.
- For the error test: **Credit Card** = 4111111111111111, **Zip code** = 95130, **Transaction Amount** = 1500.00.

Figure 56 below shows the results of the call to the Cybersource web services rendered as a return message in the sequence diagram from the call to the *authorizePayment* method.

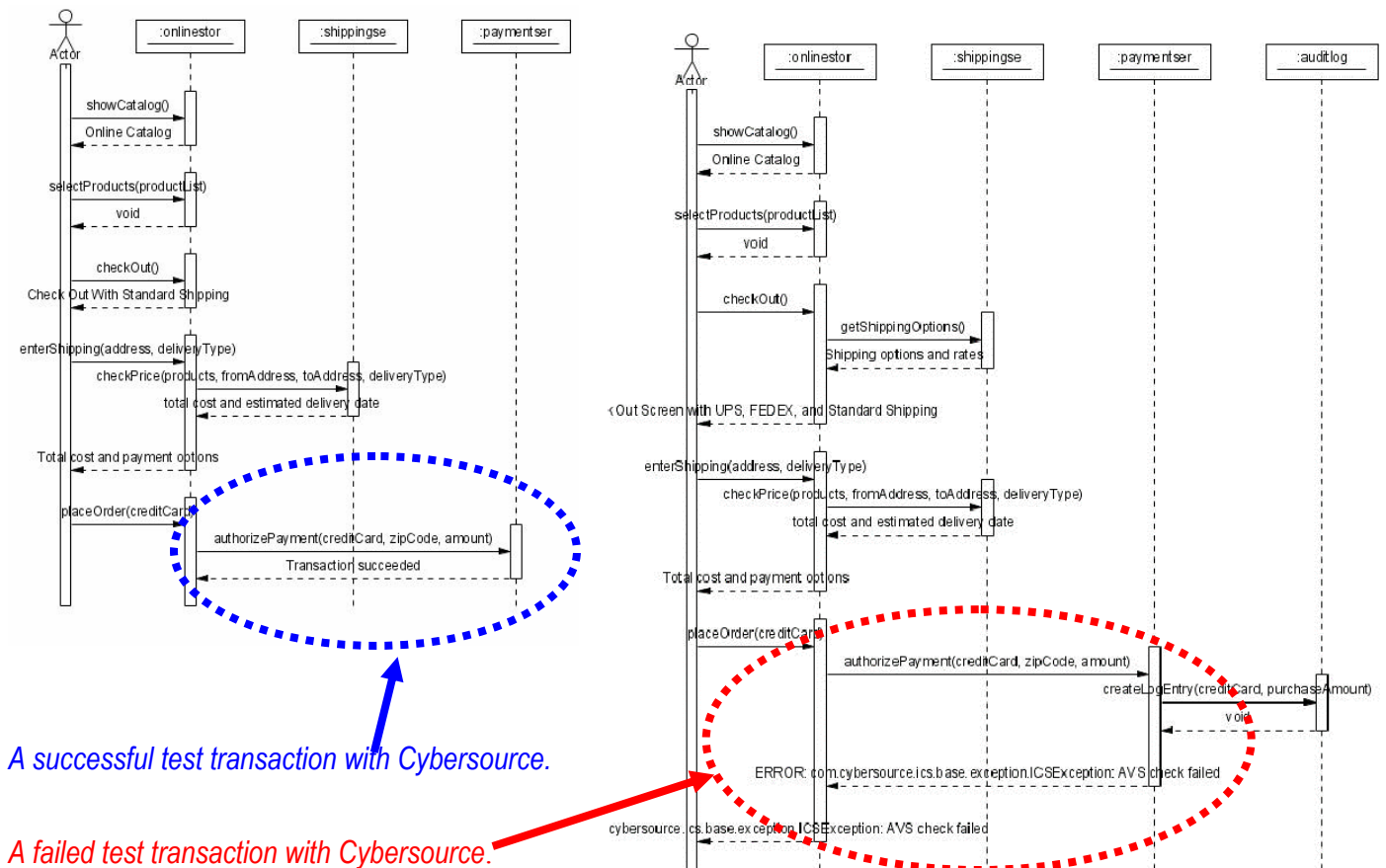


Figure 56. Test Results From Calling Cybersource via Web Services

7.3.4 Case Study Summary

Modeling at many levels of abstractions provides tremendous benefits to the analysis of system behavior based on a composition of a number of subsystem components. In this case study, a subsystem component (an external payment service exposed as a web service) was integrated seamlessly into a modeling environment with minimal effort. In fact, the exact sample code from the service provider was used with minimal changes. Normally, exploring a service API is done only at a code level when a developer is trying to understand and use the API. Making this capability available to a modeler allows for a broader view of the entire software solution. For example, many web services and/or internal systems can be modeled collectively in this manner. This approach supports better interface design and comprehension of end-to-end integration scenarios

Although the case study reveals a promising direction, there are still some limitations that were not addressed by the simulator and further improvements are possible. A more robust data exchange and messaging protocol could be used to integrate the simulator with existing code. For example, in order to understand the difference between an “Error” and a “Successful” result, the simulator scans for the “ERROR” text string within the result message. As such, Java exceptions have to be manually converted into this convention in the code to signal to the simulator that the result should be treated as an error. Furthermore, it was evident that some sort of state management is needed at the model level. One possibility for adding this is with a UML state diagram that tracks the state of the session based on the inputs and outputs of messages from source to target objects. Another shortcoming also is the lack of support for asynchronous messaging. Modeling these types of messages will enable a more robust real time analysis of system properties.

7.4 Case Study C: Refactoring Database Access Code to the Hibernate Framework

In this case study the body of work (source code) from an evolving student project at San Jose State's Computer Engineering Department "*Software Systems Analysis and Design – CMPE 221*" course was used. Unlike other "single semester" software projects, the results of each semesters work are passed on to the next in an iterative style of evolution. Each group of students would study the work of prior semesters from the project documents and source code and then develop enhancements and refactorings to evolve the code base. The results from three such iterations were studied from the Fall 2003 to Fall 2004 groups. The first team focused on enhancing an existing online DVD rental system, similar to the NetFlix DVD Rental service. The second team added online Game Rentals; while the third enhanced the system with Online Advertising.

The main architecture of the application runs on a Java application server (Tomcat) and uses MySQL as the database. In studying the source code, one major issue with the project became apparent. Each team took a different approach to managing persistent and object-to-relational mapping. Two teams took a one-class-has-all-business-objects approach where a huge class with numerous methods for all domain objects was used to convert messages to SQL queries and commands. The other team took the approach on the opposite end of the spectrum, where a small utility class managed basic database access and took as input only SQL queries and commands. In effect, this approach distributes all database access code to the domain objects themselves. Figure 57 shows the class diagrams of the database access code. The inconsistent management of object persistence is a common problem amongst object-oriented applications and many object-relational mapping tools have been developed to address this. For this case study, one such tool, Hibernate, was used to refactor a portion of the database access code.

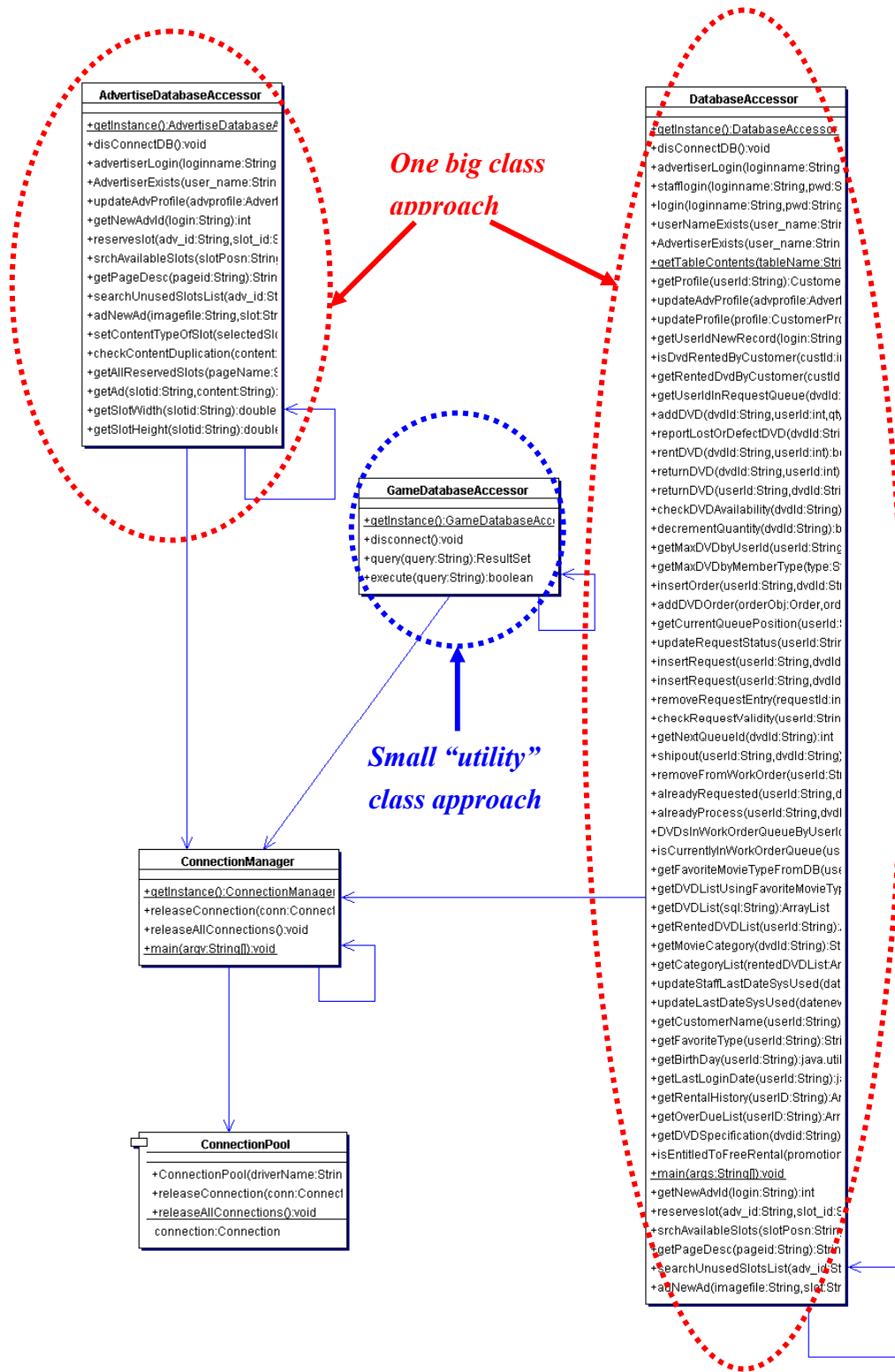


Figure 57. Database Access Classes in CMPE 221 Student Projects

7.4.1 Reverse Engineering To a Use Case

Focusing on the new user registration process for the refactoring, the project documentation was used to explore the design details -- in particular, a dynamic domain model in the form of a sequence diagram. One was not to be found in any of the project documentation. This is probably due to the fact that this part of the system was designed by an earlier team (before the three teams under study).

As a result, a study of the source code was undertaken with the assistance of a dynamically generated sequence diagram. This dynamically generated diagram was developed as part of the research prototype due in part to the need for this feature as called for by this case study. The User Interface screen and resulting sequence diagram from the reverse engineering is shown below in Figure 58 and Figure 59. This reverse engineering effort differs from current UML tool based approaches which generate diagrams from source code. The diagram below in Figure 59 was generated from runtime messages amongst objects in the Java JVM.

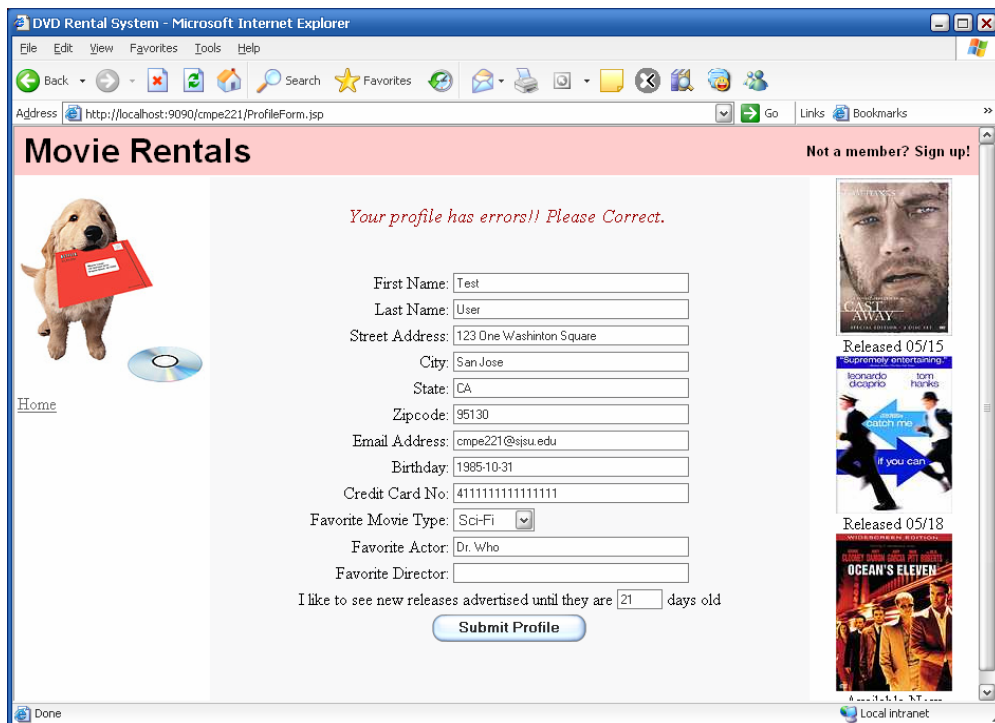


Figure 58. New User Registration Screen

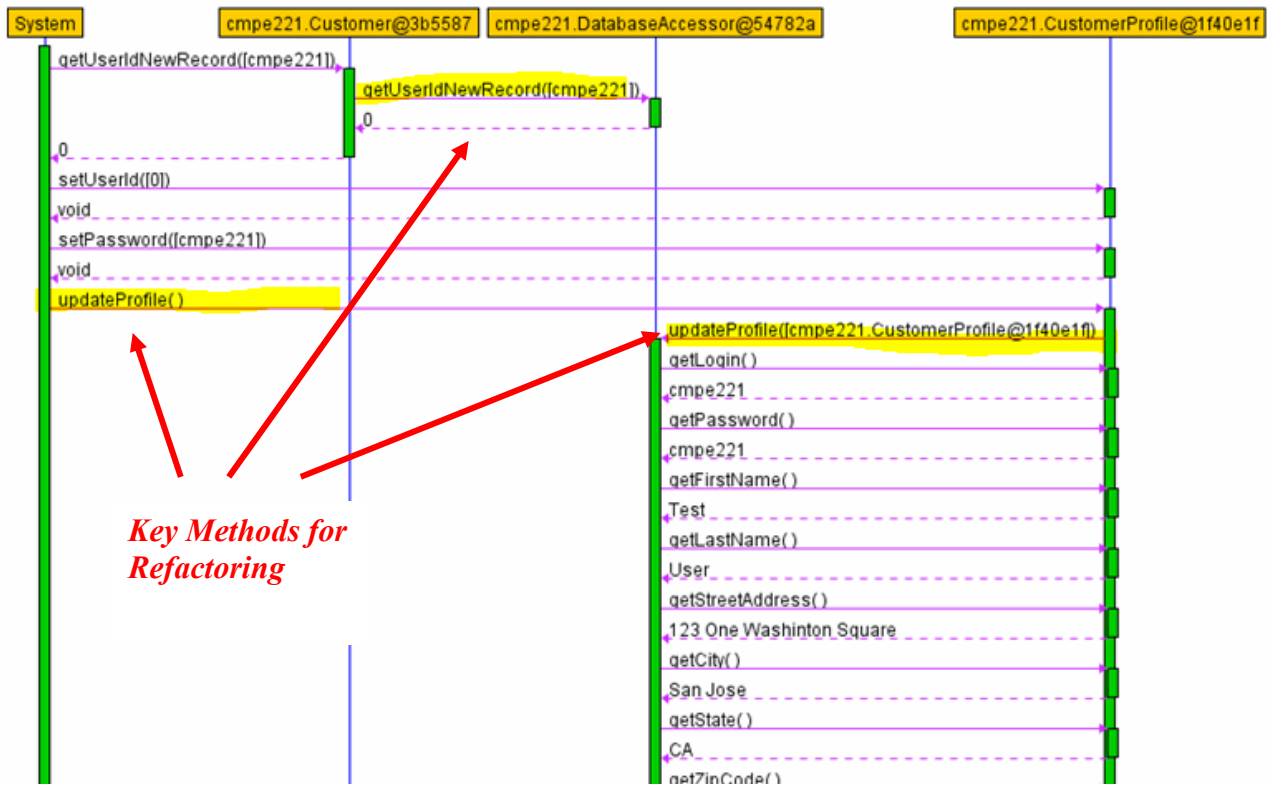


Figure 59. Dynamically Generated Sequence Diagram Trace

The results from the dynamic trace reveals two key classes involved in the new user registration process: the *CustomerProfile* class and the *DatabaseAccessor* class. It was also discovered that the key database table in the MySQL database involved in this transaction is the *Customer* table. To proceed, this information helped form a use case for the modeling environment which would be used to explore the refactoring to the Hibernate framework. Below in Figure 60 is the source code which defines this use case and the HTML documentation it generates.

```

create use case create_account described as "Customer wants to create a new account for the online store"
steps
    step 1 "Customer clicks on create profile link"
    step 2 "System display create account screen asking for new userid and password"
    step 3 "Customer enters login user id and password"
    step 4 "System verifies that the userid has not been taken and displays a profile registration
page"
    step 5 "Customer enters profile information and submits the request"
    step 6 "System display profile confirmation page"
    step 7 "Customer acknowledges the confirmation and submits a request to complete the ..."
    step 8 "System displays a successful registration message"

with
actor "Customer"
goal "Register for a new account"
extension new_record at step 3
    described as "System checks for an existing userid and creates a new record if ... "
extension registration_confirmed at step 7
    described as "System receives confirmation and creates the new account"

```

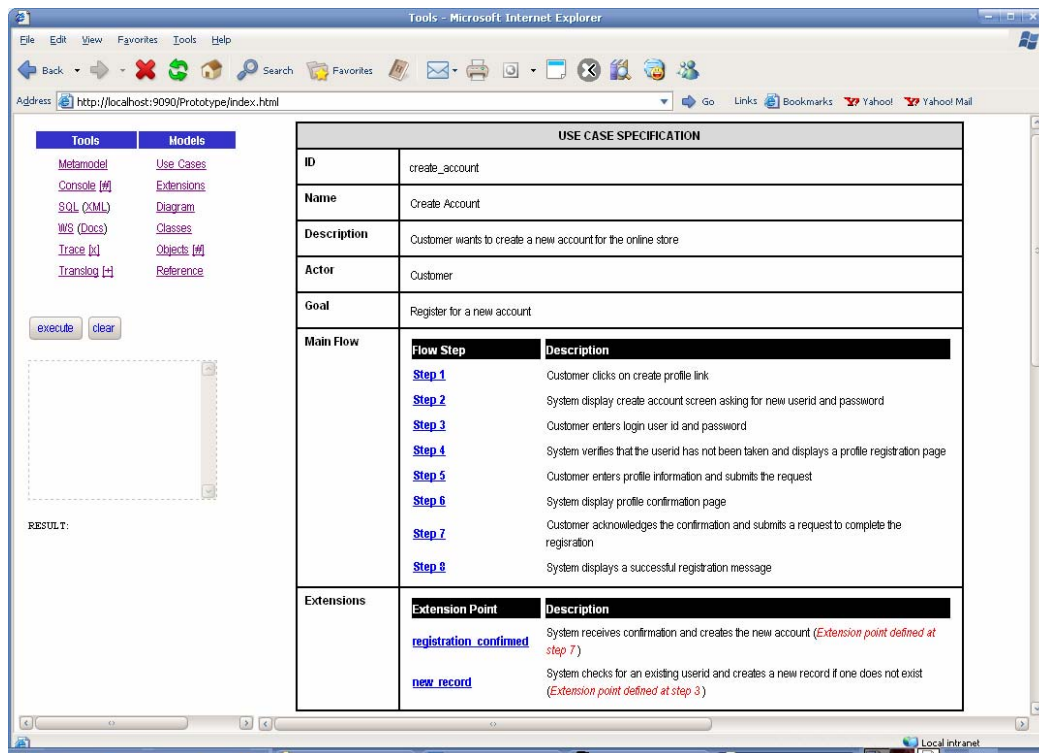


Figure 60. Create Account Use Case for Refactoring Case Study

From the study of how the current system handles the new user registration scenario, two methods on the *DatabaseAccessor* class will be the target for extension and exploration within the active model. The *getUseridNewRecord()* and the *updateProfile()* methods. To prepare for this, two extension points

were introduced into the base use case: *new_record* and *registration_confirmed*. Extension use cases can then be defined to extend the behavior of the current system at these extension points. Two extension use cases were created for this purpose as shown below from the generated use case diagram in Figure 61.

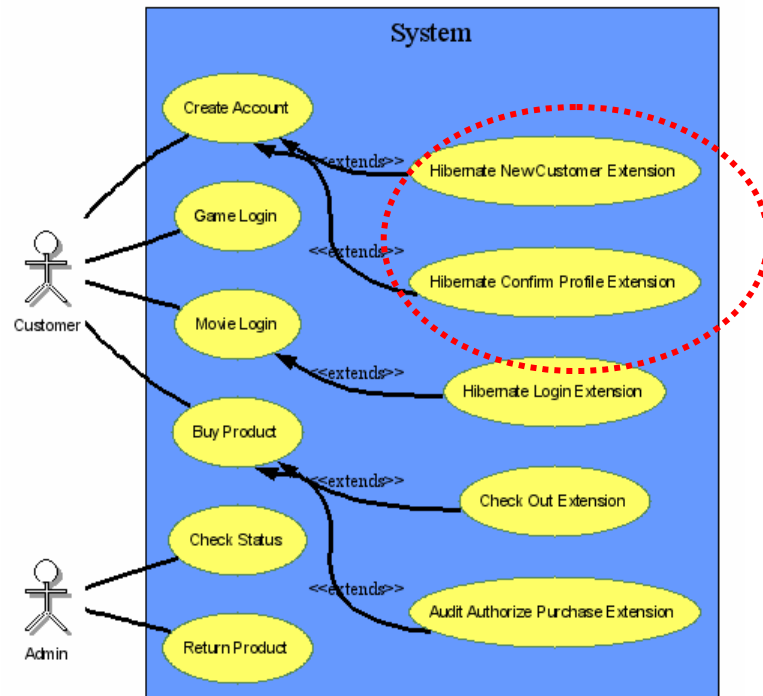


Figure 61. Hibernate Extension Use Cases

The Hibernate extension use cases add behavior for creating a new customer record and updating the customer record, which are two distinct messages within the current system. Additionally, since Hibernate will handle all of the database access activities, an Oracle database will be used instead of the currently used MySQL. This situation will more realistically simulate a real life scenario where a core architecture component (i.e. the Database Server) is changed and the migration of the code base planned in iterations. The next section will discuss the details of the Hibernate component. This part of the case

study leverages Hibernate in a standard way as specified by Hibernate APIs and technical documentation.

7.4.2 Creating a New Hibernate Component to Map the Customer Table

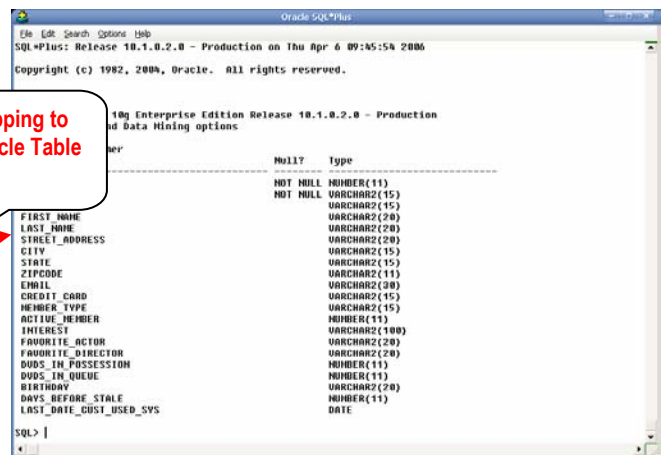
An Oracle database table for the Customer data was created mirroring the logical structure of the table from the MySQL database. Using standard Hibernate tools, a mapping file was created and the Java class for the *Customer* object generated. A partial view of these artifacts is shown in Figure 62.

```

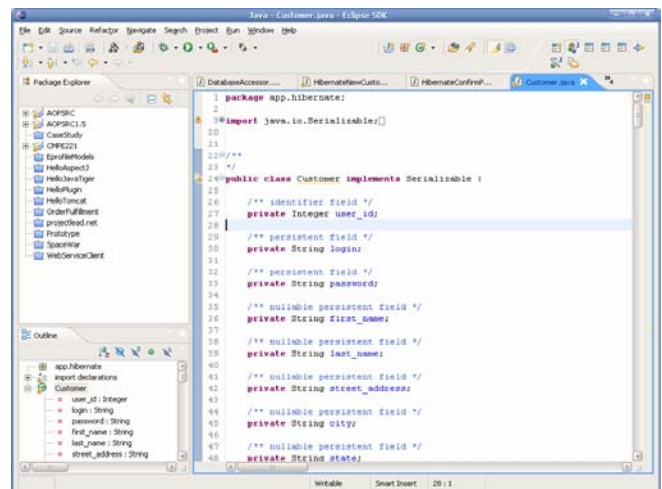
<hibernate-mapping>
<class name="app.hibernate.Customer" table="customer">
<meta attribute="class-description"/>
<id name="user_id" type="int" column="user_id">
<meta attribute="scope-set">protected</meta>
<generator class="increment"/>
</id>
<property name="login" type="string" not-null="true"/>
<property name="password" type="string" not-null="true"/>
<property name="first_name" type="string" not-null="false"/>
<property name="last_name" type="string" not-null="false"/>
<property name="street_address" type="string" not-null="false"/>
Etc...
</class>

<query name="app.hibernate.CustomerByUserId">
<![CDATA[
from app.hibernate.Customer as c
where c.user_id = :uid
]]>
</query>
</hibernate-mapping>

```



Mapping to Oracle Table

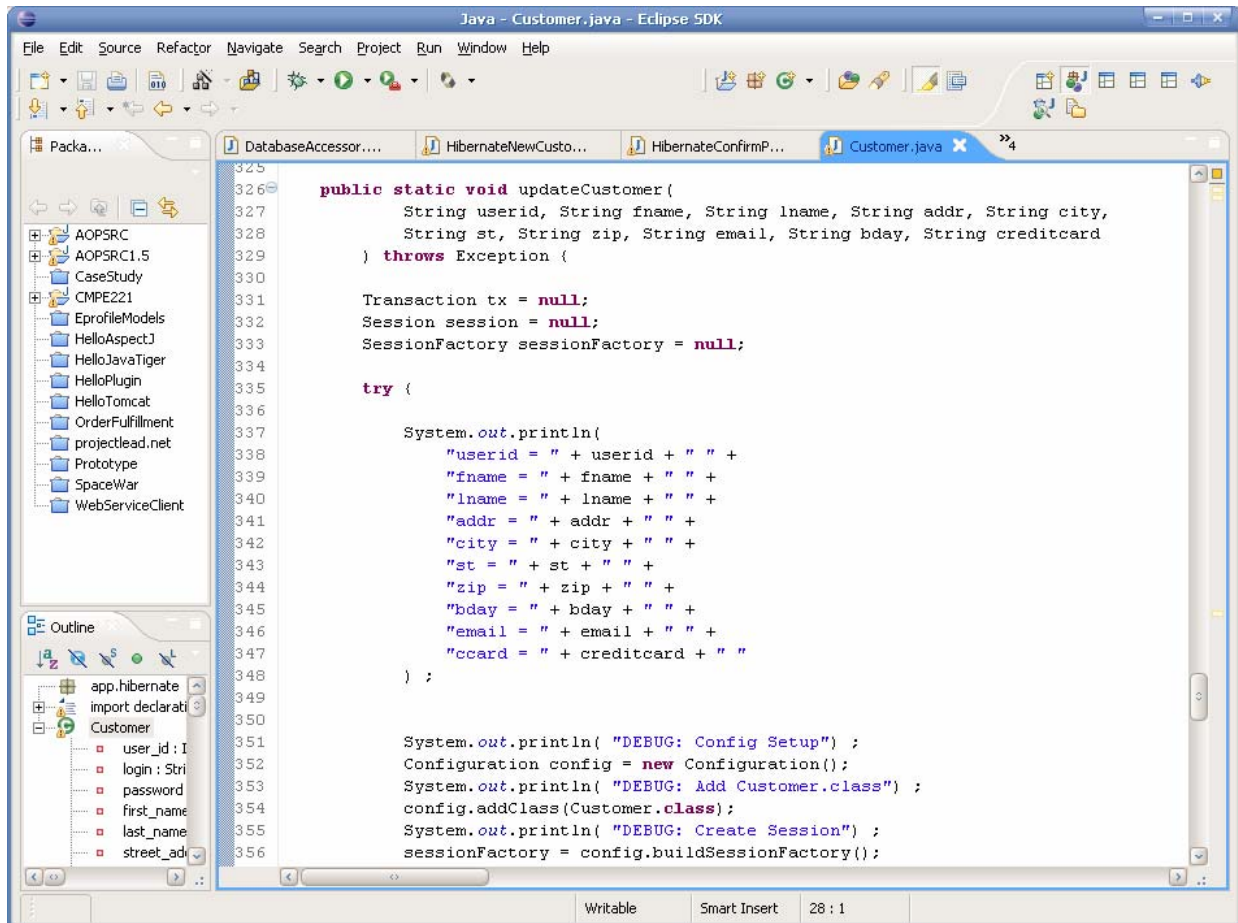


Generate Java Class using Hibernate Tools

Figure 62. Hibernate Mapping and Generated Java Class

Thus, a new Java Class: *Customer* was created for use in an application. Using Hibernate, all SQL queries and commands will be generated by Hibernate or can be customized using the Hibernate

mapping file. One such customization that was done was a query for the Customer object based on Userid (i.e. Key). This was defined in the mapping file as a named query – i.e. *app.hibernate.CustomerByUserid*. A few convenience methods were also added to the Customer object as class “static” methods to simplify the interaction with the simulation environment. An example of this is the “*updateCustomer()*” method which uses the named query to find the Customer object and sets the attributes of the object for update to the database. A partial view of this method is shown in Figure 63.



```
325
326 public static void updateCustomer(
327     String userid, String fname, String lname, String addr, String city,
328     String st, String zip, String email, String bday, String creditcard
329 ) throws Exception {
330
331     Transaction tx = null;
332     Session session = null;
333     SessionFactory sessionFactory = null;
334
335     try {
336
337         System.out.println(
338             "userid = " + userid + " " +
339             "fname = " + fname + " " +
340             "lname = " + lname + " " +
341             "addr = " + addr + " " +
342             "city = " + city + " " +
343             "st = " + st + " " +
344             "zip = " + zip + " " +
345             "bday = " + bday + " " +
346             "email = " + email + " " +
347             "ccard = " + creditcard + " "
348         );
349
350
351         System.out.println( "DEBUG: Config Setup" );
352         Configuration config = new Configuration();
353         System.out.println( "DEBUG: Add Customer.class" );
354         config.addClass(Customer.class);
355         System.out.println( "DEBUG: Create Session" );
356         sessionFactory = config.buildSessionFactory();
```

Figure 63. UpdateCustomer Method in Hibernate Mapping Component

7.4.3 Using Model Simulating to Test the Refactoring

Using JBoss AOP interceptors, extension code was written to call the Java Hibernate Component “Customer” to create and update customer data. This code is shown below in Figure 64. Note that the code targets two different classes, one used in the simulator only (*app.java.DatabaseAccessor*), and the other in the existing system (*cmpe221.DatabaseAccessor*). This technique is equivalent to rewiring software components at the message level (i.e. the AOP joinpoint).

```
public Object invoke(Invocation invocation) throws Throwable {
    Object[] args = null;
    Object result = null;

    if (invocation instanceof MethodInvocation) {
        MethodInvocation m = (MethodInvocation) invocation;
        args = m.getArguments();
        String targetClassName = m.getTargetObject().getClass().getName();
        System.out.println("TARGET OBJECT CLASS: " + targetClassName);

        if ("app.java.DatabaseAccessor".equals(targetClassName)) {
            String userid = (String) args[0];
            String fname = (String) args[1];
            String lname = (String) args[2];
            String addr = (String) args[3];
            String city = (String) args[4];
            String st = (String) args[5];
            String zip = (String) args[6];
            String email = (String) args[7];
            String bday = (String) args[8];
            String creditcard = (String) args[9];
            Customer.updateCustomer(userid, fname, lname, addr, city, st, zip, email, bday,
            creditcard) ;
            return "true" ;
        }

        if ("cmpe221.DatabaseAccessor".equals(targetClassName)) {
            result = invocation.invokeNext() ; // invoke the base behavior first
            try {
                CustomerProfile p = (CustomerProfile) args[0] ;
                String userid = (String) p.getUserId() ;
                String fname = (String) p.getFirstName() ;
                String lname = (String) p.getLastName() ;
                String addr = (String) p.getStreetAddress() ;
                String city = (String) p.getCity() ;
                String st = (String) p.getState() ;
                String zip = (String) p.getZipCode() ;
```

```

String email = (String) p.getEmail() ;
String bday = (String) p.getBirthday() ;
String creditcard = (String) p.getCreditCardNo() ;
String login = (String) p.getLogin() ;
String password = (String) p.getPassword() ;
userid = Customer.createCustomer( login, password ) ;
Customer.updateCustomer( userid, fname, lname, addr, city, st,
zip, email, bday, creditcard ) ;
}
catch ( Exception e ) {}
return result ; // return result from base behavior
}
}
return "false" ;
}

```

Figure 64. AOP Interceptor Code Calling Hibernate Component

During a simulation session, the invocation of the Hibernate component can be enabled and or disabled via the *execute use case* command or directly against the extension object using convenience methods. Figure 65, below demonstrates the enabling of hibernate features using a convenience method and the invocation of the *updateProfile* method which calls Hibernate.

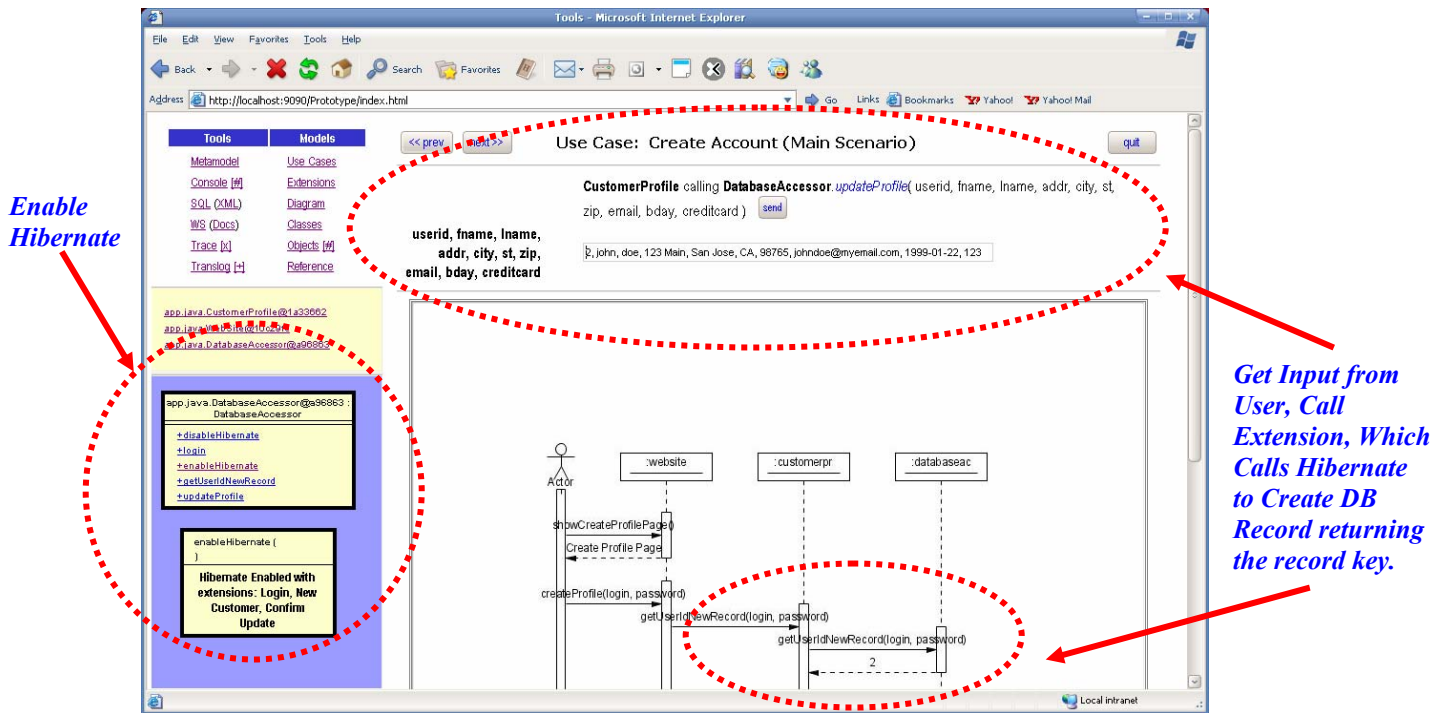
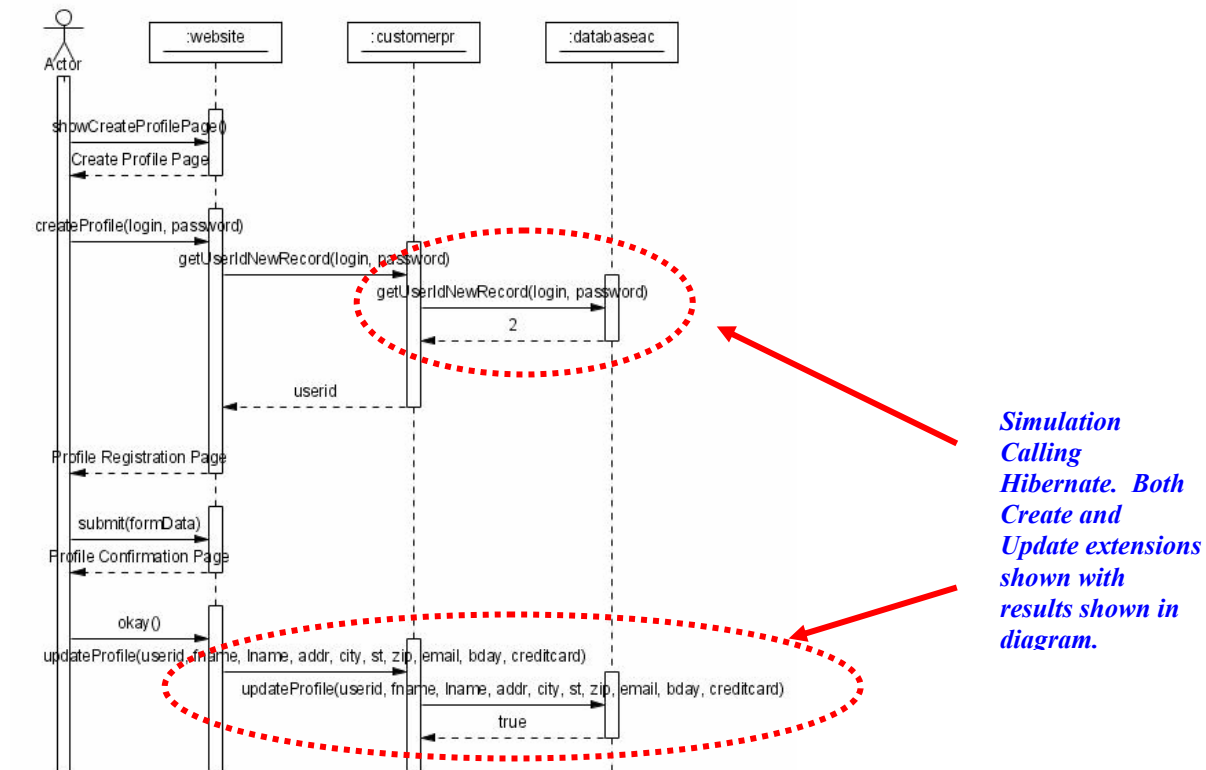
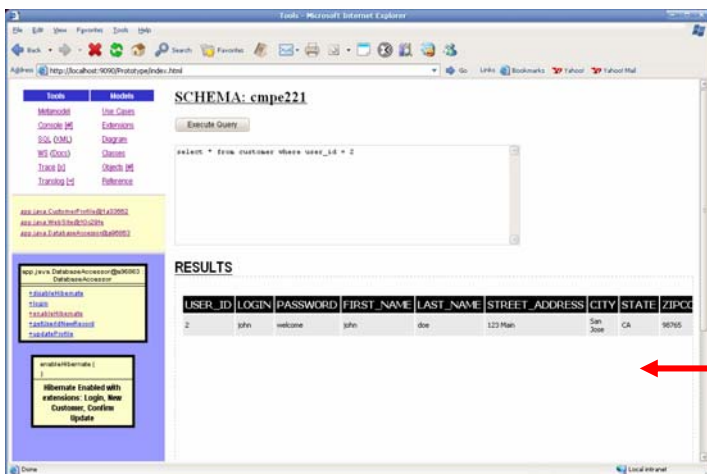


Figure 65. Enabling Hibernate in a Simulation Session

The final results from the simulation session are shown below in Figure 66. The two methods: *getUseridNewRecord* and *updateProfile* in the *DatabaseAccessor* class where invoked with results from Hibernate shown in the sequence diagram. To validate the test, a query was used to find the record in the Oracle database to confirm the creation and update of the record by Hibernate.



Simulation Calling Hibernate. Both Create and Update extensions shown with results shown in diagram.



Successful validation of results via query to database.

Figure 66. Simulation Results From Hibernate Case Study

7.4.4 Hot Deployment of the New Hibernate Component

In the simulation session above, the Hibernate extension was enabled programmatically via dynamic JBoss AOP features. The code below shows how this is done.

```
public static void enableExtension() {
    try {
        AdviceBinding binding = null;
        binding = new AdviceBinding("HibernateConfirmProfileExtension",
            " execution(* app.java.DatabaseAccessor->updateProfile(..)) "
            + " ", null);
        binding.addInterceptor(app.java.HibernateConfirmProfileExtension.class);
        AspectManager.instance().addBinding(binding);
        System.out.println("DEBUG: HibernateConfirmProfileExtension Enabled");
    } catch (Exception ex) {
        System.out.println(ex.getMessage());
    }
}
```

Figure 67. Code that enables Aspects in JBoss AOP

To hot deploy the Hibernate Extension into the existing application, there are no facilities to invoke the method above. As such, JBoss AOP has a hot deploy configuration file. Within this configuration file, the aspect can be enabled. When JBoss AOP reads the update, it will enable the extension.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<aop>
<prepare expr="all(app.java.*)"/>
<bind pointcut="execution(* cmpe221.DatabaseAccessor-&gt;login(..)">
<interceptor class="app.java.HibernateLoginExtension"/>
</bind>
<bind pointcut="execution(* cmpe221.DatabaseAccessor-&gt;updateProfile(..)">
<interceptor class="app.java.HibernateConfirmProfileExtension"/>
</bind>
</aop>
```

Figure 68. JBoss AOP Hot Deployment Configuration File

Note, in Figure 68, above, the binding pointcut extends CMPE221 classes – the existing code base. This was all done without modifying a single line in the current code. As a result, the new Hibernate component can be easily removed, simply with a configuration change in the hot deployment file.

This case study only shows the migration to Hibernate using one database table and Java class (i.e. Customer). As such, this is not a comprehensive database persistence solution – additional tables and classes need to be created and tested. However, to verify whether or not this process actually works and gathers results for the case study, the existing application must be used. To enable this, the new Hibernate components have to co-exist with the existing code base. Aspects make this seamless by adding behavior to existing systems without changing existing functionality – much in the same manner as a *Logging Aspect*. The deployment of the Hibernate components were configured to work in conjunction with existing code by piggy backing on the successful return from a normal create user to MySQL. The results of a test is shown below, which confirms records created in MySQL and Oracle.

Create Profile in MySQL and Oracle

Transaction Successful!

USER_ID	LOGIN	PASSWORD	FIRST_NAME	LAST_NAME	STREET_ADDRESS	CITY
1	jane	welcome	Jane	Doe	123 First Street	Sunnyvale

Figure 69. Test Results of Case Study: App Page, Hibernate Logs, & Db Tables with new Records in MySQL and Oracle

7.4.5 Case Study Summary

This case study confirms the main thesis of this research on active models: that working both from top down (with models and simulation) and bottom up (from the code or via reverse engineering) better supports systems evolution. The case study's work started out with the existing code and used reverse engineering to capture a use case scenario. From there, the existing classes and code was studied for refactoring (or evolutionary) design options. In the case study, Hibernate was chosen as the target database persistent management component for the code base to evolve towards. Using model simulation and the power of aspect-oriented programming (with JBoss AOP) the ease of evolving the new user registration process was demonstrated.

Some limitations with the current research prototype were also noted. In reviewing the CMPE221 student code base, various areas of the application and use cases were studied before one was chosen. The new user registration use case was chosen because evolving that part of the system matches well with the current capabilities of the prototype and its emphasis on method execution and sequence diagrams. Other use cases, the *Game Login* use case (for example), did not match well because it was designed using the Struts framework based on a state based event driven model. Also, to augment or change the behaviors of "struts actions" in this framework requires numerous struts context objects, which is currently not possible to pass to the simulation environment because the prototype only works currently with simple *String* types. However, future work could easily address this.

8. Analysis

The results from the case studies and experience from developing and using the research prototype will be discussed in this section with respect to the challenge problems identified in *Section 3.2 Key Problems*. To summarize, these problems were:

- **Evolution.** *Enterprise Systems are hard to maintain and evolve and are often replaced.*
- **Incompatibility.** *Domain models amongst multi-vendor applications are often incompatible.*
- **Requirements Mismatch.** *The problem of the evolving “dynamic domain model”*
- **Documentation Centric.** *The out-of-date documentation problem.*
- **Lack of logical/physical isolation.** *Configuration management of deployments is a challenge with Java Application Servers.*
- **Distributed Teams.** *Teams are distributed making collaborative work a challenge.*

In addition, it is noted that this work currently addresses only a subset of the vision for a fully integrated Active System. Putting this into context, the work and case studies presented here addresses the following areas:

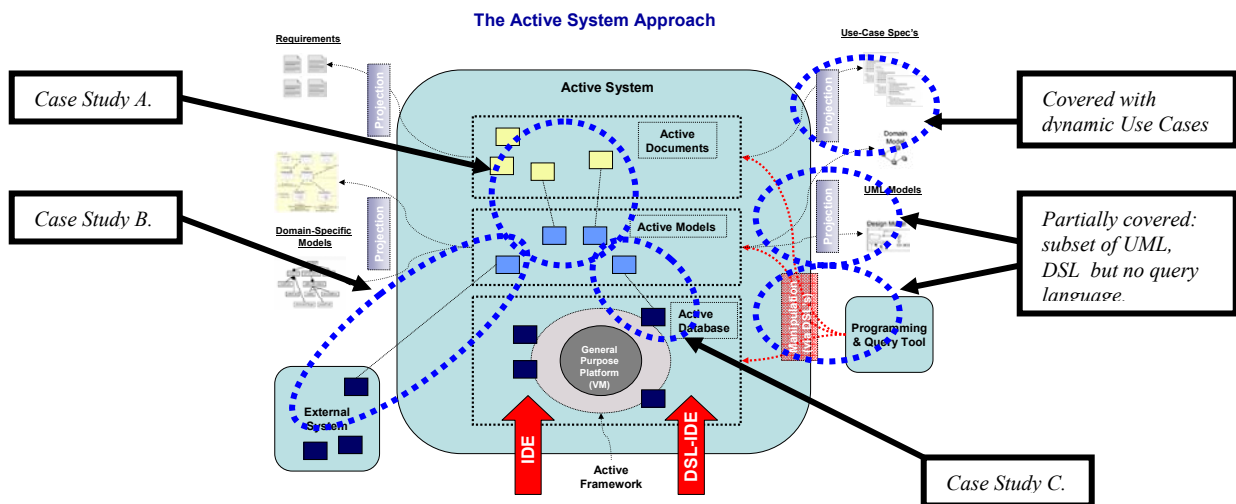


Figure 70. Areas of Active Systems Covered or Demonstrated in This Work

Evolution

Well modularized solutions are easier to maintain and evolve. Conversely, a system that is not well modularized can not be maintained and often ends up in an evolutionary “dead end”. At the code level, agile approaches address this issue with test-driven techniques and refactoring. That is, by architecturally reorganizing the internals of the software while still maintaining external interfaces, and validating the refactoring using automated tests. As shown in Case Study C, refactoring can also be applied at a higher level of abstraction – at the model level -- with the assistance of Active UML Models. In addition, married with Aspect Oriented Middleware, the refactoring can be incrementally deployed to coexist with the current functionality. As a result, validation can occur both at the model level and in the actual system to provide a gradual migration path towards the new architecture.

Incompatibility

Incompatible domain models from multiple vendors were not addressed directly in this research. However, having a textual programming language to describe UML models and a readily available simulator for the language will help a great deal in solving this problem. In addition to SDK’s, API’s, and technical documentation, Vendors of packaged solutions could make available the source code to the underlying domain models (both static and dynamic) so a customer or value added reseller could better understand the “semantics” of the model using a simulator. Furthermore, one vendor’s model can be easily adapted to models of another vendor using the simulator, which could be used to integrate various “multi-vendor” domain objects prior to actual implementation.

Requirements Mismatch

The problem of the “dynamic domain model” is addressed with Active Documents in this research. The solution, however, is an aged old programming language approach --basically, focusing on capturing domain concepts into a declarative “domain specific” language (DSL). The DSL approach in this research differs, however, from other DSL approaches, since the language itself focuses on the semantics of UML (which is a general purpose language). Describing requirements (or documents) using a language could be considered “code”, which is equivalent to the “code first” philosophy in the Agile Community and the thinking behind “Code as Design” [54].

Documentation Centric

Active Documents addresses the problem of “static documents” in current practice. But a deeper problem related to static documents is the “static models”. It is my belief, from experience, that the main reason behind a large amount of technical documentation currently in practice is due primarily to the fear of complexity, or more precisely, the fear of one’s ability to evolve a complex system. I believe that less documentation will be needed if we have interactive modeling environments that assist us in maintaining software systems and automatically generate the latest documents whenever we need them.

Lack of logical and Physical isolation

This problem was described earlier as a key problem to managing the configuration of Java Application Servers and the deployments of application components. This research did not address this directly, but can offer a direction towards solving this using the concept of an “Active System”. That is, the Active System could manage all the physical deployment aspects of the environment, freeing the development work to only the logical layers. As such, techniques in code generation, configuration generation, automated build systems, and version control can all be integrated into the physical

infrastructure behind Active Systems. For example, when I create a “Class” in the modeling environment, and proceed to produce “code” to support the class, I should not have to worry about compiling this into a Java Class file, packaging it into some JAR file, and deploying this file (along with a number of configuration files) into a Java Application Container.

Distributed Teams

Collaboration is “key” to the future of software development. And, as the nature of building complex distributed system force us to work with colleagues from various corners of the world, we must address this issue! Current software development methodologies have not addressed this issue directly. The more distributed the team, the less we can feasibly have face-to-face time, and therefore, the more we tend to documentation centric and waterfall approaches. Although, in the literature, it has been well argued that waterfall approaches do not work, the solutions to these problems as offered in Agile approaches do not scale well to large distributed teams who often don’t even speak to one another. The success of open source software development teams offer some clues to how best to address this issue. Successful open source project have one thing in common. They all have an automated configuration management, code management, and regular build system. This not only provides a means to control the evolution of the software, but also provides a means of communication amongst teams. Every developer should be able to easily build a “sandbox” of the software and to contribute to the source code repository without the fear of his/her contribution causing havoc to the developer community at large. The infrastructure for an Active System could provide all of these benefits, but also provide a modeling and simulation environment to enable creation of high-levels of abstractions and exploration – i.e. a sandbox for models!

9. Conclusion

This research investigated interactive modeling with UML diagrams and Aspect-Oriented middleware. The research prototype explored the feasibility of implementing an integrated “Active System” using existing open source AOP technology, the Java JVM, and a commercial relational database system. In three case studies, the prototype was tested against three common problems in software development. In the first case study, it was shown that dynamic documentation can be easily generated and maintained when the semantics of the specification is expressed without ambiguity using a declarative programming language. In the second case study, it was demonstrated that modeling doesn’t have to just be visual diagrams. When models are dynamic and interactive, the modeling elements themselves can be executing code, or better yet, live systems. Using the Cybersource simulation infrastructure, the second case study demonstrated how calling a web service could be integrated into an interactive sequence diagram. Effectively, the case study shows that when working with interactive UML models, the level of abstraction can be mixed; that is, the modeling elements can be pure models, real code, or real systems. In the third case study, it was demonstrated that systems evolution can be supported with aspect-oriented technology, effectively, rewiring software components. In the case study, the power of AOP was demonstrated from different aspects. First, AOP can be leveraged to dynamically reverse engineer a live system. The research prototype implemented this tracing mechanism using JBoss AOP code and the model diagram generation framework develop as part of the active documentation facility. Second, AOP can be used in the simulator to explore the system behavior when aspects are applied. The case study attempted to migrate an existing code base towards a entirely different database architecture. Such an attempt using traditional software development with static designs and documentation would be very difficult as the implications of such a change can not be well understood until code is written. And lastly, the case study shows that system evolution can be

done in small steps using AOP. That is, parts of the new code can co-exist non-intrusively with existing code and gradually replace existing code over time. Effectively, with AOP, refactoring can be done at a larger scale.

In implementing the research prototype, it was found that using the Java JVM's reflective capabilities requires explicit identification of classes and methods. That is, there were no query facilities to search for classes. Unlike a database, where the metadata can be queried, the metadata about objects in the Java JVM can not. This severely limits the power of AOP and forces each AOP framework to implement their own language for pointcuts specification. In the implementation of the declarative use case specification language, use case extensions were implemented as aspects. The metadata for the use case extensions were stored in a relational database and as a result SQL was used to implement pointcuts for use case extensions. This greatly simplifies working with early aspects in interactive models. Further evolution of Java's Reflective capabilities and perhaps integration with a metadata repository would provide a better platform for implementing an interactive modeling environment for exploring early aspects.

To sum up, this research work has demonstrated that the combination of interactive modeling and Aspect-Oriented Middleware provides a powerful combination to tackle the current problems in software development. Static documents can be active and always up-to-date, UML designs can be interactive and more faithfully representative of the software, and systems evolution can be better explored and implemented with confidence.

10. Related and Future Work

This work is based primarily on the work of Ivar Jacobson and Pan-Wei Ng [3] and Diomidis Spinellis's Declarative UML rendering library, UMLGraph [71]. As discussed in the introduction, most model driven development efforts to date (MDA [26] and Software Factories [16]) focus on automated code generation and very little on simulation. However, a case study on executable use cases in [59] and research work on testing UML designs [60] uses simulation as an important tool for validation. More common, are approaches to executing use cases or UML using a virtual machine. For example, the work on UML virtual machines from Trygve Reenskaug [17], Dirk Riehle, et al. [53], and Executable Z [57]. The simulator in this research work is tightly integrated with the Java JVM and can be considered a virtual machine for UML sequence diagrams and Use Cases. However, the main focus of this research is not to implement a virtual machine that runs all of the UML models, but rather to bring the dynamic models closer to the code. In essence, the simulator makes very little distinction between a simulated object, vs. a scripted object in Groovy, vs. an implemented object written in Java. As such, this research leverages the Java JVM itself as the execution environment.

This work also shares a common goal with the Model-Centric Software Development (MCSD) work at Lockheed Martin [61], which aims at integrating the models into every aspect of software development, but, differs in the way it deals with legacy code. In MCSD, reverse engineering is used to create models from code. In our approach, legacy code is either incorporated directly into the simulation models or wrapped by a simulated component. In addition, since we work at the JVM bytecode level, the simulation models can also incorporate third-party components where source code is not available.

The body of work in generative software development [1] provides many of the techniques used in this research. Currently, feature models from product line engineering and domain analysis methods are being considered at the metamodel layer for integration with the use cases and UML models. In addition, there seems there is considerable synergy and promise in integrating with the Naked Objects Framework [28] and the further exploration of additional language constructs for expressing design concepts, such as the UML state diagram, design patterns, or other AOP design techniques [69].

On the user interface front for modeling, additional rendering tools will help provide different perspectives against the same metamodel. For example, integration with TouchGraph [68] to render different aspects of the model and link them with one another to provide for a dynamic view of traceability.

11. Appendices

11.1 Sample UI Screens From the Research Prototype

11.1.1 Dynamic Use Case Document

The screenshot shows a web browser displaying a 'USE CASE SPECIFICATION' document. The document is structured as follows:

USE CASE SPECIFICATION		
UC-01		
Name	Reserve Room	
Description	The use case begins when a customer wants to reserve a room. (or multiple rooms)	
Actor	Customer	
Actions	Reserve a room	
Flow Step	Description	
Step 1	The customer selects to reserve a room.	
Step 2	The system displays the type of rooms the hotel has and their respective rates.	
Step 3	The customer checks room cost for a desired room type and period of stay.	
Step 4	The system computes the cost and displays the information to the customer.	
Step 5	The customer makes the reservation for the chosen room(s).	
Step 6	The system reserves the rooms for the customer.	
Step 7	The reservation is displayed to the customer with a confirmation number and check-in instructions.	
Step 8	The use case terminates.	
Extensions		
Extension Name	Description	
Extension	Description	
Participants		
Object Name	Class Name	Action

Callouts in the image provide additional context:

- Views into Meta Model... Use Cases Features Components Configuration Etc...**: Points to the 'Meta Models' menu in the browser's left sidebar.
- Console for Declarative Programming with A DSL**: Points to the 'Console' area in the browser's left sidebar.
- Active Documents "Use Cases" Derived directly from Meta Model**: Points to the 'USE CASE SPECIFICATION' document content.

11.1.2 Interactive UML Model

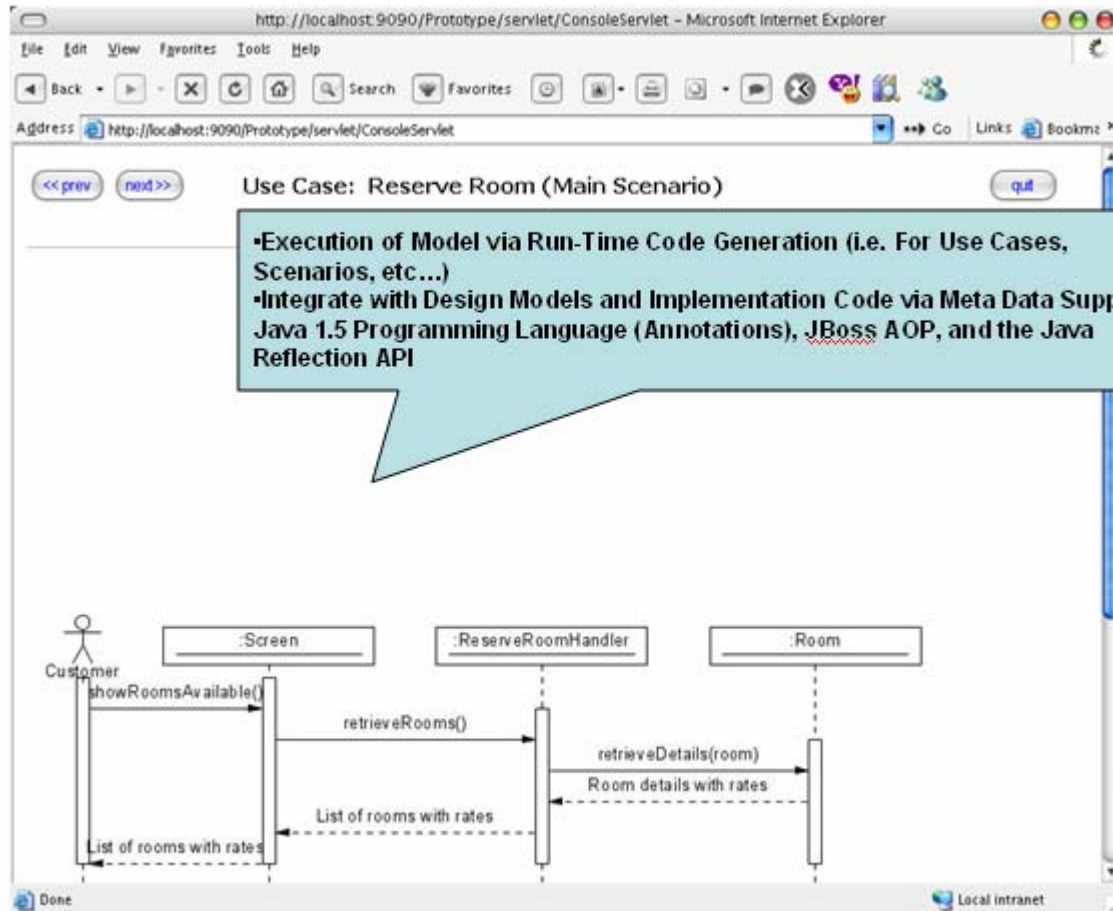
The screenshot displays a web browser window titled "Tools - Microsoft Internet Explorer" showing a web application interface. On the left, there are sections for "Tools" (Database Browser, Command Console, SQL (HTML), SQL (XML)) and "Meta Models" (Use Cases, Extensions, Features, Analysis Classes). Below these are "execute" and "clear" buttons and a "RESULT:" label. The main content area shows a sequence diagram titled "Reserve Room" with ID "reserve_room" and description "The use case begins when a customer wants to reserve a room".

The sequence diagram involves four lifelines: Customer, System, Screen, and Reservation. The process is divided into five steps:

- Step 1:** Customer sends "1.1: showRoomsAvailable()" to System. System sends "1.1.1: retrieveRooms()" to Screen. Screen returns "List of rooms with rates" to System. System returns "List of rooms with rates" to Customer.
- Step 2:** Customer sends "3.1: computeCost(room, period)" to System.
- Step 3:** System sends "3.1.1: addRoomToReservation(room)" to Screen. Screen sends "3.1.1.1: addRoom(room)" to Reservation. Reservation returns "void" to Screen. Screen returns "void" to System. System sends "3.1.2: computeReservationRates()" to Screen. Screen sends "3.1.2.1: retrieveRates()" to Reservation. Reservation returns "Rates" to Screen. Screen returns "List of rooms with rates" to System. System returns "Final cost for request" to Customer.
- Step 4:** Customer sends "5.1: makeReservation(room, period)" to System.
- Step 5:** System sends "5.1.1: makeReservation(room, period)" to Screen.

Annotations include a callout box pointing to the Reservation lifeline: "Generated UML view from Meta Model" and another callout box pointing to the interface: "Direct Manipulation of Meta Model via DSL Programming".

11.1.3 Simulation Environment



11.1.4 Dynamic Use Case Diagram

The screenshot displays a web browser window titled "Tools - Microsoft Internet Explorer" with the address bar showing "http://localhost:9090/Prototype/index.html". The browser interface includes a menu bar (File, Edit, View, Favorites, Tools, Help), a toolbar with navigation buttons, and a status bar at the bottom indicating "Done" and "Local Intranet".

On the left side of the browser, there is a "Tools" and "Models" panel. The "Tools" section includes links for Browser, Console, SQL (XML), W3 (Docs), Java Query, and Demo App. The "Models" section includes links for Use Cases, Extensions, Diagram, Classes, Objects, and Roles. Below these panels are "execute" and "clear" buttons.

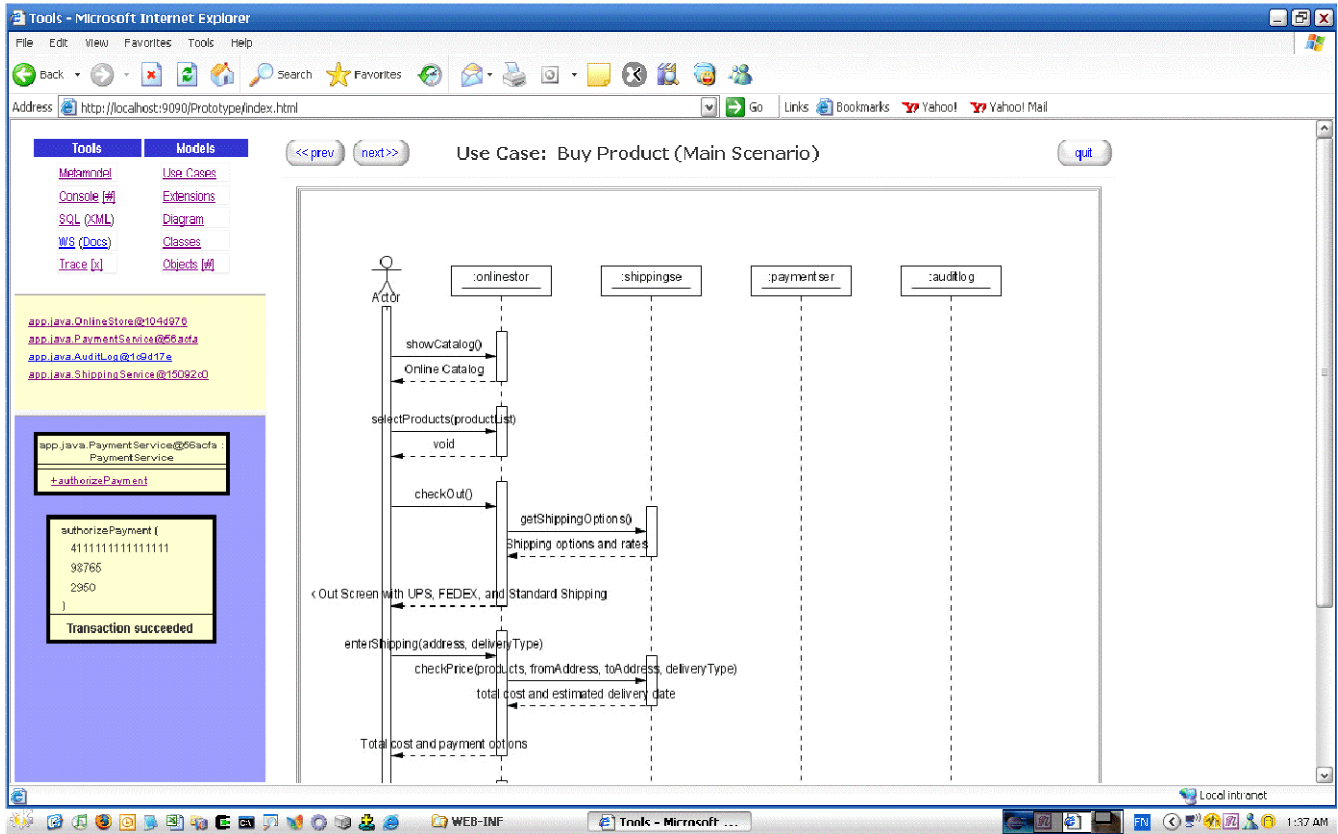
The main area of the browser displays a dynamic use case diagram. A stick figure actor labeled "Customer" is connected to two use cases: "Check Order" and "Buy Product". Both use cases are contained within a blue rectangular boundary labeled "System". "Check Order" is extended by "Audit Authorize Purchase Extension" (indicated by a dashed arrow labeled <<extends>>). "Buy Product" is extended by "Check Out Extension" (indicated by a dashed arrow labeled <<extends>>).

Below the diagram, there is a code editor with the following text:

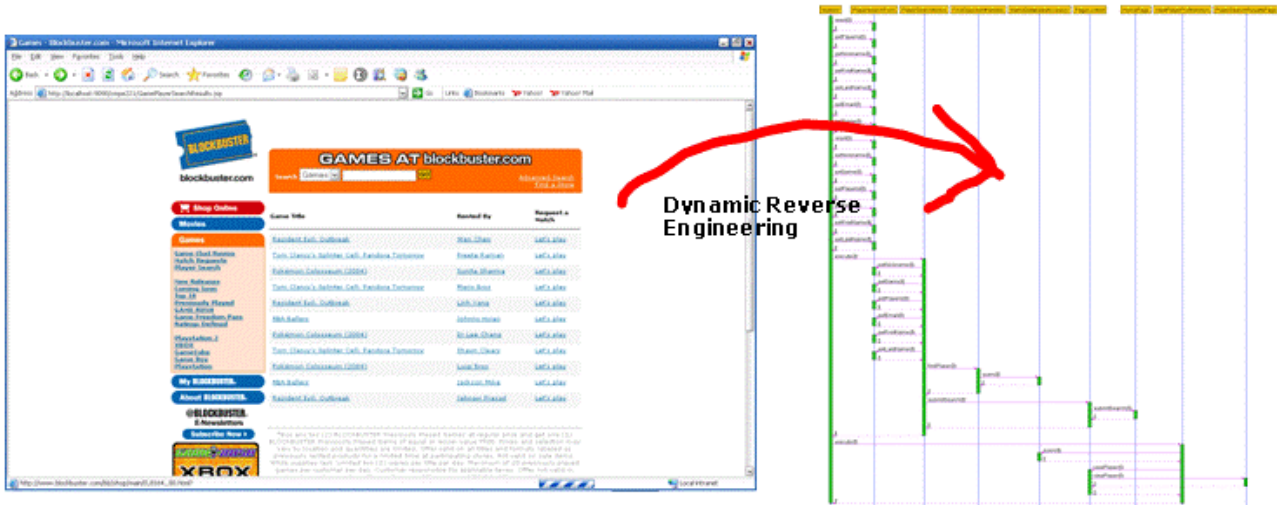
```
create use case
check order described
as "Customer wants to
check order status."
with
actor "Customer"
goal "Check Order
Status"
```

Below the code editor, the text "RESULT:" is followed by "Success!".

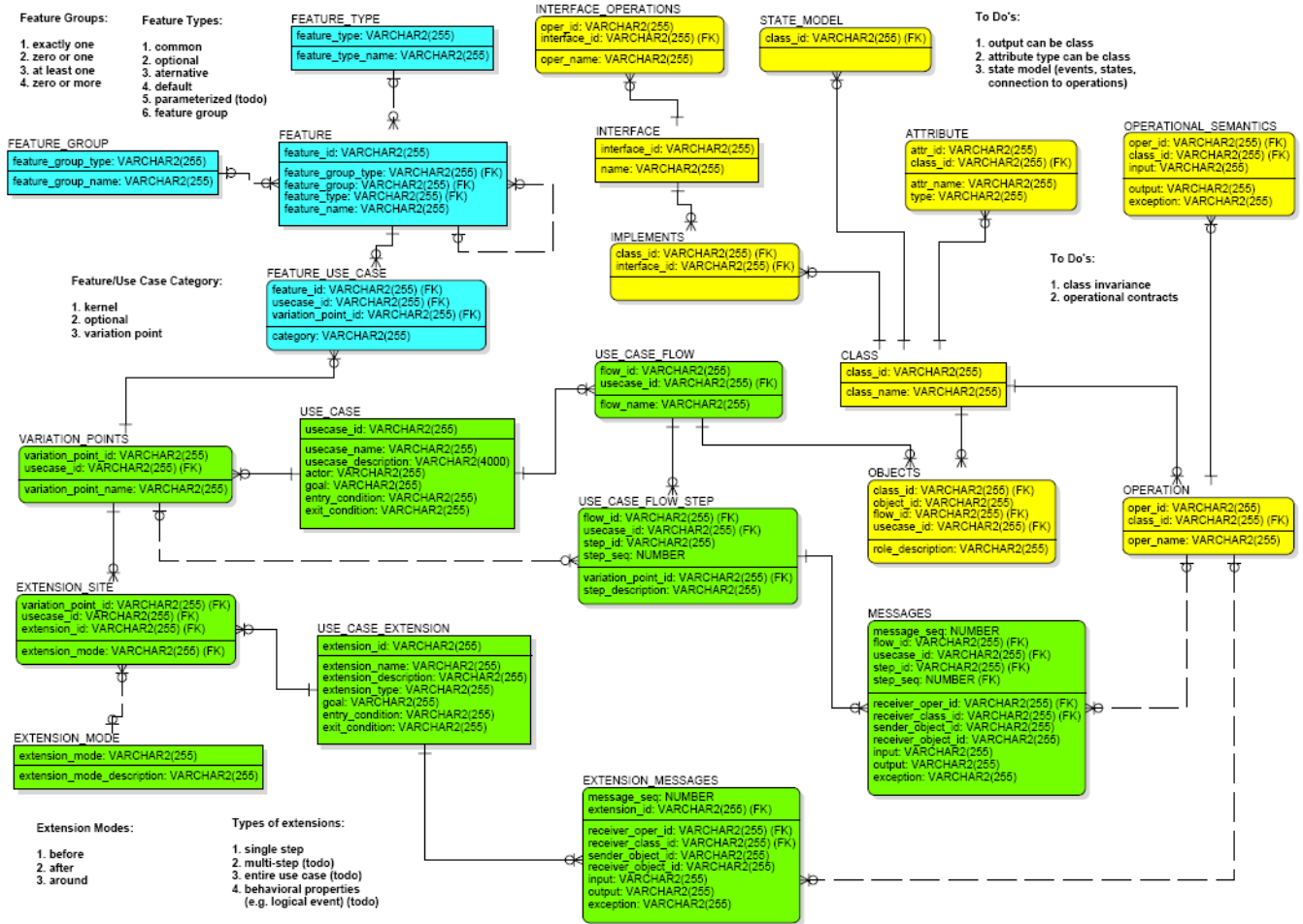
11.1.5 Direct Interaction with Java Objects during Modeling



11.1.6 Live System Reverse Engineering to Sequence Diagram



11.2 Metamodel

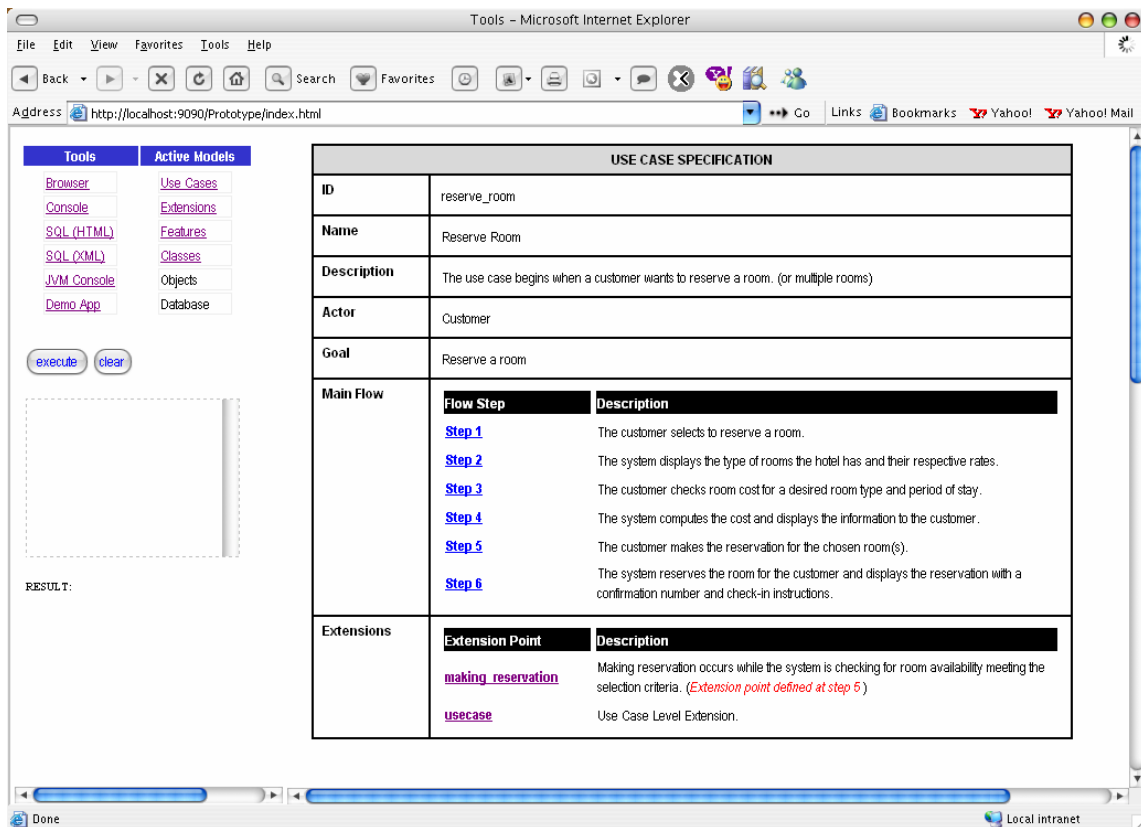


11.3 DSL Syntax and Examples

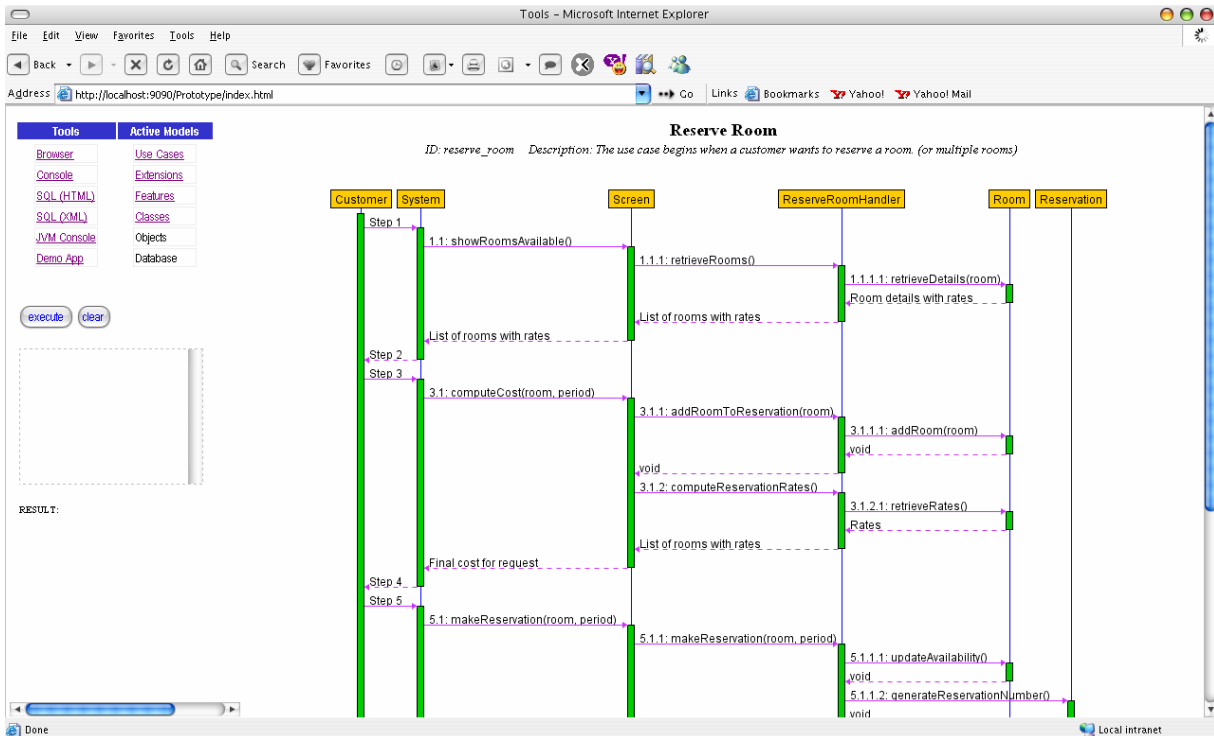
```

create use case reserve_room described as "The use case begins when a customer wants to reserve a room.
(or multiple rooms)"
steps
    step 1 "The customer selects to reserve a room."
    step 2 "The system displays the type of rooms the hotel has and their respective rates."
    step 3 "The customer checks room cost for a desired room type and period of stay."
    step 4 "The system computes the cost and displays the information to the customer."
    step 5 "The customer makes the reservation for the chosen room(s)."
    step 6 "The system reserves the room for the customer and displays the reservation with a
confirmation number and check-in instructions."
with
    actor          "Customer"
    goal           "Reserve a room"
    extension
    making_reservation at step 5
    described as "Making reservation occurs while the system is checking for room availability
meeting the selection criteria."

```



```
update use case reserve_room
add main scenario messages
  1.1    from System to Screen
        requesting showRoomsAvailable
        returning "List of rooms with rates"
  1.1.1  from Screen to ReserveRoomHandler
        requesting retrieveRooms
        returning "List of rooms with rates"
  1.1.1.1 from ReserveRoomHandler to Room
        requesting retrieveDetails with "room"
        returning "Room details with rates"
  3.1    from System to Screen
        requesting computeCost with "room", "period"
        returning "Final cost for request"
  3.1.1  from Screen to ReserveRoomHandler
        requesting addRoomToReservation with "room"
        returning "void"
  3.1.1.1 from ReserveRoomHandler to Room
        requesting addRoom with "room"
        returning "void"
  3.1.2  from Screen to ReserveRoomHandler
        requesting computeReservationRates
        returning "List of rooms with rates"
  3.1.2.1 from ReserveRoomHandler to Room
        requesting retrieveRates
        returning "Rates"
  5.1    from System to Screen
        requesting makeReservation with "room", "period"
        returning "Confirmation number and instructions"
  5.1.1  from Screen to ReserveRoomHandler
        requesting makeReservation with "room", "period"
        returning "List of rooms with rates"
  5.1.1.1 from ReserveRoomHandler to Room
        requesting updateAvailability
        returning "void"
  5.1.1.2 from ReserveRoomHandler to Reservation
        requesting generateReservationNumber
        returning "void"
  5.1.1.3 from ReserveRoomHandler to Reservation
        requesting createReservation
        returning "void"
```



```

create use case extension waitlist_extension
described as "put the customer on a waiting list if no rooms are available"
with
    goal = "a waiting list reserved for the customer"
    entry = "no rooms available"
returning error
advice steps
    step 1 = "A make reservation request failed due to room unavailability."
    step 2 = "The System creates a pending reservation and returns the details"
advice scenario messages
    1.1 from Extension to WaitingListHandler
        requesting putCustomerOnWaitList with "customer"
        returning "Waiting list reservation."
    1.1.1 from WaitingListHandler to Reservation
        requesting generateReservationNumber
        returning "reservation number"
    1.1.2 from WaitingListHandler to Reservation
        requesting createPendingReservation with "reservation number"
        returning "Success"
    1.1.3 from WaitingListHandler to WaitingList
        requesting addPendingReservation with "reservation number"
        returning "Success"
extending
    reserve_room at making_reservation returning error

```

Tools - Microsoft Internet Explorer

Address: http://localhost:9090/Prototype/index.html

USE CASE SPECIFICATION							
ID	wallist_extension						
Name	Wallist Extension						
Description	put the customer on a waiting list if no rooms are available						
Actor	ExtensionPoint						
Goal	a waiting list reserved for the customer						
Returning Error	<table border="1"> <thead> <tr> <th>Flow Step</th> <th>Description</th> </tr> </thead> <tbody> <tr> <td>Step 1</td> <td>A make reservation request failed due to room unavailability.</td> </tr> <tr> <td>Step 2</td> <td>The System creates a pending reservation and returns the details</td> </tr> </tbody> </table>	Flow Step	Description	Step 1	A make reservation request failed due to room unavailability.	Step 2	The System creates a pending reservation and returns the details
Flow Step	Description						
Step 1	A make reservation request failed due to room unavailability.						
Step 2	The System creates a pending reservation and returns the details						
Extends	<table border="1"> <thead> <tr> <th>Use Case</th> <th>Pointcut Description</th> </tr> </thead> <tbody> <tr> <td>reserve room</td> <td>Making reservation occurs while the system is checking for room availability meeting the selection criteria.. This Extension occurs at extension point: <i>making_reservation</i>.</td> </tr> </tbody> </table>	Use Case	Pointcut Description	reserve room	Making reservation occurs while the system is checking for room availability meeting the selection criteria.. This Extension occurs at extension point: <i>making_reservation</i> .		
Use Case	Pointcut Description						
reserve room	Making reservation occurs while the system is checking for room availability meeting the selection criteria.. This Extension occurs at extension point: <i>making_reservation</i> .						

RESULT:

returning error flow

Microsoft Office OneNote 2003
The screen clipping was created successfully.
You can view the screen clipping in a side note window, or paste it into your notes or into other programs.

11.4 Software Tools and Development Frameworks Used

- Oracle Database 10g XDK for SQL to XML to HTML document generation
- JBoss AOP Framework (during the course of the research initially worked with 1.0, later updated to 1.1 and finally at version 1.3)
- Java 1.5 (Tiger) SDK
- Groovy Scripting Language (version 1.0 – JSR05)
- Eclipse (various versions: SDK 2.0, 3.0, 3.1, and Jboss IDE 1.5)
- Tomcat 4.1.30
- Hibernate Framework 2.1.8
- JavaCC Parser Generator Toolkit
- UML and Graph Rendering Frameworks: UMLGraph, Sequence, and GraphViz

12. Tables and Figures

List of Tables

Table 1. Summary of Contributions from Machine-Independent Programming	10
Table 2. Summary of Contributions of Virtual Machines	11
Table 3. Summary of Contributions from Programming Language Interoperability & Domain-Specific Languages.....	12
Table 4. Summary of Contributions from Increasing Modularity	13
Table 5. Innovations and Problems that Motivate Active Models.....	32

List of Figures

Figure 1. Influences on MDA and Software Factories	18
Figure 2. Influences on BabyUML (reproduced from [30]).....	20
Figure 3. Logging not modularized in Tomcat (reproduced from [34])	21
Figure 4. Extracting Operational Contracts (diagram adapted from [49]).....	26
Figure 5. Oracle Dictionary Tables For user “tables” and “columns”	27
Figure 6. Sample Database Trigger Template	27
Figure 7. An Example of a committed transaction captured by database triggers.....	28
Figure 8. Sequence Diagram Showing Operational Contracts Extracted Using Database Triggers.	29
Figure 9. Tracing Aspect in AspectJ with before and after advice on constructors and method calls	30
Figure 10. Login Scenario Demonstrating Tracing Aspect	31
Figure 11. Traditional Software Development & MDA approach	34
Figure 12. Software Product Lines & Generative Software Development.....	35
Figure 13. Software Factories Tools for Product Lines & Generative Software Development	36
Figure 14. Active Systems Integrate Domain-Specific and General-Purpose Development Methods.....	37
Figure 15. UML Models and Their Relationships.....	38
Figure 16. Metamodel for Use Cases.....	39
Figure 17. Architecture of Experimental Platform	42
Figure 18. DSL Non-Terminal For Create Use Case Command.....	43
Figure 19. Example of JavaCC Integration with Java DSL Interpreter Objects.....	43
Figure 20. DSL Interpreter Command Class Diagram	44
Figure 21. UML Sequence Diagram for DSL Interpretation.....	45
Figure 22. Object Model For Create Use Case Command (Class Diagram)	46
Figure 23. Debug <i>dump()</i> method in Create Use Case Command.....	46
Figure 24. Example Create Use Case Command & Parser Results	47
Figure 25. Portions of the execute() command for Create Use Case Command	48
Figure 26. Example Update Use Case Command.....	49
Figure 27. Object Model for Update Use Case Command (Class Diagram).....	49
Figure 28. Example Create Use Case Extension Command.....	50
Figure 29. Object Model for Create Use Case Extension Command (Class Diagram)	50
Figure 30. Example Execute Use Case Command	51

Figure 31. Object Model for Execute Use Case Command.....	51
Figure 32. Examples of DSL Code for Diagram Generators.....	53
Figure 33. Example DSL Code Generator for Use Case Diagrams (partial view).....	54
Figure 34. View Diagram Collaboration	55
Figure 35. View Document Collaboration.....	56
Figure 36. View Use Case XDK Page and XSLT Template.....	56
Figure 37. Simulation Collaboration (Sequence Diagram).....	57
Figure 38. Data Model for Execution State Tables	58
Figure 39. Sample Code Generated by Research Prototype.....	60
Figure 40. Basic Support for Documenting Use Cases	61
Figure 41. WYSIWYG HTML Editor Support For Use Cases	62
Figure 42. Buy Product Use Case Specification Document (from [67]).....	63
Figure 43. Buy Product Use Case Specification using a Declarative Language	63
Figure 44. Generated HTML Use Case Document.....	64
Figure 45. Example of a Use Case Extension Declaration	65
Figure 46. Example of Use Case Extension Sequence Diagram Interaction Frame.....	65
Figure 47. A Jointpoint Model for Use Cases	66
Figure 48. Adding Sequence Diagrams to Use Cases	66
Figure 49. Generated Class Diagram From Use Case Metadata.....	67
Figure 50. Example Use Case Execution Command with an Extension	67
Figure 51. A Simulation Session with a Returning Error Advise Enabled.....	68
Figure 52. Groovy Code Editor.....	69
Figure 53. Simple Input Dialog for a Java Method	70
Figure 54. Options and API for Integrating with Cybersource [74].....	72
Figure 55. Payment Service Java Code for Calling Cybersource.....	72
Figure 56. Test Results From Calling Cybersource via Web Services.....	73
Figure 57. Database Access Classes in CMPE 221 Student Projects.....	76
Figure 58. New User Registration Screen	77
Figure 59. Dynamically Generated Sequence Diagram Trace.....	78
Figure 60. Create Account Use Case for Refactoring Case Study	79
Figure 61. Hibernate Extension Use Cases.....	80
Figure 62. Hibernate Mapping and Generated Java Class.....	81
Figure 63. UpdateCustomer Method in Hibernate Mapping Component.....	82
Figure 64. AOP Interceptor Code Calling Hibernate Component.....	84
Figure 65. Enabling Hibernate in a Simulation Session	84
Figure 66. Simulation Results From Hibernate Case Study	85
Figure 67. Code that enables Aspects in JBoss AOP	86
Figure 68. JBoss AOP Hot Deployment Configuration File	86
Figure 69. Test Results of Case Study: App Page, Hibernate Logs, & Db Tables with new Records in MySQL and Oracle.....	87
Figure 70. Areas of Active Systems Covered or Demonstrated in This Work.....	89

13. References

- [1] Krzysztof Czarnecki and Ulrich W. Eisenecker. **Generative Programming: Methods, Tools, and Applications**. Addison-Wesley, 2000.
- [2] Robert E. Filman, Tzilla Elrad, Siobhan Clarke, and Mehmet Aksit. **Aspect-Oriented Software Development**. Addison-Wesley, 2005.
- [3] Ivar Jacobson and Pan-Wei Ng. **Aspect-Oriented Software Development with Use Cases**. Addison-Wesley, 2005.
- [4] Walter Hirsch and Cristina Lopes. **Separation of Concerns**. College of Computer Science, Northeastern University, 1995.
- [5] Brett McLaughlin and David Flanagan. **Java 1.5 Tiger**. O'Reilly Media, Inc. 2004.
- [6] Ira R. Forman and Nate Forman. **Java Reflection in Action**. Manning Publication. 2005.
- [7] Harold Ossher and Peri Tarr. **Multi-Dimensional Separation of Concerns and the Hyperspace Approach**. Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development (KLUWER, 2000).
- [8] Peri Tarr, Maja D'Hondt, Lodewijk Bergmans, and Cristina Videira Lopes. **Workshop on Aspects and Dimensions of Concern: Requirements on, and Challenge Problems For, Advanced Separation of Concerns**. Springer-Verlag, 2000, ECOOP 2000 Workshop Reader. See Workshop website <http://trese.cs.utwente.nl/Workshops/adc2000>.
- [9] Thomas Patzke and Dirk Muthig. **Product Line Implementation Technologies – Programming Language View**. IESE-Report No. 057.02/E, 2002.
- [10] Stefan Kettemann, Dirk Muthig, and Michalis Anastasopoulos. **Product Line Implementation Technologies – Component Technology**. IESE-Report No. IESE-015_03, 2003.
- [11] Raul Silaghi and Alfred Strohmeier. **Integrating CBSE, SoC, MDA, and AOP in a Software Development Method**. Software Engineering Laboratory, Swiss Federal Institute of Technology, 2003. See <http://icwww.epfl.ch/publications/list.php>.
- [12] Michalis Anastasopoulos and Dirk Muthig. **An Evaluation of Aspect-Oriented Programming as a Product Line Implementation Technology**. Fraunhofer Institute for Experimental Software Engineering (IESE). Springer-Verlag, 2004.
- [13] Manali Bhole and Karl Lieberherr. **Use Case Modularity using Aspect Oriented Programming**. College of Computer and Information Sciences, Northeastern University, 2004.
- [14] Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, Nosa Omorogbe. **The Architecture of a UML Virtual Machine**. OOPSLA 2001.
- [15] R.J.A Buhr, R.S. Casselman, T.W. pearce. **Design Patterns with Use Case Maps: A Case Study in Reengineering an Object-Oriented Framework**. Department of Systmes & Computer Engineering, Carleton University, Ottawa Canada. 1996.
- [16] Jack Greenfield, Keith Short, Steve Cook, Stuart Kent. **Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools**. Wiley, 2004.

- [17] Trygve Reenskaug. **A Rudimentary UML Virtual Machine as a Smalltalk Extension**. Book Draft. 2004.
- [18] Frederick P. Brooks, Jr. **No Silver Bullet - Essence and Accidents of Software Engineering**. IEEE Computer Magazine, April 1987.
- [19] Frederick P. Brooks, Jr and [co-author]. **The Mythical Man-Month**. [Publisher], 1995 Edition.
- [20] Jason Bloomberg. **Software's Dirty Little Secret**. ZapThink Document ID: ZAPFLASH-09012004, 2004.
- [21] Lt. Col. Thomas M. Schorsch, Ph.D. and David A. Cook, Ph.D.. **Evolutionary Trends of Programming Languages**. STSC CrossTalk, Feb 2003.
- [22] E. Gamma, R. Helm, R. Johnson, J. Vlissides. **Design Patterns – Elements of Reusable Object-Oriented Software**. Addison-Wesley, 1994.
- [23] Martin Fowler. **Is Design Dead?** URL: <http://www.martinfowler.com/articles/designDead.html> (May 2004).
- [24] Neal Leavitt . **Whatever Happened to Object-Oriented Databases?** IEEE Computer, August 2000.
- [25] Scott W. Ambler. **The Object-Relational Impedance Mismatch**. URL: <http://www.agiledata.org/essays/impedanceMismatch.html>. (2005).
- [26] OMG. **Model Driven Architecture - A Technical Perspective**. Document number ORMSC 2001-07-01.
- [27] Martin Fowler. **Language Workbenches and Model Driven Architecture**. URL: <http://martinfowler.com/articles/mdaLanguageWorkbench.html>. (June, 2005)
- [28] Richard Pawson (PhD Thesis). **Naked Objects**. Department of Computer Science, Trinity College, Dublin. June, 2004.
- [29] Dan Haywood. **Agile MDA - Naked Objects & Together Control Center. (Presentation)**. URL: <http://blog.haywood-associates.co.uk/page/DanHaywood>.
- [30] Trygve Reenskaug. **Empowering People with BabyUML (ECOOP 2004 Opening Talk)**. 2004.
- [31] Trygve Reenskaug. **The BabyUML Discipline of Programming (DRAFT)**. October, 2005.
- [32] Walter Hürsch and Cristina Lopes. **Separation of Concerns**. Technical report by the College of Computer Science, Northeastern University. 1995.
- [33] Cristina Lopes. **Aspect-Oriented Programming A Historical Perspective**. ISR Technical Report # UCI-ISR-02-5. December, 2002.
- [34] Mik Kersten. **AspectJ - The Language and Development Tools (OOPSLA2002 Demo)**. URL: <http://aspectj.org>.
- [35] Lihua Xu, Hadar Ziv, Debra Richardson, Zhixiong Liu. **Towards Modeling Non-Functional Requirements in Software Architecture**. Early Aspects at AOSD, 2005.
- [36] **Early Aspects: The Current Landscape**. Technical Report, Lancaster University. February, 2005.
- [37] **Survey of Aspect-Oriented Analysis and Design Approaches**. AOSD Europe, 2005.
- [38] Siobhan Clarke and Elisa Banlassad. **Aspect-Oriented Analysis and Design: The Theme Approach**. Addison-Wesley, 2005.

- [39] João Araújo and Ana Moreira. **An Aspectual Use-Case Driven Approach**. Departamento de Informática, Faculdade de Ciências e Tecnologia. 2003.
- [40] Manali Bhole and Karl Lieberherr. **Use Case Modularity using Aspect Oriented Programming**. College of Computer and Information Sciences, Northeastern University, Boston, MA. 2004.
- [41] Ivar Jacobson. **Use Cases and Aspects – Working Seamlessly Together**. IBM, 2003.
- [42] Gary Chastek and John D. McGregor. **Early Aspects in Software Product Line in Product Production**. Aspects & Product Lines Workshop at SPLC, 2005.
- [43] Mik Kersten. **AOP@Work - AOP tools comparison, Part 1**. IBM DevWorks, 2005. URL: <http://www-128.ibm.com/developerworks/java/library/j-aopwork1/>.
- [44] Mik Kersten. **AOP@Work - AOP tools comparison, Part 2**. IBM DevWorks, 2005. URL: <http://www-128.ibm.com/developerworks/library/j-aopwork2/>.
- [45] Devon Simmonds, Sudipto Ghosh, and Robert France. **An Aspect Oriented Model Driven Architecture Framework for Middleware Transparency**. AOSD, 2003.
- [46] Adrian Colyer. **AOP@Work - Dependency injection with AspectJ and Spring**. IBM DevWorks, 2005. URL: <http://www-128.ibm.com/developerworks/java/library/j-aopwork13.html>.
- [47] Mariano Cilia, Michael Haupt, Mira Mezini, Alejandro Buchmann. **The Convergence of AOP and Active Databases Towards Reactive Middleware**. GPCE, 2003.
- [48] Kuldeep Kumar and Jos van Hillegerberg. **ERP Experiences and Evolution**. Communications of the ACM, April 2000/Vol. 43, No. 4.
- [49] Craig Larman. **Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process, Second Edition**. Prentice Hall, 2001.
- [50] Marc J. Balcer. **An Executable UML Virtual Machine (Presentation, 2003)**. URL: <http://www.ModelCompilers.com>.
- [51] J.Bhasker. **A VHDL Primer, 3rd Ed**. Prentice Hall, 1998.
- [52] Diomidis Spinellis. **On the Declarative Specification of Models**. IEEE Software, March/April 2003.
- [53] Dirk Riehle, Steven Fraleigh, Dirk Bucka-Lassen, Nosa Omorogbe. **The Architecture of a UML Virtual Machine**. OOPSLA 2001.
- [54] **Code as Design: Three Essays by Jack W. Reeves**. (1992-2005). Online Article at Developer Dot Star. URL: http://www.developerdotstar.com/mag/articles/reeves_design_main.html
- [55] R.M. Greenwood, I. Robertson, R.A. Snowdon, B.C. Warboys. **Active Models in Business**. Proceedings 5th. Conference on Business Information Technology CBIT '95.
- [56] Oliver Radfelder, Martin Gogolla: **On Better Understanding UML Diagrams through Interactive Three-Dimensional Visualization and Animation**, ACM Press, New York, 2000.
- [57] Grieskamp, W.; Lepper, M.; **Using use cases in Executable Z**. Formal Engineering Methods, 2000. ICFEM 2000. Third IEEE International Conference on 4-6 Sept. 2000 Page(s):111 – 119.

- [58] **OMG MDA Guide**. Version 1.0.1 (03-06-01). URL: <http://www.omg.org/docs/omg/03-06-01.pdf>.
- [59] Jens Bæk Jørgensen, Claus Bossen, **Executable Use Cases: Requirements for a Pervasive Health Care System**, *IEEE Software*, vol. 21, no. 2, pp. 34-41, Mar/Apr, 2004.
- [60] Dinh-Trong, S. Ghosh, R. B. France, M. Hamilton, and B. Wilkins (2005), **UMLAnT: An Eclipse Plugin for Animating and Testing UML Designs**, Eclipse Technology Exchange Workshop, in conjunction with OOPSLA, San Diego, USA.
- [61] John M. Slaby and Steven D. Baker. **Model-Centric Software Development**. IEEE Computer, Feb 2006.
- [62] Behzad Karim. **Behavioral Software Architecture Language**. The Architecture Journal, Journal 6, 2006.
- [63] Robert Wigetman and Jurgen Moortgat. **Know Your UML with XML**. Oracle Magazine, Jan-Feb 2006.
- [64] Craig Larman. **Applying UML and Patterns**. Prentice Hall PRT, 3rd Edition, 2004.
- [65] Conrad Bock. **UML Without Pictures**. IEEE Software, September/October 2003.
- [66] **OMG MOF Core Specification Version 2**. URL: http://www.omg.org/technology/documents/formal/MOF_Core.htm.
- [67] Martin Fowler, Kendall Scott. **UML Distilled: A Brief Guide to the Standard Object Modeling Language**, 2nd edition. Addison-Wesley Professional. (August 25, 1999).
- [68] TouchGraph. URL: <http://www.touchgraph.com/>.
- [69] Siobhan Clarke and Elisa Baniassad. **Aspect-Oriented Analysis and Design: The Theme Approach**. Addison-Wesley Professional. (March, 2005).
- [70] Groovy Programming Language. URL: <http://groovy.codehaus.org/>.
- [71] UMLGraph. URL: <http://www.spinellis.gr/sw/umlgraph/>.
- [72] Scott W. Ambler. **The Object Primer: Agile Model-Driven Development with UML 2.0**. Cambridge University Press, 3rd Edition, 2004.
- [73] **CyberSource SDK for Java 3.7.12**, December 2005. URL: <http://www.cybersource.com>.
- [74] Cybersource Integration Options: http://www.cybersource.com/support_center/implementation/downloads/.
- [75] Eclipse. URL: <http://www.eclipse.org>
- [76] Aspect-Oriented Software Development. URL: <http://www.aosd.net>
- [77] Sequence Diagram Generator. URL: http://www.zanthan.com/itymbi/archives/cat_sequence.html
- [78] GraphViz. URL: <http://www.research.att.com/sw/tools/graphviz/>
- [79] Generating UML Use Case Diagrams with GraphViz DOT. URL: <http://www.iaa.upf.es/~dgarcia/DotUseCases/DotUmlUseCases.html>