2002

Incorporating Aspects into the Software Development Process in Context of Aspect-Oriented Programming

Mark Alan Basch
University of North Florida

Suggested Citation
https://digitalcommons.unf.edu/etd/112
INCORPORATING ASPECTS INTO THE
SOFTWARE DEVELOPMENT PROCESS IN THE CONTEXT
OF ASPECT-ORIENTED PROGRAMMING

by

Mark Alan Basch

A thesis submitted to the
Department of Computer and Information Sciences
in partial fulfillment of the requirements for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
DEPARTMENT OF COMPUTER AND INFORMATION SCIENCES

December 2002
ACKNOWLEDGEMENT

I wish to thank my thesis adviser, Dr. Arturo J. Sanchez, for his assistance and patience in guiding me through the thesis process.
The thesis "Incorporating Aspects into the Software Development Process in the Context of Aspect-Oriented Programming" submitted by Mark Alan Basch in partial fulfillment of the requirements for the degree of Master of Science in Computer and Information Science has been

Approved by the thesis Committee

Signature Deleted

Arturo S. Sanchez
Thesis Adviser and Committee Chairperson

Signature Deleted

Robert F. Roggio

Signature Deleted

Neal S. Coulter

Date

12/12/02

12/10/02

12/12/02

Accepted for the Department of Computer and Information Sciences

Signature Deleted

Judith L. Solano
Chairperson of the Department

Date

12/11/02

Accepted for the College of Computing Sciences and Engineering

Signature Deleted

Neal S. Coulter
Dean of the College

Date

12/12/02

Accepted for the University

Signature Deleted

Thomas S. Serwatka
Dean of Graduate Studies

Date

12/12/02

iii
CONTENTS

List of Figures...........................................................................................................viii
Abstract.......................................................................................................................ix
Chapter 1: Introduction.........................................................................................1
Chapter 2: Aspect-Oriented Programming.........................................................4
  2.1 History of AOP Research..............................................................................4
  2.2 Description of AOP......................................................................................7
    2.2.1 AspectJ.................................................................................................10
    2.2.2 Other Aspect-Oriented Languages......................................................14
Chapter 3: The Software Development Process.............................................17
  3.1 The Unified Process.....................................................................................18
    3.1.1 History of the Unified Process..........................................................18
    3.1.2 Description of the UP........................................................................19
  3.2 Agile Processes...........................................................................................22
    3.2.1 Extreme Programming.......................................................................24
  3.3 The Unified Process and Extreme Programming......................................25
Chapter 4: The Unified Modeling Language...................................................27
  4.1 History of UML............................................................................................28
CONTENTS (continued)

4.2 UML Basic Concepts .......................................................... 29
  4.2.1 Static View .......................................................... 30
  4.2.2 Use Case View ..................................................... 31
  4.2.3 Interaction View ................................................... 32
  4.2.4 State Machine View .............................................. 32
  4.2.5 Other Views ........................................................ 33
  4.2.6 Extensibility ........................................................ 33

Chapter 5: Aspects in the Software Development Process ............... 35
  5.1 Aspects Created in the Unified Process ................................ 36
    5.1.1 Aspects in the Inception Phase .................................. 36
    5.1.2 Aspects in the Elaboration Phase ............................... 39
    5.1.3 Aspects in the Construction Phase ............................. 43
    5.1.4 Aspects in the Transition Phase ............................... 46
  5.2 Aspects Created in Extreme Programming ........................... 47

5.3 Chapter Summary .................................................................. 52

Chapter 6: Modeling Aspects in the Unified Modeling Language ......... 54
  6.1 Aspects and UML Proposals from the Netherlands Conference ....... 55
    6.1.1 Aspects as Packages .............................................. 55
    6.1.2 Using Extensibility Mechanisms .................................. 56
CONTENTS (continued)

Appendix B: UML Specifications of Join Points.............................................113
Appendix C: UML Specifications of Aspects.................................................116
Vita..............................................................................................................119
LIST OF FIGURES

Figure 2.1: Cross-cutting aspect ........................................................... 7
Figure 2.2: Output of Hello World AspectJ program ......................... 12
Figure 3.1: The Unified Process ............................................................ 21
Figure 4.1: Class diagram ................................................................. 30
Figure 4.2: Use case diagram ............................................................. 31
Figure 4.3: Collaboration diagram ...................................................... 31
Figure 4.4: Sequence diagram ........................................................... 32
Figure 4.5: Statechart diagram ........................................................... 33
Figure 6.1: Aspect packages .............................................................. 67
Figure 6.2: Aspect class diagram ....................................................... 71
Figure 6.3: Sequence diagram with notes indicating aspects ............... 74
Figure 6.4: Sequence diagram with aspect objects ............................ 75
Figure 6.5: Collaboration diagram with aspect objects ....................... 76
Figure 6.6: Sequence diagram with Before and After aspect advice ...... 77
Figure 6.7: Sequence diagram with aspect and join point elements ...... 79
Figure 6.8: Aspect sequence diagram ............................................. 80
Figure 6.9: Statechart diagram with links to aspects ......................... 83
Aspect-oriented programming is a relatively new approach to programming that is design to resolve issues of separation of concerns. Rather than focusing on commonality of objects, as in object-oriented programming, aspect-oriented programming focuses on commonality of concerns, or more precisely as described by experts in the field, of “cross-cutting concerns.” These are aspects that cut across different modules of a program, such as security and authentication issues.

While much research in the area has focused on developing programming languages, little attention has been given to dealing with aspects in the software development process and the Unified Modeling Language. This thesis will examine how aspects are created during the software development process and how they can be modeled in the UML.
Chapter 1

INTRODUCTION

Much attention in computer programming techniques these days focuses on object-oriented programming, the practice of creating a program by identifying objects and developing classes related to the objects. But very recently, some programming experts have suggested a supplementary approach called aspect-oriented programming. Rather than focusing on commonality of objects, aspect-oriented programming focuses on commonality of concerns, or more precisely as described by experts in the field, on “cross-cutting concerns.” These are aspects that cut across different modules of a program, such as security and authentication issues. Proponents of aspect-oriented programming, or AOP, argue that it improves efficiency of coding by making the program more modular [Elrad01B].

Since this is a relatively new field, much of the AOP research to date has been focused on developing programming languages that can accommodate aspect-oriented techniques, but attention toward incorporating AOP into the software development process is just beginning. For instance, there are no standards for representing aspects in the Unified Modeling Language, or UML [Rumbaugh 99]. Within the past year, there have been workshops on aspect-oriented modeling at the 1st International Conference on Aspect-Oriented Software Development in April and
at the Fifth International Conference on the Unified Modeling Language in September. With research in the early stages, there is no consensus on how aspects should be modeled in the UML.

While aspect-oriented programming is a new and even unknown concept to many professional software developers, its adoption into the mainstream is already being touted by some observers. It was identified by the MIT Technology Review as one of “ten emerging areas of technology that will soon have a profound impact on the economy and how we live and work [Demeter02],” stating that “widespread adoption of aspects holds out the promise of less buggy upgrades, shorter product cycles and, ultimately, better and less expensive software [Tristram01].”

While AOP’s impact on the world is still debatable, it is worthwhile to develop standards for developing aspects and modeling them in the UML.

This thesis examines how software developers can express aspects in the software development process and how aspects should be modeled in the UML. It will include a proposal to incorporate aspects in the next revision of the UML standards. The thesis focuses on the well-known Unified Process [Arlow02], but it also addresses other development processes such as Extreme Programming [Wake02] and other Agile Software Development processes [Cockburn02].
The thesis will trace the textbook example of a real estate system [Kulak00] as it goes through the software development process and demonstrate how the AOP-related elements of the system are modeled in the UML.
Chapter 2

ASPECT-ORIENTED PROGRAMMING

Aspect-oriented programming, or AOP, is a programming technique that has been developed over the past decade. It aims to increase efficiency in coding by offering a method of dealing with concerns that are scattered repeatedly throughout a system across different modules, resulting in “tangled” code [Kiczales97]. Examples of these concerns include areas such as security, logging, error handling or synchronization that are re-used at various points in the program. These concerns are said to be “cross-cutting” because they cut into various modules of a system, such as classes and objects of an object-oriented program.

In the field of aspect-oriented programming, these cross-cutting concerns are known as “aspects.” The aspects are separated from the main program, thus untangling the code and providing better modularity and an enhanced ability to reuse the aspects [Elrad01B].

2.1 History of AOP Research

In the late 1980s and early 1990s, several research groups were studying programming approaches to solve the problems of separation of concerns. Those
groups included the Trese Group at the University of Twente in the Netherlands, which developed an approach called Composition Filters; a team at the IBM Thomas J. Watson Research Center that developed an approach called Multidimensional Separation of Concerns; and the Demeter research team at Northeastern University that developed Adaptive Programming.

The Demeter group did much early research on the problem of separation of concerns and in 1987 proposed the “Law of Demeter,” which states “each unit should have only limited knowledge about other units: only units closely related to the current unit.” The law’s motto is “only talk to your immediate friends” [Demeter02].

The group introduced Adaptive Programming in 1991 [Connections02], a technique in which programs are decomposed into cross-cutting building blocks.

The Demeter group published a 1995 paper [Hursch95] that identified “a new paradigm in software engineering called separation of concerns.” Although the paper does not use the term “aspect-oriented,” Karl Lieberherr, the leader of the Demeter team, refers to this paper as one of the early works on aspect-oriented programming [Lieberherr00].

The Trese Group began working on its Composition Filters approach to handling cross-cutting concerns in the late 1980s. The group described its approach as early as 1992 [Aksit92].
The term “aspect-oriented programming” and much of the development of AOP is credited to Xerox’s Palo Alto Research Center, or PARC, which began working on the discipline in 1992 [Morales02]. Much of the current research in the field points back to a seminal paper [Kiczales97] presented by the PARC group in 1997 to the European Conference on Object-Oriented Programming, or ECOOP. The paper describes aspects as issues that address “design decisions” that are difficult to capture with traditional object-oriented programming techniques. And it says the aspects are difficult to capture because “they cross-cut the system’s basic functionality.”

The paper breaks down elements of a program into two terms: components, which are elements that “can be cleanly encapsulated in a generalized procedure” and aspects, which cannot be cleanly encapsulated. In simplified terms, the component can be thought of as a module of the main program, while the aspect is a separate module that interacts with components repeatedly throughout the main program.

The goal of AOP, according to the PARC team [Kiczales97], is “to support the programmer in cleanly separating components and aspects from each other.”

Research in AOP increased considerably after publication of PARC’s 1997 paper. IBM’s approach to aspect-oriented issues, called Multidimensional Separation of Concerns, was introduced in 1999.
Research in the field has advanced further in the past two years, with the 1st International Conference on Aspect-Oriented Software Development held in the Netherlands in April 2002. The second annual AOP conference is scheduled to be held in Boston in March 2003. Additionally, major software development-related conferences such as ECOOP, Object-Oriented Programming, Systems, Language and Applications (OOPSLA) and the International Conference on Software Engineering regularly feature AOP-related papers in their proceedings.

2.2 Description of AOP

A further illustration of aspects as cross-cutting concerns can be found in a description offered by PARC team leader Gregor Kiczales in [Elrad01B]. It is illustrated in the simple diagram of a figure editor depicted in Figure 2.1.

Figure 2.1: Cross-cutting aspect
The illustration shows a figure editor that is used to draw points and lines. The drawing of a point or a line is contained in separate classes, called Point and Line. The program includes a concern that every time a new point or line is drawn, the user will be notified. This is the concern called DisplayUpdating. Since this concern is invoked if either a point or line is drawn, the DisplayUpdating concern cross-cuts the point and line classes. This becomes an aspect.

In most research about AOP, examples of aspects include concerns such as error handling functions, synchronization, logging or security, such as authenticating a user when a database is accessed. But aspects are not limited to those concerns. According to PARC's definition of aspects in its 1997 paper [Kiczales97], any concern that cross-cuts a component and "can not be cleanly encapsulated in a generalized procedure" can be programmed as an aspect.

In a traditional program, there are at least two ways the DisplayUpdating concern (or any other aspect) can be handled. In one way, the DisplayUpdating code can be rewritten in every class in which it cross-cuts. In the above example, it would be written once for the line class and repeated in the point class.

This creates the problem referred to as "tangling" [Miller01]. Simply put, the code becomes a tangled mess with too many concerns running together in one module, making it difficult to trace and understand.
Related to tangling is the concept of “scattering,” [AspectJ02] which refers to the same code being scattered throughout different modules of the program. Problems arise when the scattered code needs to be changed, and the programmer has to find every instance of the code and ensure that the changes made are identical.

The other way the cross-cutting concern can be handled is to create a separate function for the concern. The main program can then use the concern with a simple function call. But a function call can be complicated because of the parameters that are passed between the main program and the function. The programmer has to ensure that the correct form and number are passed to the function and if the parameters of the function are altered, the programmer must find every function call and re-check the parameters.

By separating the cross-cutting concern into an aspect, the code is simplified and modularity of the program is improved. The component (the main program module) makes no reference to the aspect being called, so the main functions of the program can be traced without interference from the aspects. However, at certain points in the component, an aspect will be invoked, although it is invisible to the main program.

These points where the aspects are instantiated are referred to as “join points” [Elrad01B]. They are simply a designation of where the aspects cross cut the components. The complication in aspect-oriented programming is developing
techniques in which the aspects and components can properly refer to the join points, so that the aspects are invoked at the proper times.

The aspects and components are joined together in a software system through the use of an “aspect weaver” [Kiczales97]. The aspect weaver is basically a compiler that weaves aspects and components together into a single, executable program.

There are several AOP languages under development that include a compiler that will perform the aspect weaving. The most well-known language and probably the most advanced is being developed by the PARC researchers, and it is known as AspectJ [AspectJ02].

2.2.1 AspectJ

AspectJ is an extension of Java that is intended to be easily learned and used by Java programmers. As Kiczales said in an interview:

“Normally when people have a new programming paradigm, they make a whole new language. But by putting it in Java, a Java programmer can become an aspect-oriented programmer in 15 minutes” [Morales02].

Any Java program can be cleanly compiled by the AspectJ compiler, but an AspectJ program containing aspects can not be compiled by a Java compiler.
AspectJ consists of three main elements: [AspectJ02]

- Join points are the points in the Java program where an aspect is called and executed.
- Pointcuts are references in the aspect to the join points. Basically, while the joint point is the point in the program where the aspect and component cross-cut from the component’s (or main program’s) view, the pointcut is the cross-cutting point from the aspect’s view.
- Advice is the code in the aspect that is executed when the main program reaches a join point.

Here is a simple AspectJ example based on a similar example from a recent research paper [Laddad02B]. It is based on the traditional “Hello World” program and begins with a HelloWorld Java class that has two methods that print messages:

```java
public class HelloWorld {
    public static void sayHello() {
        System.out.println("Hello World");
    }
    public static void sayItAgain() {
        System.out.println("Hi again");
    }
}
```

The program is complicated by an aspect in AspectJ that incorporates tasks to print “Begin message” before any message is printed and “End message” when the message is completed. The following coding represents the aspect.
The first line of the code above declares an aspect called PerformTask, similar to the way a Java class is declared. The second line declares the pointcut. The information on the left side of the colon defines a pointcut named callPerformTask. The information on the right side of the colon tells us this aspect will be invoked every time a public static method beginning with say is called.

The rest of the aspect consists of the advice. The before code indicates code that will be executed as soon as the joint point is reached, before the code in the main

```java
public aspect PerformTask{
    pointcut callPerformTask () : call(public static void say*());

    before() : callPerformTask() {
        System.out.println("Begin message");
    }

    after() : callPerformTask() {
        System.out.println("End message");
    }
}
```

Figure 2.2: Output of HelloWorld
AspectJ program
program is executed. The after code is executed once the code in the main program is executed. Figure 2.2 shows what the output of this AspectJ program would be.

While AspectJ is often demonstrated with before and after advice, there is no mandate that an aspect have either or both. There is also a third type of advice, around [AspectJ02], which basically executes around the component, and there are variations on both before and after. There are no required types of advice that must be used in AspectJ.

The AspectJ joint points are most commonly found at method calls in the main Java program, but joint points can also be found at constructor calls, read/write access to a field, exception handler execution and object and class initialization [Laddad02B].

The PARC group is continuing to develop AspectJ. The latest version as of this writing is release 1.0.6 and is available for free download from the group’s web site at http://aspectj.org.

AspectJ is also getting support from other developers. For example, the Eclipse development environment [Eclipse02] is now including AspectJ as one of its supported languages.
2.2.2 Other Aspect-Oriented Languages

AspectJ is the most widely-used aspect-oriented language, and much of the research concerning AOP in the software development process and the Unified Modeling Language is based on programming with AspectJ. However, there are other aspect-oriented languages and techniques in various stages of development.

AspectJ was designed as a compatible extension to Java [AspectJ02], and some of the developers of that language have naturally extended their work to other languages. For example, AspectC was designed from the non-object-oriented areas of AspectJ [AspectC02] and is being developed from there. It is not publicly available as of this writing.

That group also has used the AspectJ design to develop AspectC++. A prototype of AspectC++ is available [AspectC++02] but is not ready for use in software development.

Outside of the work of the PARC group, the most well-known aspect-oriented tool is HyperJ, which is being developed by the aforementioned IBM researchers. HyperJ is a tool that addresses “Multidimensional Separation of Concerns,” or MDSOC. HyperJ decomposes a program into separate concerns [HyperJ01]. It makes no distinction between classes and aspects [Ossher01]. The separate concerns are known as “hyperslices.”
Each hyperslice can be implemented as a stand-alone software module. This is a difference between HyperJ and AspectJ. In AspectJ, the aspects only function as add-on modules that are weaved into an application [Miller01].

For example, the IBM developers describe a corporate organization system in which information about employees, such as job titles (personnel feature) and salaries (payroll feature) are kept [Ossher01]. While the personnel and payroll features could be described as cutting across a class called “employee,” in HyperJ each feature – employee, personnel and payroll – is modeled as a separate hyperslice that may or may not overlap with other hyperslices.

HyperJ was designed not as an extension to Java but as a tool that can be used to adapt any off-the-shelf Java component [Ossher01]. It is written in standard Java and is also available for free download [HyperJ01].

The Demeter group’s Adaptive Programming technique is, in the group’s own words, a “special case of Aspect-Oriented Programming” [Connections02]. In this technique, an adaptive method (its version of an aspect) contains definitions of join points and instructions that will be performed at those join points [Lieberherr01]. The technique uses a traversal strategy to weave the join points in the program.
Like HyperJ, Adaptive Programming is not an extension of Java but offers a Java package called the Demeter/Java, or DJ, library that, the group says, “covers a large portion of real-world programming tasks” [Liberherr01].

The Trese group’s Composition Filters approach provides filters as an enhancement to objects. Each filter represents an aspect [Trese01]. The filters are described as modular, in that they are independent of the implementation of the objects they are attached to, and orthogonal, in that they are independent of other filters. The group says those two properties increase adaptability and reuse of the aspects, as filters [Bergmans01].

A comprehensive list of AOP languages and techniques can be found at the Aspect-Oriented Software Development web site at www.aosd.net.

While the research groups listed here have been studying AOP-related issues for at least a decade and there are a number of language projects underway, the AspectJ language developed by the PARC team is the most advanced and the most widely used. So AspectJ will be referred to often in this thesis to illustrate aspects in the software development process.
Chapter 3

THE SOFTWARE DEVELOPMENT PROCESS

A software development process uses the best practices in the industry, combining technology, tools, people and organizational patterns, to efficiently develop quality software [Rumbaugh99].

At the current time, aspect-oriented programming is generally not part of the mainstream software development process, because the programming techniques it advocates are not in mainstream use yet. But if one is to assume that aspect-oriented programming will be incorporated into the programming world as a standard practice, it stands to reason that modeling and developing aspects will become a standard practice in software development processes.

In order to incorporate AOP into software development, it is necessary to look at software development processes in use today. One widely-known process is the Unified Process, but other so-called Agile Processes such as Extreme Programming are also gaining popularity. This thesis will examine aspects in both the Unified Process and Extreme Programming development processes.
3.1 The Unified Process

The Unified Process, or UP, is a development process that describes a set of activities needed to develop a user’s requirements into a software system. The UP is use-case driven, architecture-centric and iterative and incremental. It uses the Unified Modeling Language (which will be described in Chapter 4) to model the system [Rumbaugh99]

3.1.1 History of the Unified Process

The roots of the UP trace back to work first unveiled at Ericsson in 1967 [Arlow02], which used a set of interconnected blocks to diagram software systems. The blocks were created from so-called “traffic cases” (which evolved into what are now known as “use cases”), which described how the system would be used.

In 1976, the Specification and Description Language, or SDL, was issued by the CCITT, the international body for standardization in the telecommunications field [Rumbaugh99]. The SDL, which was influenced by the Ericsson design system, used a set of components that communicated with each other by sending signals.

In 1987, Ivar Jacobson left Ericsson and founded a new company called Objectory AB, which developed a software development process called “Objectory” (short for Object Factory) [Rumbaugh99].
Objectory was acquired in 1995 by Rational Software Corp. This software development process, as it evolved, was known as the Rational Objectory Process from 1995 through 1997. In 1998, the name was changed to Rational Unified Process, reflecting the fact that Rational had acquired a number of different companies and incorporated their work into its own unified software development process [Rumbaugh99].

The Rational Unified Process, or RUP, is commonly used and referred to in software development studies. But the RUP refers to a proprietary software development process owned by Rational. This thesis will use the Unified Process, or UP, which is closely related to RUP but is a separate, generic process.

3.1.2 Description of the UP

According to the three Rational developers who are credited as the primary inventors of the UP (Jacobson, James Rumbaugh and Grady Booch), there are three characteristics of the UP that set it apart from other processes: it is use-case driven, architecture-centric and iterative and incremental [Rumbaugh99].

Use cases constitute an analysis tool that can be used to elicit software requirements from user expectations by showing what a software system is expected to do through use scenarios. It should describe what a user puts into the system and what comes out when the user does something [Kulak00]. In other words, it captures the requirements
of a system by describing the users' interactions with the system. By saying the Unified Process is use-case driven, that means the ultimate deployment of the system is driven by what is specified in the use cases.

The UP is architecture-centric because the system's architecture is a fundamental element that is taken into consideration for all development decisions as the system evolves.

The UP is iterative and incremental in that it repeatedly goes through a series of complete "mini-projects" [Rumbaugh99] as the system evolves through a series of cycles that culminate with a new release of the software.

Each cycle is divided into four phases and each phase is divided into iterations [Rumbaugh99]. The phases are:

- Inception, where an idea is developed into a business case;
- Elaboration, where the use cases are specified and the system architecture is designed;
- Construction, where the system is built; and
- Transition, where the system moves into release.
Each phase can have one or more iterations (mini-projects), and the iterations include five so-called core workflows: requirements, analysis, design, implementation and test [Arlow02].

Figure 3.1 is an oft-repeated illustration of the Unified Process. For example, it shows that there is more emphasis on requirements and analysis in the elaboration phase than in other phases. And most of the implementation occurs in the construction phase.

This thesis will examine a project as it goes through the phases and iterations and demonstrate how aspects are unveiled during the project’s life cycle.
3.2 Agile Processes

An agile software development process is one that can be described as “both light and sufficient” [Cockburn02]. A description of agile processes can be best explained by looking at a manifesto issued by the Agile Alliance, a group of 17 developers of various agile processes who met in February 2001 and agreed, in the manifesto, on four core values of agile development [Agile02]:

- “Individuals and interactions over processes and tools;”
- “Working software over comprehensive documentation;”
- “Customer collaboration over contract negotiation;” and
- “Responding to change over following a plan.”

Beyond the four core values, the alliance also agreed on 12 principles: [Agile02]

1. “Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.”
2. “Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage.”
3. “Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.”
4. “Business people and developers must work together daily throughout the project.”
5. “Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.”

6. “The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.”

7. “Working software is the primary measure of progress.”

8. “Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely.”

9. “Continuous attention to technical excellence and good design enhances agility.”

10. “Simplicity – the art of maximizing the amount of work not done – is essential.”

11. “The best architectures, requirements and designs emerge from self-organizing teams.”

12. “At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.”

In short, agile development emphasizes development teams working together in constant communication, including communication with the users, and turning out new software releases as quickly as possible.

There are various agile methods in use, with names such as Adaptive Software Development, Scrum, Crystal, Feature-Driven Development and Dynamic System
Development Method [Cockburn02]. This thesis will look at one of these methodologies, known as Extreme Programming.

3.2.1 Extreme Programming

Extreme Programming is a relatively new software development process that emphasizes a repeated cycle of coding and testing to incrementally build a system. Once the code is working, a process called refactoring is used to attempt to improve the code design. No one developer “owns” the code, so any member of the programming team is free to change code when he or she feels it is necessary [Wake02].

The programming team, working in pairs, is put together in an open workspace to foster communication between team members. The customer is on site to interact with the programming team as much as possible. Extreme Programming also calls for a 40-hour work week to avoid burnout [Wake02].

The core values of Extreme Programming are simplicity, testing, communication and courage – that is, the courage to go ahead and make whatever improvements are deemed necessary [Cockburn02].

The Extreme Programming process begins with a “Release Planning Game” [Wake02], in which the customer writes “stories” on cards describing features of the
system. The customer and the programmers then work together to decompose and prioritize the stories and decide what is needed to implement them.

After the stories are written, the development team divides them into tasks, which are also written on cards, for the purpose of planning iterations. The individual programmers then estimate the tasks and decide which tasks they will implement in the iteration.

Although the term “Extreme” is used in today’s popular culture to describe something that is new and youth-oriented, proponents emphasize that Extreme Programming is derived from development practices that have been around for a long time [Highsmith00]. The difference, they say, is Extreme Programming is more flexible and more responsive to customers’ changing needs.

3.3 The Unified Process and Extreme Programming

Rational has produced a white paper [Smith02] to compare the Unified Process, as conceived in its Rational Unified Process, and Extreme Programming to “dispel the notion that XP is a lightweight, and therefore desirable, alternative to heavyweight RUP.”

The paper compares Extreme Programming’s story-writing phase to the inception phase of the Unified Process, with the stories substituting for use cases. It contrasts
the two processes by saying that Extreme Programming moves straight from
requirements gathering into planning for the first release, while the Unified Process
goes into the elaboration phase after inception in which the architecture is stabilized.

It says the UP’s construction phase is equivalent to Extreme Programming’s series of
product releases, although each iteration in the construction phase does not result in a
release to the customer. And Extreme Programming does not have a transition phase,
since the customer is constantly receiving new releases.

A major difference between the Unified Process and Extreme Programming is that
there are a far greater number of artifacts produced in the UP. Extreme Programming
de-emphasizes documentation.

The Unified Process is tied in with the Unified Modeling Language, while Extreme
Programming appears to have no modeling standards, or perhaps no need for them.
This thesis is intended to demonstrate how aspects are uncovered in the software
development purpose and then modeled in the UML, so much of the thesis will
concentrate on a project developed in the UP and modeled in the UML. But while it
will not offer modeling examples for Extreme Programming, it does consider a
project developed through Extreme Programming and how aspects would be unveiled
and developed in that process.
Chapter 4

THE UNIFIED MODELING LANGUAGE

The Unified Modeling Language, or UML, is a general-purpose visual modeling language that is used to document the design and analysis activities of a software system [Rumbaugh99]. While it is closely tied with object-oriented programming and with the Unified Process and some people consider the UP the preferred development process for use with UML, the UML is not tied to any particular software development process or paradigm [Arlow02].

The word “Unified” is used to denote that UML:

• Brings together historical methods and notations;
• Is used seamlessly through each phase of the development cycle;
• Is used across application domains;
• Is used across programming languages and platforms;
• Is used across software development processes; and
• Attempts to be consistent with its internal concepts [Rumbaugh99].
4.1 History of UML

The Unified Modeling Language was adopted as an industry standard relatively recently. A number of researchers were working on modeling languages for object-oriented development when, in 1994, James Rumbaugh, who had developed a language called Object Modeling Technique, or OMT, joined Rational Software Corp. There he worked with Grady Booch, who had developed a technique called the Booch method [Arlow02]. In 1995, Ivar Jacobson brought his Objectory software development process to Rational, and he joined with Rumbaugh and Booch in developing UML [Rumbaugh99]. These three were also the main developers of the Unified Process at Rational, which is why UML is so closely associated with UP.

In 1996, the Object Management Group requested proposals for a standard approach for object-oriented modeling. The three Rational developers, with input from other developers at other companies, refined the UML and submitted it to the OMG for approval in September 1997. It was adopted by the OMG as the standard modeling language in November 1997 [Rumbaugh99].
4.2 UML Basic Concepts

The UML is comprised of three building blocks [Arlow02]:

- Things, which are the modeling elements;
- Relationships, which specify how things are tied together;
- Diagrams, which are views that show collections of things.

The modeling constructs of the UML are classified into “views.” The classifications are listed in various ways, depending on the researcher. Once concept is known as the 4+1 architecture because it consists of four views, all integrated by a fifth view, which is the use case view [Arlow02]. That view consists of use case diagrams and interaction diagrams.

The other four views are the logical view, which consists of class diagrams, statecharts and object diagrams; the process view, which has class diagrams and object diagrams; the implementation view, which has component diagrams; and the deployment view, which has deployment diagrams.

The three Rational developers, in their UML reference manual [Rumbaugh99], give a different description of the views. They divide them into four main areas:
• Structural area, which describes things in the system and their relationship with other things. This area includes the static view, use case view, implementation view and deployment view;

• Dynamic area, which describes the system behavior. It includes the state machine view, the activity view and interaction view;

• Model management area, which organizes the models and crosses the other views; and

• Extensibility area, which gives developers the ability to extend the UML.

4.2.1 Static View

The static view models classes and their relationships through a class diagram. The classes are drawn as rectangles, as shown in Figure 4.1.¹

Figure 4.1: Class diagram

¹ All UML diagrams in this chapter represent screen shots taken from a sample project in Together Control Center, a CASE tool for modeling a system with the UML.
4.2.2 Use Case View

The use case view, as shown in Figure 4.2, models the system as outside users interacting with the system. The users are called actors.
4.2.3 Interaction View

The interaction view shows the flow of control across many objects through a sequence of message exchanges. It can be modeled as a collaboration diagram, as in Figure 4.3, which models the objects and links of an interaction. Or it can be modeled as a sequence diagram, as in Figure 4.4, which shows a set of messages in sequence.

4.2.4 State machine view

The state machine view models a set of objects at a period in time in which a certain event occurs. It can be modeled with a statechart diagram, as in Figure 4.5.
4.2.5 Other Views

Other views include the activity view, which uses an activity diagram to describe an operation; the implementation view, which uses a component diagram to display the software units of an application; the deployment view, which uses a deployment diagram to show the nodes of a system; and the model management view, which models the system as a set of packages.

4.2.6 Extensibility

The UML has three extensibility mechanisms [Arlow02]:

- Constraints, which use a text string to specify rules about a modeling element;
• Stereotypes, which allow developers to define a new modeling element as a
  variant of an existing element. The stereotype is indicated with a
  «stereotype» designation; and

• Tagged values, which allow developers to add properties to a modeling
  element.

The extensibility mechanisms give developers flexibility to tailor a model for their
own needs.
In order to determine how aspects arise in the software life cycle, this thesis will go through the steps of the Unified and Extreme Programming processes and indicate points along the path of a project where a developer would determine aspects are relevant in connection with the solution domain.

In addition to generically describing the process, in this chapter we will use a classroom project that was studied in a two-semester Engineering of Software class at the University of North Florida’s CIS Department in the fall semester of 2001 and spring semester of 2002. The project is based on a textbook case described in [Kulak00, pp. 175-250].

The case is an application for a real estate agency that wants a system that can list properties for sale and let buyers and sellers, as well as real estate agents working for the company, list properties up for sale, peruse the list, and work through the process of selling a property. The case study, much of which was outlined in the textbook but was also expanded by the students, includes a number of use cases, a list of business rules and a list of nonfunctional requirements. The full set of these documents can be found in Appendix A.
Developing use cases from meetings between developers and users is the first step in creating an application in the Unified Process, so the use cases already developed in this case study are a good starting point for unveiling aspects in the software development process through the Unified Process.

Although Extreme Programming uses “stories” instead of use cases for requirements gathering, the information in the use cases will also be the basis for examining aspects in the Extreme Programming process.

5.1 Aspects Created in the Unified Process

As discussed in Section 3.1, the Unified Process consists of four phases. Each of the phases includes one or more iterations through the five main workflows. This thesis will go through the four phases in order to see where aspects would be created during the software development process.

5.1.1 Aspects in the Inception Phase

The goal of the Inception phase of the Unified Process is to develop a business case that justifies moving forward with the project [Jacobson99]. Much of the activity centers on initial requirements gathering, and while some decisions may be made about architecture, there is relatively detail of the functional requirements considered in this phase. So only a small number of aspects will be identified in this phase.
There would likely be one iteration through the five workflows in the Inception phase. Most of the work, as illustrated in Figure 3.1 in Chapter 3, would come in the first workflow, which is the gathering of requirements. This workflow consists of identifying actors who interact with the system and use cases that illustrate the functional requirements of the system. This workflow will also create the list of nonfunctional requirements. While the use cases are not completed in the Inception phase, making it difficult to make determinations about functional requirements, a developer may make the decision early on that a nonfunctional requirement will be designed as an aspect.

The analysis workflow in this iteration is where a developer may begin to consider aspects from the list of nonfunctional requirements. Although very few details about the system implementation are considered at this point in the process, the developer may make a "first rough cut" [Rumbaugh99] at an analysis model by identifying classes and packages. Therefore, potential aspect packages could be identified at this point. As will be discussed in Chapter 6, aspects should be modeled as independent packages for the purposes of the Unified Modeling Language.

In examining the list of nonfunctional requirements for the real estate agency (Appendix A), two aspect candidates emerge. Requirement 12 sets certain performance levels for online response time that must be maintained by the system. If the developers decide that response time should be monitored from within the system, this could be modeled as an aspect.
The requirement mandates that agents should get a response within 10 seconds 90 percent of the time. At this point in the process, there may not be enough detail in the use cases to pinpoint every instance where an agent will need a system response. But the agent will be interacting with the system in several use cases, such as Close on Property, Offer Price on Property and Agree to Terms. Since response time will have to be monitored during each of these use cases, an aspect to monitor the time will cross-cut these concerns many times throughout the system.

Likewise, other users will interact with the system in several use cases, including Select Agent, Agree to Terms and Process Loan. The nonfunctional requirement mandates that external users receive a response from the system within seven seconds 95 percent of the time. An aspect to monitor response to external users will also cross-cut many concerns of the system.

The second aspect found in the list of nonfunctional requirements would come from requirement 15, which considers recoverability from a server crash. There will be many instances in the system where information is stored in a database, such as when a user lists a property, makes an offer on a property or closes a property. An aspect to create an audit trail that tracks database transactions will also cross-cut with those system functions.

It should be noted that we consider these as “candidates” for aspects at this stage of the process, because it may be determined later during design, implementation or test
workflows in this phase or later phases that these features are better handled without aspects.

Moving on with the single iteration of the Inception phase, the design workflow is concerned with determining the architectural design of the system. So there would be no consideration of possible aspects in this workflow.

The implementation and test workflows are generally not used in this phase, unless a minimal prototype must be built to confirm the viability of the architecture.

5.1.2 Aspects in the Elaboration Phase

While developers in the elaboration phase continue to capture requirements for the system, the principal objective of this phase is to form the architecture baseline for the project [Rumbaugh99]. Most of the analysis and design decisions made during this phase are made to the extent that they are necessary to establish that architecture baseline, not in the interest of efficiency of coding. So searching for aspects in this phase is not a primary concern. Yet, some aspects can potentially be unveiled here.

By the end of the elaboration phase, about 80 percent of use cases should be identified but as little as half of those will be described in detail and even fewer than that may be fully analyzed [Rumbaugh99]. However some steps will be taken in this phase toward separation of concerns.
In the requirements workflow, as use cases are identified, the developer will look for certain redundancies. These redundancies could be modeled as "include" associations, in which the redundant procedure becomes a separate use case that is included by association with other use cases. At this point in the process, the developer may not have enough detail about the use cases to determine whether these types of relationships exist. So in the elaboration phase, the developer is looking for obvious redundant features.

These features may be candidates for aspects, but they are more likely to be complex procedures that are a central feature of the software system, so they are very likely to be modeled as functions or methods in the main program.

An example of this may be a retail establishment that makes several types of transactions with its customers. Regardless of the type of transaction, the establishment would go through the same type of procedure to process the customer's payment, so "Process Payment" would become a use case that is included in all of the transaction use cases. That would be obvious even if the use cases, as written at this point in the elaboration phase, did not detail the payment process. So the process would be separated from the transaction use cases. However, the complexity of "Process Payment" would make it an unlikely candidate for an aspect.
In looking at the use cases for the real estate agency in Appendix A, no obvious candidates for “include” associations emerge and no new candidates for aspects emerge at this point.

In the elaboration phase, the analysis workflow is the likely point where aspects would be identified. As part of the architectural analysis, the developer in this phase will identify “generic analysis mechanisms.” These mechanisms include [Rumbaugh99]:

- Persistence;
- Distribution and concurrency;
- Security features;
- Fault tolerance; and
- Transaction management.

These are the concerns that are typically associated by researchers with aspects. Even without detailed procedures outlined in use cases, which have not yet been completed at this stage, the developer can reasonably assume that some of these generic mechanisms will be needed and that they will be modeled as aspects.

In the case of the real estate agency, we have already determined some of these mechanisms in Section 5.1.1 from the list of non-functional requirements. These aspects were for performance time and fault tolerance.
While it was not on the list of non-functional requirements, at this point we will add another mechanism that should be modeled as an aspect – a security function. There is a separate use case to authenticate users when they enter the system. However, once users are in the system, there should be another concern that ensures that they only have authorized access to certain features. For example, a potential home purchaser should not have access to personal records of the seller that may be in the database, and vice versa. A security concern, modeled as an aspect, would ensure that the user has the proper clearance to view certain records or make certain transactions. This will cross-cut many concerns in the system.

In the design workflow of the elaboration phase, while developers may not have much detail to work with from the use cases, they do have enough information to continue identifying classes and potential packages and relationships between those classes and packages. Again, most of the work in this phase is concerned with establishing the architecture, so there is no mandate to create detailed designs of the aspects at this point. But at least some aspects can be identified and separated.

The generalized information in the real estate agency use cases that the developer would have at this point produces no additional candidates for aspects.

Just as in the inception phase, the implementation and test workflows are minimal in the elaboration phase. Some significant use cases may be implemented, but the
aspects would not be considered significant to the business case, which is still being established at this point, to make further work on aspects worthwhile in this phase.

It should be noted that the elaboration phase may contain two or more iterations through the five workflows. As the developers go through the iterations, they may identify more aspects on follow-up passes through the analysis phase.

5.1.3 Aspects in the Construction Phase

The purpose of the construction phase is to produce a “beta release” version of the software system [Jacobson99]. In the first two phases, much of the groundwork was laid for the project but much of the detail was skipped. In the construction phase, most of the remaining details of the use cases are fleshed out, which gives the developer an opportunity to examine detailed procedures of the system and determine where the use of aspects would be appropriate.

In this phase, there are likely to be several iterations through the five workflows as the beta version of the system is built in small increments. So there may be several points in the phase where aspects are discovered and created. But for the purposes of our example, we will go through each workflow once.

In the construction phase, the requirements gathering and analysis, which were the major workflows of the first two phases, are minimized, as indicated in Figure 3.1.
Early in the construction phase, the design workflow is featured but toward the end of the phase, the emphasis is on implementation.

Since the requirements and analysis workflows are largely complete by this phase, it is unlikely that new aspects will be found during those workflows in the construction phase iterations. Most of the use cases have already been identified (in requirements) and the architecture of the system (in analysis) is basically already determined.

In the design workflow of the construction phase, the developer will be studying the details of the use cases that have been overlooked, to an extent, in previous phases. So this is an opportunity to uncover aspects.

In looking through the details of the use cases for the real estate agencies, three candidates for aspects emerge.

The first aspect here would be a “no agent” warning. One of the use cases is a “Select Agent” case in which a buyer or seller of property chooses a real estate agent, who works for the agency, to be his or her representative. However, the system also allows buyers and sellers to act without an agent, with acknowledgement that the use of an agent can be advantageous in working out a transaction.

At various points in the system, a buyer or seller may attempt to negotiate a contract or close a deal without an agent. That may be an oversight on their part. The party
may want to bring in an agent for help at that point. The system could send the user a
warning message that he or she can still select an agent at this point. In fact, such a
warning is indicated in the procedures of the “Offer Price for Property” use case. That
would be modeled as an aspect that cross-cuts the system at many points.

As a buyer or seller goes through the transaction process through several use cases,
including making an offer, negotiating a contract, inspecting the property and closing
the deal, the details of the transaction (whether referred to as an offer or contract) go
through many changes. It may be worthwhile to log the changes by date and time and
who made the changes. The logging concern would ideally be modeled as an aspect
that is outside the flow of the system’s main procedures but cross-cuts many of the
procedures.

Another logging concern regards user access to the system. From a marketing
perspective, it would be worthwhile to log access to the system, so the agency knows
what outside users have entered the system and where and when. And from a “big
brother” perspective, the agency’s management might want to know how often the
agents, who are employees of the company, are accessing the system. This would also
be modeled as an aspect.

Following design, there will be much implementation and testing of the software in
this phase. As the developer attempts to implement and then test certain features, he
or she may find some concerns have been modeled inefficiently and may be better
designed as aspects. Conversely, the developer may find that an aspect that was modeled in the design workflow or in the earlier phases does not work well in the system. At this point, candidate aspects identified earlier would be created or dropped, and new aspects may emerge.

Since the real estate agency modeled in this thesis will not be implemented, we will have to assume that no aspects were found or deleted in the implementation and test workflows.

5.1.4 Aspects in the Transition Phase

The basic purpose of the transition phase is to get the software into the hands of the end-users and ensure the system works to their liking, and that the bugs are worked out. There is very little new analysis and design. However, users may belatedly decide there are new features they need after they get their hands on the beta release [Jacobson99]. For example, the logging aspects designed in the construction phase, if they were not in the beta release, are something users might decide they want, because they want to keep track of changes in contracts or the number of times customers access the system.

In the transition phase, these additional features, if they cross-cut many other features of the system, would ideally be modeled as aspects, because the aspects could be
added without disturbing the basic functionality of the main program. This is one of
the big advantages of using aspects.

In the transition phase, the five workflows are given minimal, if any attention. Most
of the work done by the developer would be in response to user needs. So aspects can
be added at any point in this phase.

Since the real estate agency in this example is hypothetical, we have no user
feedback. So no new aspects are added in this phase for that project.

5.2 Aspects created in Extreme Programming

Extreme Programming appears, at first glance, to be an unstructured approach to
developing software. But there are procedures that are followed and if the
programming team follows standard practices, the creation of aspects can still be an
orderly process.

Extreme Programming builds a system incrementally through a constant process of
implementing and testing, so aspects can be unveiled anywhere from the initial
planning to the final build. The process goes roughly like this:
- The customer writes a "story" describing a feature of the system.
- The programmers determine if the story is doable, or if it has to be split into smaller tasks.
- After a number of stories are gathered, the customer sorts the stories in terms of their value.
- The programmers select the tasks determined by the stories and begin the iterations to implement them.
- Once a section of code is successfully implemented and tested, the refactoring process begins in which programmers look for ways to improve the code.

Through this process, there are two points where aspects will emerge. The first is when the programmers take the stories and decide on their feasibility. The second, which is very obvious, is the refactoring process, which is designed to clean up code and solve the problems of code tangling and code scattering addressed by aspect-oriented programming.

As the development process begins and the customer hands stories to the programmers, the first thing the programmers do is "estimate" the stories – that is, determine how long it will take to implement them [Wake02]. If they don’t know how to implement them, they "do a spike," which is quick throwaway solution to the problem that is used to make the estimate [Wake02].
At this point, an aspect-oriented developer should be looking for aspects and will likely find them.

In order to examine how aspects will be found for the real estate agency example in an Extreme Programming development environment, we make the assumption that the customer’s stories would all be contained within the use cases in Appendix A.

For example, the two aspects created in the Inception phase of the UP, which all came from the list of non-functional requirements, would likely be created from the stories in the Extreme process. The non-functional requirements would probably be submitted as simple stories by the customer, as follows:

- Online response time for agents should be 10 seconds 90 percent of the time; Online response for external users should be within seven seconds 95 percent of the time.
- The system should be able to handle customers with poor Internet connections and recover from a server crash.

One difference in extreme programming is that the first aspect might be divided into two stories and eventually be programmed as two aspects.
Moving on to the one aspect created in the UP’s Elaboration phase, that security aspect would also emerge from the stories of the Extreme process. The security aspect ensured that once a user entered the system, that user only had access to data and features that were authorized for that type of user (an agent or a customer).

That system requirement was not included in the UP use cases. It was an aspect that the developer determined would be a necessary feature. So we might assume that the customer for the Extreme programmers did not write a story for it. However, while only customers write stories for the system, these stories can be derived from suggestions by the programmers. And the customer would probably decide that this security feature is a good idea.

The logging aspects that were created in the UP’s Construction phase might emerge in the same way in Extreme Programming. Although those aspects were not specified in the use cases (or in the original stories), the programmers might suggest them to the customer as good ideas that would make good stories.

The agent warning aspect from the Construction phase did emerge from the detailed look at the use cases, and we can assume it would be included in the customer stories and would be created as an aspect.

In the UP example, all aspects emerged as the developers examined the customer’s requirements for the system. However, in Extreme Programming, after the
requirements are implemented, the developers will have a second chance to hunt for aspects during refactoring.

Refactoring is a process of re-examining code, after it has been successfully implemented and tested, to determine if the code can be streamlined. There are several "danger signs" that the programmer might find in finished code, including [Wake02]:

- Classes that are too long;
- Methods that are too long;
- Duplicate code.

These danger signs, particularly the duplicate code, can indicate the presence of the code tangling or code scattering problems which aspect-oriented programming seeks to improve.

This is a significant advantage for Extreme Programming as a development process, in terms of aspect-oriented programming. In following the Unified Process, you would have fewer opportunities to uncover aspects by examining code. Most of the aspects will be created by modeling, rather than in the coding process. Refactoring is a major part of the Extreme process, which ensures that an aspect-oriented programmer will have plenty of opportunities to examine code for possible restructuring into aspects.
However, Extreme Programming has its drawbacks. A major problem could be its lack of documentation. In aspect-oriented programming, the UML models that demonstrate relationships between concerns are a very important tool to help developers use aspects consistently. Extreme Programmers are unlikely to formally use the UML to model their system, so it would be difficult for developers to visualize the concerns and their relationships.

5.3 Chapter Summary

In this chapter, we examined a software application that was developed for a real estate agency as part of a classroom exercise, looking for opportunities to create aspects that cross-cut other features of the system. We identified six aspects:

- A performance aspect that monitors system response time;
- A recoverability aspect that creates an audit trail to recover transactions in case of a system crash;
- A security aspect to ensure that only authorized users have access to certain features;
- An agent warning aspect that ensures users have an opportunity to use a real estate agent to assist in their transactions;
- A transaction logging aspect that tracks changes in real estate sales deals in progress; and
- A user logging aspect that keeps track of users accessing the system.
The identification of these aspects demonstrates that aspects can be found at several different stages of the software development process, in both the Unified Process and Extreme Programming.
Chapter 6

MODELING ASPECTS IN THE UNIFIED MODELING LANGUAGE

Over the past decade, most of the research into aspect-oriented programming has been focused on developing programming techniques to incorporate aspect-oriented principles into the final realization of software systems. However, until very recently, little attention was given to how to model aspects in the software development process. After all, without programming languages that support aspects, there was no reason to consider aspect-oriented development.

However, with the advancement of AspectJ and other aspect-oriented programming languages, we may be reaching the point where aspects will be incorporated into major software projects. The analysis and design of those projects will require modeling standards for aspects in the Unified Modeling Language. So within the past year, researchers have begun offering proposals for how to model aspects in the UML. As the Object Management Group considers a new release of the UML standards, from current version 1.4 to version 2.0, it is also appropriate to propose new modeling elements that accommodate aspects.

This chapter begins by reviewing the proposals that have been made to date. The proposals are compiled from the proceedings of two workshops that have been held
this year on the UML for aspects, at the 1st International Conference on Aspect-Oriented Software Development in the Netherlands in April and at the Fifth International Conference on the Unified Modeling Language in Germany in September.

After reviewing and critiquing proposals from those two workshops, this chapter will conclude with a new proposal for modeling aspects in the UML, based on research about aspects and the UML.

6.1 Aspects and UML Proposals from the Netherlands Conference

The Netherlands conference was the first major conference devoted solely to aspect-oriented programming, and it included a workshop dedicated to aspects and the UML. The proposals showed no clear consensus on standards for modeling aspects, but offered a number of ideas.

6.1.1 Aspects as Packages

Stephan Herrmann of Technical University of Berlin offered a proposal in which packages, rather than classes would be the main design entity [Hermann02]. Using packages, with relationships drawn between them to show the relationships between aspects and other concerns, allows for separation of concerns and re-usability of aspects.
Hermann also argued that design notations should be language-specific to allow developers “to smoothly move from analysis to implementation.” But at this time, no aspect-oriented language has been widely adopted, not even AspectJ. With AspectJ being the most advanced aspect-oriented language, it is useful to use it as a guide in specifying UML elements for aspects. However, the UML should also be able to accommodate other aspect-oriented languages, if needed.

6.1.2 Using Extensibility Mechanisms

A group of French researchers [Ho02] said existing UML mechanisms can be used to model separation of concerns, namely through the use of extensibility mechanisms.

Stereotypes can be used to label a class as an aspect or with the name of the aspect being modeled. Also, constraints and tag values can be used in a non-aspect class to denote join points in the class, where an aspect would be invoked.

Obviously, extensions will have to be used when using existing CASE tools to model aspects in the UML. The extensibility mechanisms are there to allow developers to go beyond standard designs and model new types of elements. However, the characteristics of aspects do not fit neatly into UML stereotypes. For example, some proposals may model an aspect as a stereotype of a UML class, but an aspect does not really have the characteristics of a class. So ultimately, extensions may not be able to properly model aspects.
Another paper stated that the use of extensibility mechanisms is only a "compromise solution" in the transition from pure object-oriented development to aspect-oriented development [Chavez02]. It proposes that AOP should have its own set of modeling elements. Again, it seems reasonable to assume that UML 2.0 should accommodate aspects.

6.1.3 Extensions for AspectJ

The French group was not the only group to propose the use of extensions. Researchers at the University of Essen, Germany [Stein02A], proposed a system of using existing extensibility mechanisms to model the concepts of AspectJ, namely join points, pointcuts and advice.

The join points, which are the points in a Java program where aspects are invoked, are modeled using links that can be added to a UML sequence diagram.

Pointcuts, which are specifications in the aspect that indicate where the join points are from the aspect’s view, are represented within the aspect by a stereotype. The aspects themselves are modeled as classes with a stereotype indicating that they are aspects. The aspect classes include "crosscut" stereotypes which indicate the Java classes that are cross-cut by the aspect. The crosscut stereotypes are contained in the operations section of the aspect classes.
The advice, which is the code of the aspect that is executed when a join point is reached, is also indicated by a stereotype within the aspect.

The German researchers also propose modeling the aspect weaver through use case models, which indicate the order in which java classes and aspects are reached.

Some of the other researchers would likely consider this proposal to be too specific to AspectJ. As another paper presented at the workshop [Clarke02] pointed out, “any design language that simply mimics the constructs of a particular AOP language is likely to fail to achieve implementation language independence.” As previously stated, since AspectJ, or any other aspect-oriented language, has not been adopted as a standard, the UML notation for aspects should be adaptable for use with other languages, even if it is closely modeled after AspectJ.

6.1.4 Making Aspects First-Class Citizens

Researchers at the Swiss Federal Institute of Technology Lausanne recognized a major problem of using existing UML elements [Kande02]; the parts of an aspects (such as advice and pointcuts in an AspectJ program) can be spread out among different diagrams in the model, making it difficult, if not impossible, to distinguish the aspects from normal Java classes.
They proposed treating the aspects as “first-class citizens” to clearly distinguish them, which is a sound idea. Unfortunately, their solution makes it somewhat difficult to distinguish aspects from classes. The aspects are modeled in a class diagram using stereotypes to indicate the aspect is not a Java class. But it looks like a standard Java class diagram.

Their solution also inserts the aspect as an object in a collaboration diagram, with a special “CP” notation to indicate connection points where the join points between the aspects and classes would be. Again, it is difficult to make the distinction between this and a standard collaboration.

Another paper [Jezequel02] raised the point that the aspect’s pointcuts (to use the AspectJ terminology) should have their own separate designation. If the pointcuts are included with the aspect, it may make it difficult to reuse the aspect.

The CEDRIC group in France [Pawlak02C] proposed a simple “pointcut” stereotype to associate an aspect with a class. Other notations along the association line link the aspect with a particular method in the class.
6.1.5 Aspects and Statecharts

Another UML modeling element that could be used for aspects is the statechart, as proposed by researchers at Lucent Technologies. [Aldawud02]. The statechart provides a view of an object over time as it goes through different states.

This proposal has some merit, as the state of the object in some cases will trigger the execution of an aspect. However, the focus of a statechart diagram is on the object, not the aspect, so it may not be the best model for the use of an aspect. As stated in [Fowler00], “state diagrams are not very good at describing behavior that involves a number of objects collaborating.”

6.2 Aspects and UML Proposals from the Germany Conference

The papers presented at the AOP conference in the Netherlands revealed a wide range of ideas but no consensus on how aspects should be modeled in the UML. The second major workshop on the issue was held on Sept. 30 at the UML conference in Germany. Some of the researchers who submitted papers to the Netherlands conference also submitted papers at the Sept. 30 workshop, but there were also some new proposals presented.
6.2.1 Aspects in Activity Diagrams

Two Portuguese researchers [Barros02] proposed using activity diagrams as the main modeling diagram for aspects. Their model was based on a new type of activity diagram, which is part of a proposal for new standards for UML 2.0. These activity diagrams use Petri net-based semantics, which allow activities to have “multiple token flows” at the same time. The aspects would be modeled as activities, with certain nodes along the activity path indicating where the aspects merge with the rest of the model.

The drawback to this proposal is that it puts the focus of the interaction between aspects and other concerns on the aspect. In other UML schemes, the concerns of the system as a whole are the focus of the modeling elements, with aspects added at points in diagrams that describe the system’s behavior. After all, the aspects are invoked by the system’s components, not the other way around.

6.2.2 Modeling Aspects at the Requirements Level

One determination of how aspects should be modeled is derived from where the aspects are created in the software development process. Developers will have to consider including aspects in use case diagrams if those aspects are found during the requirements gathering process.
One proposal [Araujo02] models aspects found at the requirements level as a special stereotype use case that is linked to a main use case using a “wrapped-by” stereotype, similar to the way a use case diagram is modeled with “include” or “extend” stereotypes to account for repeatable activities.

Unfortunately, the addition of the aspect stereotype to diagrams showing other system use cases suggests that the aspect is an extension, or somehow a part of the use case. The main goal of aspect-oriented programming is separation of concerns, so the aspect should be separate.

Another proposal in [Araujo02] models aspects in sequence diagrams by simply adding special identifiers at points in the sequence to reference an event. This is a sound proposal on how to model the interaction between aspects and other concerns, while still indicating their separation.

6.2.3 Creating New UML Icons for Aspects

Researchers at The American University in Cairo [Zakaria02] proposed a very simple class diagram to represent the relationship between aspects and other classes. They created a new icon to represent the aspect and another icon for what they referred to as a pointcut, although other researchers would refer to it as a join point (In much of the research, in which AspectJ terminology is not used, the terms “pointcuts” and
"join points" are used interchangeably to indicate the points where aspects cross-cut other concerns).

In the class diagram, the aspect and the class are both linked with associations to the pointcut. It is a very simple, yet effective representation.

6.2.4 Principles of Aspect-Oriented UML Modeling

Two papers from the Germany conference raised important points that form the basis for sound principles on how aspects should be modeled in the UML.

Researchers from Tel-Aviv University [Sapir02] emphasized the separation of the aspect from other modules of the application under development, or AUD. They state that:

"An aspect is considered to be a separate encapsulated and modular application that crosses the AUD. It does not merge into the AUD, nor changes its logic and static structure. The AUD is kept modular, untouched and consistent."

The key principle here is that the aspects are completely separated from system concerns, other than at the points where the aspects cross-cut them.

That leads to the second principle, which was raised by the University of Essen researchers who had originally presented their proposal on using extensibility mechanisms to model the concepts of AspectJ at the Netherlands conference. In a
follow-up paper presented at the Germany workshop [Stein02C], they emphasized the importance of identifying join points in an aspect-oriented system. They stated:

"It is a primary task for an aspect-oriented modeling language to provide suitable representations for join points."

These two principles, separation of concerns and identification of join points (or point cuts, according to the semantics of the development paradigm) should be the guiding principles of aspect-oriented modeling in the Unified Modeling Language.

6.3 Proposal for Modeling Aspects in the Unified Modeling Language

This proposal for standards for incorporating the modeling of aspects in the Unified Modeling Language is based on several areas: previous research on aspect-oriented programming, previous research on incorporating aspects into the UML, current UML specifications and the creation of aspects in the software development life cycle. The proposal assumes that as the Object Management Group considers revisions to UML specifications as it moves from version 1.4 to version 2.0 [OMG02], the updated version will be able to accommodate new elements that may be needed to properly model aspects.

We begin by re-introducing the terminology used in previous research on the structure of aspect-oriented programming, chiefly the pioneering work of Xerox's PARC group and the standards it set in its 1997 paper [Kiczales97].
PARC broke down elements of a system into two terms: components, which essentially are system modules, and aspects, which cross-cut the components repeatedly through the system. We will use that terminology in describing the proposal for incorporation of aspects into the UML.

The term “join points” is used repeatedly in discussions about AOP, although it can have different meanings in different contexts. For the purposes of this proposal, join points will refer to connection points where aspects cross-cut components.

The proposal will incorporate some of the ideas presented in previous research, as described earlier in this chapter. Most notably, we will use the two guiding principles outlined in Section 6.2.4:

- Each aspect will be considered as its own encapsulated module separated from the system as a whole, and also separated from other aspects;
- A key objective in modeling the system will be to detail the join points where aspects cross-cut the components.

The use of UML elements in this proposal will be based on procedures outlined in the Object Management Group’s specifications for the current version of UML, version 1.4 [OMG01], and on several UML manuals, including the reference manual published by the three UML pioneers, Rumbaugh, Jacobson and Booch [Rumbaugh99].
Finally, the lessons learned in Chapter 5 as we traced the creation of aspects through the software development process will be incorporated into this proposal. Some UML artifacts are closely tied to the stage in the process where the artifacts are created, so it is important to note the stages at which aspects were revealed. That influences the inclusion or exclusion of certain UML elements to the proposal for modeling aspects.

6.3.1 Aspect Packages in the UML

In order to accommodate the separation of aspects and components, we use UML packages. Each aspect is encapsulated within its own package, and all the functionality of the aspect can be modeled within the package. This should include class diagrams and interaction diagrams at a minimum, as well as anything else needed to specify the aspect’s functionality.

According to UML specifications [OMG01], the purpose of a package is to provide a grouping mechanism. It further states that “the package construct can be used for organizing elements for any purpose; the criteria to use for grouping elements together into one package are not defined within UML.”

In this case, the purpose of using packages is to separate and encapsulate the aspects.
As illustrated in Figure 6.1, all of the functionality of the system may be contained in one package, although the components can be distributed among several packages if that design is preferred.

Each aspect package is labeled with the stereotype <<aspect>> to distinguish them from component packages.
The package diagram also indicates that the aspects will cross-cut components in the main system at certain join points. Circles with a cross inside (to indicate their cross-cutting nature) indicate the join points. The definition of the join points is contained in brackets. The definitions in Figure 6.1 correspond to the AspectJ point cut definition, which is used in an aspect to describe where the aspect will cross-cut system components. For example, the definition "public static void *A" indicates a join point at every occurrence of a public static void method beginning with the letter "A."

The syntax can easily be adapted for a definition required by another aspect-oriented language.

The join point is, of course, a new modeling element. So this is an element that would be proposed for UML 2.0. The specifications of the join point element can be found in Appendix B. The format of the specifications correspond to the format currently used in Chapter 3 of UML 1.4.

An important modeling consideration here is that the join points are modeled outside both the aspects and the components. That way, these points of cross-cutting belong to neither concern individually but are an indication only of their cross-cutting relationship. The package diagram gives an overview of where to find the join points in the system.
The links from the aspects and components to the join points show a dependency relationship between the concerns and the join points. The cross-cutting features of the system are dependent upon both the implementation of the join points and the underlying AOP model.

During the software development process, the package diagram would begin to evolve during the inception phase, as we begin high-level domain analysis. The detail of the aspects within the packages would be modeled in later phases, and more aspect packages would be added, refined or expanded in the analysis workflow of later phases as more aspects are unveiled.

Recall from Chapter 5 that some aspects are identified in the Inception and Elaboration phases of the Unified Process before details about functionality are filled in. Using packages as the main unit to model aspects allows us to identify aspects early on and diagram their relationships with components, before having to consider the functional details of the aspects.

6.3.2 Aspects and Use Case Diagrams

The purpose of a use case diagram is to document the relationship between use cases and actors who interact with the system [OMG01]. As such, they represent an external view of the system [Fowler00].
Because of that, use case diagrams are not a useful tool for modeling aspects. The use of aspects is entirely an internal feature of the system, as they interact solely with internal system components. The actors, who represent users of the system, do not directly interact with the aspects.

As we examined the creation of aspects during the Unified Process in Chapter 5, we found that aspects were generally not found during the identification of use cases in the Inception and Elaboration phases. This is another reason why use case diagrams would not be useful to model aspects.

6.3.3 Class Diagrams for Aspects

Within the aspect package, we will use a class diagram to show which component classes are cross-cut by the aspect.

Figure 6.2 models a class diagram for a generic aspect. The relationships are drawn between a join point and the component classes, since the aspect’s relationship with the component occurs through the join point. The definition of the join point in Figure 6.2 shows that this aspect will cross-cut at a join point that occurs in methods beginning with “A,” so the component classes contain methods beginning with “A.”
With this diagram, we can see at a glance which components are cross-cut by the aspect.

The class diagrams in the main system package would not model the relationship between aspects and components, since there is no aspect class that can be modeled and no additional insight into the relationships between aspects and classes could be drawn from a component class diagram. An interaction diagram in the main system package will demonstrate those relationships.
6.3.4 Interaction Diagrams for Aspects

Interaction diagrams – that is, sequence and collaboration diagrams – are a key design element when modeling aspects in the UML. The diagrams will describe the specific relationships between aspects and components and the points at which they cross-cut. This would be modeled in interaction diagrams within the main system package.

The aspect package would also include interaction diagrams that display the self-contained sequence of events that take place when an aspect is invoked. Since the events of an aspect will not invoke a call to a component, the interaction diagrams encapsulated with an aspect package do not need to show relationships with the components. However, while researchers have not described such a scenario and none are included in our real estate agency example, a circumstance can be foreseen in which an aspect cross-cuts another aspect. In that case, the aspect interaction diagrams would need to show a relationship with elements from another aspect package.

We will look at several examples of sequence and collaboration diagrams, first to show how aspects could be modeled with existing UML elements and the drawbacks of those models. Secondly, we will demonstrate how new standards can be proposed for UML version 2.0 to model aspects and show their interactions with components.
Figure 6.3 shows a sequence diagram for the List Property use case (Appendix A) in the real estate agency example. In this sequence, four aspects cross cut components of the system:

- When a seller of real estate enters the system to list a property, the logging aspect is invoked to note that he or she has entered the system.
- After the listing is created and the seller adds his or her property to the overall property list stored in the database, the security aspect is invoked to ensure the seller is authorized to list properties in the system.
- Also at that point in the sequence, the agent warning aspect is invoked to check if the seller has selected one of the company’s agent to represent him or her.
- Finally, when the agent enters the system to request additional information from the user, the other logging aspect is invoked to note that the agent has entered the system.
Figure 6.3 models the aspects as notes that indicate the join points are reached when
the method in question is invoked. It is a simple representation of the interaction that
can be modeled with existing tools. However, the use of notes may not be adequate to
represent the activity that occurs at these join points.
The sequence diagram in Figure 6.4 attempts to remedy that problem by modeling the aspects as classes that interact with the main sequence. The aspect classes are invoked at join points when certain messages are called.
This approach also has an implementation problem. The CASE tools do not allow a connection between the aspect classes and a message in progress. That would be the ideal connection, because the aspect's relationship to the sequence is defined by a join point triggered in the message. But UML tools do not allow it.

The UML specifications [UML01] do not describe a situation in which an association from an element can interact with a message in progress. That suggests there are no rules against such a design, but it is obvious from the CASE tools' rejection of the connection that it is not necessarily supported by UML standards either. That problem is illustrated further in Figure 6.5. Collaboration diagrams represent a different view

Figure 6.5: Collaboration diagram with aspect objects
of relationships modeled in sequence diagrams, and CASE tools allow developers to convert a sequence diagram to a collaboration diagram with one mouse click. Figure 6.5 is the collaboration diagram derived from the sequence diagram in Figure 6.4. As the diagram shows, the aspect classes are completely separated from the rest of the model in the collaboration diagram. The relationships between the aspects and the messages are lost.

Figure 6.6 displays a model which attempts to implement the sequence using existing UML standards and tools. Because of the length of this diagram, only the sequence between the Seller actor and the Listing object, which had previously been linked by one createListing message, is displayed.

Figure 6.6: Sequence diagram with Before and After aspect advice
In this diagram, the createListing message is broken up and the advice, which is the functionality of an aspect in AspectJ, is inserted along the path between Seller and Listing. Since this is an AspectJ aspect, it includes Before advice and After advice.

This approach creates some problems. The diagram implies the message createListing is being communicated several times between objects, such as from the Before aspect object to the After aspect object and from the After object to Listing. In actuality, the message is only being sent from Seller to Listing.

Another problem will arise from implementation. This approach, if implemented, would insert the Before and After aspect code around every message that contains the proper join point. That creates the code tangling problem that aspect-oriented programming is intended to avoid.

Because of the unusual characteristics of aspects, it would appear that the best solution would be to propose new aspect elements that can be incorporated into UML 2.0, rather than using existing modeling elements.

The sequence diagram in Figure 6.7 represents a simple yet effective approach to the problem that was created by using drawing tools to supplement the available UML CASE tools. The boxes with the crossed lines on the bottom half (indicating cross-cutting) represent aspects, and the circles with the cross inside, as indicated
Figure 6.7: Sequence diagram with aspect and join point elements

previously, are join points. The join points reside on the line of the message where they occur in components in the main system, and they are connected to the aspects which cross-cut the components by a simple association.

In this diagram, the aspects and join points are clearly represented by unique symbols that distinguish them from other modeling elements. The distinct elements would allow for implementation of the system using various aspect-oriented languages.
As with the join points, the aspects would be a new element to UML that is proposed for UML 2.0. Specifications for the aspect element appear in Appendix C.

The interaction diagrams in the main system package would be perhaps the most important artifact that models the cross-cutting relationship between aspects and components. But aspect-oriented programming also creates the need for a second set of interaction diagrams that would be included in the aspect packages, modeling the behavior of the aspect by itself.

![Aspect sequence diagram]

Figure 6.8: Aspect sequence diagram
Figure 6.8 shows a sequence diagram for the agent warning aspect. There is nothing unusual about this diagram, other than the inclusion of a join point. In a typical sequence diagram, the action is initiated by an actor representing a user of the system. But when an aspect is invoked, the join point initiates the action, and at the end of the sequence, control is returned to the join point and the main system’s program continues on. So we insert the join point into the interaction diagram.

Many illustrations of AspectJ highlight the Before and After advice, in which the Before advice is basically the method that is invoked when the join point is reached in the system and the After advice is the method that is invoked once a system function has been executed. However, recall from Chapter 2 that there is no requirement that an AspectJ aspect have both a Before and After advice. In fact, there is also a third type of advice called Around which basically invokes the advice at the same time the join point is reached. So there is no special need to model Before and After functionality in the sequence diagram. And modeling the functionality as a straightforward sequence supports other aspect-oriented languages.

6.3.5 Aspects and Statechart Diagrams

According to the specifications of UML 1.4 [OMG01], state machines “can be used to model the behavior of individual entities (such as, class instances) or to define the interactions (such as, collaborations) between entities.” That suggests statechart
diagrams, which are used to model state machines, could be another element used to model the cross-cutting relationships between aspects and components.

However, authors of UML manuals suggest that state machines are not the best method of modeling interactions. According to [Arlow02], statechart diagrams “tend to be used for modeling the lifecycle history of a single reactive object.”

Further, it is advised in [Fowler00] that “state diagrams are not very good at describing behavior that involves a number of objects collaborating. As such, it is useful to combine state diagrams with other techniques.”

In other words, the best use of a statechart is to model a single object as it transitions through its various states. However, some developers may want to use a statechart to model an object’s interaction with aspects, so it is worthwhile to consider a UML specification for such a diagram.

Figure 6.10 displays a statechart of the real estate agency’s property list, which transitions to an updated state every time a new property is added to the list or a property is deleted. This transition would involve two cross-cutting aspects, the security aspect as the system determines if the user is authorized to add or delete a listing and the agent aspect as the system checks if the user has chosen an agent who works for the real estate agency.
The statechart diagram indicates the interaction with links to the two aspects at the transition points, with the join points indicated on the association lines.

Statecharts are generally created during the design workflow of the construction phase of the software development process, as more detail of the system is modeled.

6.3.6 Aspects and Other UML Diagrams

The only remaining UML diagram that should be considered for modeling aspects would be an activity diagram, which is used to analyze a sequence of events. But as noted in the UML 1.4 specifications [OMG01], “its primary focus is on the sequence
and conditions for the actions that are taken, rather than on which classifiers perform those actions.” Since in the consideration of aspects we are particularly interested in the elements (classifiers) that perform certain actions, this would indicate activity diagrams are not useful for modeling the interaction of an aspect cross-cutting a component.

Further, Fowler’s UML manual [Fowler00] specifically advises against using activity diagrams to model collaborations. “An interaction diagram is simpler and gives you a clearer picture of collaborations.”

Since aspects are considered mainly in the analysis and design workflows, in this thesis we are not considering component diagrams and deployment diagrams, which are created in the implementation workflow.

6.4 Chapter Summary

In this chapter, we have examined previous research into modeling aspects with the UML and proposed standards for how aspects should be modeled as the OMG considers a major revision of UML standards from current version 1.4 to version 2.0.

The major principles of aspect modeling in UML should be separation of aspects and components, with aspects encapsulated in their own UML packages. The join points at which aspects cross-cut components should be clearly identified in the models.
The major diagrams for specifying the interaction of aspects and components should be sequence and collaboration diagrams, although some developers may also find statechart diagrams useful. We can also use a class diagram in the aspect package to show the aspect's relationships with specific components.

We have proposed two new elements that can be proposed for inclusion in the revised UML specifications, version 2.0, representing join points and aspects. The specifications of those elements are found in Appendix B (join points) and Appendix C (aspects).
In this thesis we have examined aspect-oriented programming in the context of the software development process. We have looked at how aspects are revealed during the software development process, using both the Unified Process and Extreme Programming, and how those aspects can be modeled using the Unified Modeling Language.

While incorporating aspects into the software development process can be handled seamlessly, modeling those aspects in the UML, using existing standards and tools, is more complex and needs more refinement. We determined that the current version of the UML (version 1.4) is not sufficient to model the use of aspects and their cross-cutting relationships with components of the main system. So we have proposed the inclusion of two new modeling elements to incorporate aspects in the upcoming revision of the UML, version 2.0. Those two elements represent the concepts of join points, which are the points at which aspects cross-cut components in the main program, and the aspects themselves, for the purpose of modeling the interaction between aspects and components.
The proposed UML specifications for join points can be found in Appendix B and the specifications for aspects can be found in Appendix C. The specifications are proposed for inclusion in Chapter 3, "UML Notation Guide," of the UML specifications published by the Object Management Group [OMG01].

We have also proposed that modeling the functionality of aspects should be encapsulated into distinct UML packages, for the purposes of maintaining separation of concerns. Within the package, we can use class diagrams to model the relationship between aspects and components of the main system, and sequence or collaboration diagrams to model the functionality of an aspect. Interaction diagrams (sequence or collaboration diagrams) will also be the key diagrams used in the main system packages to model the relationship between components of the system and aspects.

It would be beneficial to continue the work in this thesis by implementing the real estate agency project in AspectJ (or another suitable aspect-oriented development environment) in order to further study implementation issues with aspects in the software development tools.

Beyond the research in this thesis, there is more work that will be necessary for the software development community. Once standards for modeling aspects are accepted by the Object Management Group, it will be necessary for CASE software development tools such as Rational Corp.’s Rose to incorporate the new elements into future releases, so developers can use the tools to model aspect-oriented systems. This
would entail writing translators that automatically generate AspectJ code, along with Java code, that is in synch with the modeling elements in the diagram.

At the present time, few major systems are under development using aspects. But as more software developers incorporate the principles of aspect-oriented programming into projects and aspect-oriented programming languages such as AspectJ are more fully refined, it will become more important to establish standards for developing aspects during the software development life cycle and for modeling them in the UML. The proposals in this thesis are based on principles of AspectJ but we argue that they can be extended other aspect-oriented programming environments. If another aspect-oriented paradigm is established as a standard, it may be necessary to refine the UML specifications for aspects to adhere to that paradigm. Therefore, it will be necessary to continue to monitor progress in aspect-oriented development to consider ongoing revisions to the UML to accommodate aspects.
REFERENCES

[Agile02]

[Aksit92]

[Aldawud02]

[Araujo02]

[Arlow02]

[AspectC02]

[AspectC++02]

[AspectJ02]

[Barros02]
[Beier02]

[Bergmans01]

[Bruce01]

[Chavez02]

[Clarke02]

[Cockburn02]

[Connections02]

[Demeter02]

[Eclipse02]
Eclipse.org, eclipse.org, 2002.

[Elrad01A]
[Elrad01B]

[Fowler00]

[Herrmann02]

[Highsmith00]

[Ho02]

[Hursch95]

[HyperJ01]

[Jacobson99]

[Jezequel02]

[Kande02]
[Katara02]

[Kiczales97]

[Kiczales01]

[Kulak00]

[Laddad02A]

[Laddad02B]

[Lieberherr00]

[Lieberherr01]

[Lions02]

[Miller01]

[Morales02]


[Sapir02]

[Shukla02]

[Smith02]

[Stein02A]

[Stein02B]

[Stein02C]

[Trese01]

[Tristram01]

[Viega01]

[Wake02]
[Walker 99]

[Zakaria02]
Appendix A

USE CASES AND SUPPORTING DOCUMENTS

The following pages contain the use cases, non-functional requirements and business rules for the real estate agency example used to demonstrate the process of finding aspects in software development.
<table>
<thead>
<tr>
<th>Use Case Name:</th>
<th>Agree To Terms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment:</td>
<td>Finished</td>
</tr>
<tr>
<td>Summary:</td>
<td>The buyer and the seller agree to the terms of the sale, including any required changes to the existing property, the items included with the property, the date of possession, the financing, and any other conditions of sale. The agents help their respective customers by offering advice, caution, or recommendations.</td>
</tr>
<tr>
<td>Preconditions:</td>
<td>An offer has been made and accepted.</td>
</tr>
<tr>
<td>Postconditions:</td>
<td>The buyer and the seller have agreed to terms.</td>
</tr>
</tbody>
</table>
| Basic Course of Events: | 1. The buyer and seller indicate that an agreement is possible.  
2. The system responds by notifying the buyer, the buyer’s agent, the seller, the seller’s agent, the legal analyst, and the financial analyst that the agreement process is ready to begin.  
3. The buyer submits a proposal of terms.  
4. The system responds by allowing all actors to view the proposal of terms and make their changes.  
5. The indicated actors make their changes.  
6. The system responds by making the actor’s changes public to all.  
7. The actors discuss the changes and come to an agreement on each proposed change, item by item.  
8. The system responds by consolidating the agreed-upon changes and making the proposal of terms public again.  
9. The actors indicate their agreement.  
10. The system indicates that the proposal of terms is final. |
| Alternative Paths: | 3a. The buyer’s agent may submit a proposal of terms to the buyer, who may then submit it as his or her own.  
3b. The seller’s agent may submit a proposal of terms to the seller, who may then submit it as his or her own.  
3c. The seller may be the one to submit a proposal of terms. |
| Exception Paths: | 9a. The buyer or seller do not agree to the proposal of terms at it stands.  
1. The objecting party enters an objection.  
2. The system notifies all actors of the objection.  
3. The objecting party modifies the proposal.  
4. Processing returns to Step 4 of Basic Course of Events. |
<p>| Extension Points: | None |
| Trigger: | Buyer and seller indicate that agreement to terms can begin. |
| Assumptions: | None |
| Related Business Rules: | None |</p>
<table>
<thead>
<tr>
<th>Use Case Name:</th>
<th>Authenticate User</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment:</td>
<td>Finished</td>
</tr>
<tr>
<td>Summary:</td>
<td>Both internal and external users must provide identification to the system so the system can grant them appropriate access to resources.</td>
</tr>
<tr>
<td>Preconditions:</td>
<td>System is operational.</td>
</tr>
<tr>
<td>Postconditions:</td>
<td>An authorized user is granted system access.</td>
</tr>
</tbody>
</table>
| Basic Course of Events: | 1. User enters a user id and password.  
2. System looks up this user id/password combination to determine if this is an authorized user.  
3. System acknowledges the authorized user and grants them access. |
| Alternative Paths: | None |
| Exception Paths: | 2a. The user id and password combination are not found in the authorized users list. System informs the user that the user id/password combination is incorrect. User retries login. If three incorrect combinations are provided the system locks any and all combinations tried.  
If the system is locked due to three incorrect login attempts then the system informs user to contact a system administrator to unlock the system. |
<p>| Extension Points: | None |
| Trigger:      | User needs system access. |
| Assumptions:  | None |
| Related Business Rules: | None |</p>
<table>
<thead>
<tr>
<th>Use Case Name:</th>
<th><strong>Close On Property</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment:</td>
<td><strong>Finished</strong></td>
</tr>
<tr>
<td>Summary:</td>
<td>Buyer and seller close deal on property transaction and consider advice from various analysts through the process.</td>
</tr>
<tr>
<td>Preconditions:</td>
<td>Buyer and seller have agreed to terms. The buyer's source of payment has been secured.</td>
</tr>
<tr>
<td>Postconditions:</td>
<td>Property is closed and transaction is completed.</td>
</tr>
</tbody>
</table>
| Basic Course of Events: | 1. Buyer and seller request to view the contract.  
2. System presents contract and requests buyer’s and seller’s confirmation.  
3. Buyer and seller confirm the sale.  
4. System records the confirmed sale and registers the property in the buyer's name. |
<p>| Alternative Paths: | None |
| Exception Paths: | 3a. The buyer and/or the seller do not confirm the sale. System informs the parties that the transaction did not occur and records the transaction as aborted during the closing stage. |
| Extension Points: | None |
| Trigger: | Buyer and seller agree to terms and are ready to close deal. |
| Assumptions: | None |
| Related Business Rules: | None |</p>
<table>
<thead>
<tr>
<th>Use Case Name:</th>
<th><strong>Inspect Property</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment:</td>
<td>Finished</td>
</tr>
<tr>
<td>Summary:</td>
<td>The Buyer and Buyer Agent require an Inspector to look at the property before the Close on Property. The Inspector examines defects in the property and suggests ways that the property must be fixed for the Close to take place.</td>
</tr>
<tr>
<td>Preconditions:</td>
<td>The Buyer has made an offer for the property and is in the process of the Agreement of Terms.</td>
</tr>
<tr>
<td>Postconditions:</td>
<td>The Buyer may adjust his/her side of the Agreement based on deficiencies found in the Property by the Inspector.</td>
</tr>
</tbody>
</table>
| Basic Course of Events: | 11. The Inspector inspects the home and fills out a report in the Real Estate System indicating the problems found with the Property.  
12. The Buyer's Agent assesses the report from the system and reports the findings to the Buyer.  
13. The Buyer Agent reworks the Agreement of Terms, factoring in deficiencies that must be fixed before the Closing can take place on the Property. |
| Alternative Paths: | 1a. The Inspector does not find anything wrong with the Property and the Agreement of Terms does not have to be reworked.  
3a. The Buyer may not have an Agent and therefore may not use Step 2, and will have to rework the Agreement of Terms by himself/herself. |
<p>| Exception Paths: | 2a. If too many deficiencies are found by the Inspector, there may be a legal way for the Buyer to back out of the Sale. If this occurs, the Inspect Property would destroy the Agreement of Terms. |
| Extension Points: | This is an extension point from the Agree to Terms Use Case. |
| Trigger:       | Agreement of Terms   |
| Assumptions:   | Seller has given approval to have Inspector look at the Property. |
| Related Business Rules: | None |</p>
<table>
<thead>
<tr>
<th>Use Case Name:</th>
<th>List Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment:</td>
<td>Finished</td>
</tr>
<tr>
<td>Summary:</td>
<td>The seller puts a property up for sale or rent, including a set of information that classifies the property uniquely for prospective buyers or renters.</td>
</tr>
<tr>
<td>Preconditions:</td>
<td>The seller has chosen an agent or chosen not to have an agent.</td>
</tr>
<tr>
<td>Postconditions:</td>
<td>The property is available for viewing on the system by prospective buyers.</td>
</tr>
</tbody>
</table>
| Basic Course of Events: | 14. This use case begins when the seller enters the information required to list a property.  
15. The system responds by saving the listing and notifying the seller's agent that the listing has been entered.  
16. The seller's agent checks the listing and solicits the seller for additional information or clarifications. Then the seller's agent completes the listing and confirms it.  
17. The system records the confirmation and lists the property publicly. |
| Alternative Paths: | 1a. The seller may actually be renting property as opposed to selling property. This scenario can be followed in the Rent Property Use Case.  
3a. If there are no clarifications or additions required, as judged by the seller's agent, the seller's agent simply confirms the listing. |
<p>| Exception Paths: | 1a. The seller who is trying to rent the property may not be licensed to have tenants. If he/she is not authorized, the rental property will not be available to list. |
| Extension Points: | None |
| Trigger: | The Seller has decided to sell or rent a property. |
| Assumptions: | None |
| Related Business Rules: | None |</p>
<table>
<thead>
<tr>
<th>Use Case Name:</th>
<th>Offer Price For Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment:</td>
<td>Finished</td>
</tr>
<tr>
<td>Summary:</td>
<td>Buyer makes offer to seller. Seller’s agent and buyer’s agent offer their respective customers advice.</td>
</tr>
<tr>
<td>Preconditions:</td>
<td>Seller has listed property. Buyer and seller have chosen agents, if they intend to use an agent.</td>
</tr>
<tr>
<td>Postconditions:</td>
<td>An agreement on the price is reached.</td>
</tr>
</tbody>
</table>

**Basic Course of Events:**
1. Buyer bids on the property in the system.
2. System saves bid and presents agent criteria to buyer.
3. Buyer chooses agent criteria and agent.
4. System registers choice of agent and notifies seller and seller’s agent of offer.
5. Seller receives and reviews offer.
6. System notifies buyer that seller has received the offer.
7. Seller submits counteroffer.
8. System notifies buyer and their agent of counteroffer.
10. System notifies the seller that the counteroffer was accepted.

**Alternative Paths:**
3a. If the buyer does not choose an agent the system warns of proceeding without an agent. If the buyer still does not choose to have an agent the system continues without an agent.
7a. If the seller accepts the buyer’s offer the seller informs the system of acceptance. The system notifies the buyer and their agent of the acceptance.
9a. If the buyer does not accept the seller’s counteroffer then the buyer submits a counteroffer. The system notifies the seller and their agent of the counteroffer. Process continues as before from seller’s receipt of offer.

**Exception Paths:** None

**Extension Points:** None

**Trigger:** Buyer decides to make offer on the property.

**Assumptions:** None

**Related Business Rules:**
- Rule ID 005 - Seller may pull property almost any time
- Rule ID 006 - Number of counteroffers limited
<table>
<thead>
<tr>
<th>Use Case Name:</th>
<th>Process Loan</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment:</td>
<td>Finished</td>
</tr>
<tr>
<td>Summary:</td>
<td>The buyer and the loan provider work out terms of a mortgage loan.</td>
</tr>
<tr>
<td>Preconditions:</td>
<td>The buyer has agreed to terms with the seller to purchase property.</td>
</tr>
<tr>
<td>Postconditions:</td>
<td>The loan is granted and recorded.</td>
</tr>
</tbody>
</table>
| Basic Course of Events: | 1. This use case begins when the buyer indicates a need to obtain a loan to purchase the property.  
2. The system responds by asking the buyer to provide certain personal financial data (income, savings, expected down payment).  
3. The buyer's agent enters the data as provided by the buyer.  
4. The system responds by contacting a loan provider with the details of the buyer's financial data and the purchase agreement.  
5. The loan provider sends a loan proposal to the system.  
6. The system provides the loan proposal to the buyer and the buyer's agent.  
7. The buyer makes a counterproposal to the loan provider.  
8. The system responds by sending the counterproposal to the loan provider.  
9. The loan provider accepts the counterproposal and informs the system.  
10. The system responds by notifying the buyer and the buyer's agent of the acceptance.  
11. The buyer agrees to the terms of the loan.  
12. The system records the buyer's acceptance of the loan and uses the previously input financial data to file the loan application with the loan provider. |

**Alternative Paths:**  
7a. If the buyer accepts the loan proposal, skip to step 10.  
9a. If the loan provider rejects the counterproposal and sends a new proposal to the system, go back to step 6.  

**Exception Paths:**  
5a. If the loan provider rejects the buyer or if the buyer is unhappy with the loan provider, go back to step 4 and contact a different loan provider.  
9b. If the loan provider rejects the buyer, go back to step 4 and contact a different loan provider.  

**Extension Points:** None  

**Trigger:** The buyer indicates a need to obtain financing.  

**Assumptions:** None  

**Related Business Rules:** Rule ID 009 - Buyer may use outside loan provider
Use Case Name: Rent Property

Increment: Finished

Summary: The renter decides to lease a property from the landlord according to the terms specified by the landlord.

Preconditions: The renter chooses a property from the list of properties available for lease.

Postconditions: The renter and landlord sign a lease.

Basic Course of Events:
1. This use case begins when the renter chooses a rental property and asks to lease it.
2. The system responds by producing a standard lease according to terms previously set by the landlord.
3. The renter agrees to the terms and signs the lease.
4. The system responds by informing the landlord that a renter has chosen the property and signed a lease.
5. The landlord informs the system that he or she agrees to the lease and signs it.
6. The system records the lease and stores it.

Alternative Paths: None

Exception Paths:
3a. If the renter rejects the terms, the use case ends.
5a. If the landlord rejects the renter as a tenant, the use case ends and the lease is not recorded.

Extension Points: None

Trigger: The renter chooses a property.

Assumptions: The landlord is not open to negotiate on the amount of rent.

Related Business Rules: None
<table>
<thead>
<tr>
<th>Use Case Name:</th>
<th>Review Listings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment:</td>
<td>Finished</td>
</tr>
<tr>
<td>Summary:</td>
<td>The buyer (or renter) browses through the listings of properties to purchase or rent with the help of the buyer’s agent.</td>
</tr>
<tr>
<td>Preconditions:</td>
<td>At least one property is listed.</td>
</tr>
<tr>
<td>Postconditions:</td>
<td>None</td>
</tr>
</tbody>
</table>
| Basic Course of Events: | 1. This use case begins when the buyer (or renter) asks to view the list of properties.  
                            2. The system responds by providing the buyer with criteria for searching (price range, location, home type), including the option at looking at sale or rental properties.  
                            3. The buyer’s agent enters the buyer’s preferences into the system.  
                            4. The system responds by providing a listing of specific properties that match the buyer’s wishes. |
<p>| Alternative Paths:  | None                             |
| Exception Paths:    | None                             |
| Extension Points:   | None                             |
| Trigger:            | The buyer asks to view properties |
| Assumptions:        | None                             |
| Related Business Rules: | None                            |</p>
<table>
<thead>
<tr>
<th>Use Case Name:</th>
<th>Select Agent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment:</td>
<td>Finished</td>
</tr>
</tbody>
</table>

**Summary:** The customer browses a listing of available agents, and based on location, expertise, previous experience, the user selects and agent.

**Preconditions:** The customer has decided to choose an agent.

**Postconditions:** The customer has chosen an agent and the system has recorded the selection.

**Basic Course of Events:**
18. The customer requests a list of agents.
19. The system responds with a listing of agents, with details on the agent’s location, expertise, and availability.
20. The user responds by selecting an agent.
21. The system records the user’s selection.

**Alternative Paths:**
2a. The user has questions for an agent and wishes to engage that agent in an online chat.
   1. The user selects an agent to chat with and specifies the reason.
   2. The system notifies the agent about the chat request.
      a. The agent is available
         i. The agent chats with the user.
      b. The agent is unavailable
         i. The user can leave a message for the agent.
3a. The user chooses to proceed without being represented by an agent.
   1. The system responds with a notice regarding the legal ramifications of proceeding without an agent and prompts the user for confirmation.
   2. The user confirms the request to proceed without an agent.
   3. The system records the fact that the user is proceeding without an agent.

**Exception Paths:** None

**Extension Points:** None

**Trigger:** The customer indicates that they would like to choose an agent.

**Assumptions:** None

**Related Business Rules:** Rule ID 002 - Customer can transact without an agent
<table>
<thead>
<tr>
<th>Use Case Name:</th>
<th>Sell Property</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increment:</td>
<td>Finished</td>
</tr>
<tr>
<td>Summary:</td>
<td>System Context Use Case. The seller lists the property, a buyer purchases the property, and the agent guides them through the process and offers advice, caution, and recommendations.</td>
</tr>
<tr>
<td>Preconditions:</td>
<td>N/A</td>
</tr>
<tr>
<td>Postconditions:</td>
<td>N/A</td>
</tr>
<tr>
<td>Basic Course of Events:</td>
<td></td>
</tr>
<tr>
<td>22. The seller selects an agent.</td>
<td></td>
</tr>
<tr>
<td>23. The system responds by assigning an agent and notifying the seller’s agent.</td>
<td></td>
</tr>
<tr>
<td>24. The seller lists the property to sell.</td>
<td></td>
</tr>
<tr>
<td>25. The system responds by displaying this property in the property listing and linking it for searches.</td>
<td></td>
</tr>
<tr>
<td>26. The buyer selects an agent.</td>
<td></td>
</tr>
<tr>
<td>27. The buyer reviews the property listings by entering search criteria.</td>
<td></td>
</tr>
<tr>
<td>28. The system responds by displaying properties that match the buyer’s search criteria.</td>
<td></td>
</tr>
<tr>
<td>29. The buyer finds a property and makes an offer on it.</td>
<td></td>
</tr>
<tr>
<td>30. The system responds by notifying the seller and the seller’s agent.</td>
<td></td>
</tr>
<tr>
<td>31. The seller responds to the offer with a counteroffer.</td>
<td></td>
</tr>
<tr>
<td>32. The system responds by notifying the buyer and the buyer’s agent.</td>
<td></td>
</tr>
<tr>
<td>33. The buyer and the seller agree to terms.</td>
<td></td>
</tr>
<tr>
<td>34. The system responds by recording the agreement.</td>
<td></td>
</tr>
<tr>
<td>35. The buyer indicates that a loan is required.</td>
<td></td>
</tr>
<tr>
<td>36. The system responds by locating an appropriate loan provider.</td>
<td></td>
</tr>
<tr>
<td>37. The buyer and the loan provider agree to loan terms.</td>
<td></td>
</tr>
<tr>
<td>38. The system responds by recording the terms of the loan.</td>
<td></td>
</tr>
<tr>
<td>39. The buyer and the seller close on the property.</td>
<td></td>
</tr>
<tr>
<td>40. The system responds by recording the details of the close.</td>
<td></td>
</tr>
<tr>
<td>Alternative Paths:</td>
<td>N/A</td>
</tr>
<tr>
<td>Exception Paths:</td>
<td>N/A</td>
</tr>
<tr>
<td>Extension Points:</td>
<td>N/A</td>
</tr>
<tr>
<td>Trigger:</td>
<td>N/A</td>
</tr>
<tr>
<td>Assumptions:</td>
<td>N/A</td>
</tr>
<tr>
<td>Related Business Rules:</td>
<td>N/A</td>
</tr>
</tbody>
</table>
### Non-functional Requirements

<table>
<thead>
<tr>
<th>Number</th>
<th>Category</th>
<th>Requirement</th>
<th>Applies to Use Cases</th>
<th>Exceptional Cases</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Availability</td>
<td>The system must be available to internal and external users 24 hours a day, 7 days a week, 99.9% of the time.</td>
<td>All</td>
<td>None</td>
</tr>
<tr>
<td>002</td>
<td>Cost of ownership</td>
<td>The system's daily total cost of ownership must not exceed $5,000.</td>
<td>All</td>
<td>None</td>
</tr>
<tr>
<td>003</td>
<td>Maintainability</td>
<td>The system must be maintainable by the IT staff, who currently have skills in Visual Basic, Active X, MS IIS, MS SQL Server, and MS Transaction Server.</td>
<td>All</td>
<td>None</td>
</tr>
<tr>
<td>004</td>
<td>Data Integrity</td>
<td>Information on property closures must be unalterable by anyone after the closing occurs.</td>
<td>All</td>
<td>None</td>
</tr>
<tr>
<td>005</td>
<td>Development cost</td>
<td>Cost of development must not exceed $2,500,000, including hardware, packages, and custom development</td>
<td>All</td>
<td>None</td>
</tr>
<tr>
<td>006</td>
<td>Delivery Date</td>
<td>The system must have at least core functionality in place by January 1, 2002.</td>
<td>All</td>
<td>TBD</td>
</tr>
<tr>
<td>007</td>
<td>Extensibility</td>
<td>The system should be built in a way such that the customer can involve other agencies on a pay-per-use basis.</td>
<td>All</td>
<td>None</td>
</tr>
<tr>
<td>008</td>
<td>Flexibility</td>
<td>The system should be able to handle interfaces with the following financial institutions: NationsBank, CitiBank, and National City Bank.</td>
<td>Process Loan</td>
<td>None</td>
</tr>
<tr>
<td>009</td>
<td>Installability</td>
<td>The system must be able to installed locally as well as remotely.</td>
<td>All</td>
<td>None</td>
</tr>
<tr>
<td>010</td>
<td>Leverageability, reuse</td>
<td>The system should be developed using the latest object-oriented technology.</td>
<td>All</td>
<td>None</td>
</tr>
<tr>
<td>011</td>
<td>Operability</td>
<td>Daily system operations must be easily handled by the current IT staff.</td>
<td>All</td>
<td>None</td>
</tr>
<tr>
<td>012</td>
<td>Performance</td>
<td>Online response time for agents should be within 10 seconds 90% of the time. Online response for external users should be within 7 seconds 95% of the time.</td>
<td>All</td>
<td>None</td>
</tr>
<tr>
<td>Number</td>
<td>Category</td>
<td>Requirement</td>
<td>Applies to Use Cases</td>
<td>Exceptional Cases</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>----------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>013</td>
<td>Portability</td>
<td>The system should be usable by customers or agents with any of the following internet browsers: Netscape Navigator 6.x or later, MS Internet Explorer 5.x or later and Opera 4.x or later.</td>
<td>All</td>
<td>None</td>
</tr>
<tr>
<td>014</td>
<td>Quality</td>
<td>The system must pass QA inspections before each increment is sent to the customer.</td>
<td>All</td>
<td>None</td>
</tr>
<tr>
<td>015</td>
<td>Fault tolerance, robustness</td>
<td>The system should be able to handle customers with poor internet connections, and recover from a server crash.</td>
<td>All</td>
<td>None</td>
</tr>
<tr>
<td>016</td>
<td>Scalability</td>
<td>The system should provide the specified response times with a load of 500 concurrent users.</td>
<td>All</td>
<td>None</td>
</tr>
<tr>
<td>ID</td>
<td>Name</td>
<td>Description</td>
<td>Category</td>
<td>Static/Dynamic</td>
</tr>
<tr>
<td>------</td>
<td>-------------------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>------------</td>
<td>----------------</td>
</tr>
<tr>
<td>001</td>
<td>Customer can change agents</td>
<td>Customers can change the agent they are using at any time during the transaction with no penalty.</td>
<td>Structural</td>
<td>Dynamic</td>
</tr>
<tr>
<td>002</td>
<td>Customer can transact without an agent</td>
<td>Customers can transact business as buyers or sellers without having an agent. However, each time a customer chooses to transact business without an agent, he or she must be warned of the downside of not having an agent.</td>
<td>Structural</td>
<td>Dynamic</td>
</tr>
<tr>
<td>003</td>
<td>Customers not using agents are charged less</td>
<td>When customers decide not to use an agent to transact business, they will be charged less than customers who use agents.</td>
<td>Structural</td>
<td>Dynamic</td>
</tr>
<tr>
<td>004</td>
<td>Agent fees depend on number of interactions</td>
<td>The fees the customer is charged are based on the number of interactions with an agent. Each interaction requiring an agent will be charged to the customer as a one-hour minimum. If an agent spends more than one hour, the customer will be charged in quarter-hour segments.</td>
<td>Computation</td>
<td>Dynamic</td>
</tr>
<tr>
<td>ID</td>
<td>Name</td>
<td>Description</td>
<td>Category</td>
<td>Static/Dynamic</td>
</tr>
<tr>
<td>------</td>
<td>-----------------------------------------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------</td>
<td>-------------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>005</td>
<td>Seller may pull property almost any time</td>
<td>The seller may retract the offer to sell the property at any time except after the seller has made a counteroffer to a prospective buyer. After this has occurred, if the seller and buyer are still in active negotiation, the seller must first notify all involved agents and the active buyer.</td>
<td>Action restricting</td>
<td>Dynamic</td>
</tr>
<tr>
<td>006</td>
<td>Number of counteroffers limited</td>
<td>The number of counteroffers between buyer and seller is limited to ten for the buyer and ten for the seller. After this maximum has been reached, one or both agents or an agency designee will step in and conduct a review of the transaction with the customers.</td>
<td>Action triggering</td>
<td>Dynamic</td>
</tr>
<tr>
<td>007</td>
<td>Definition of “active negotiation”</td>
<td>An active negotiation is a transaction that has occurred within the past five business days.</td>
<td>Structural fact</td>
<td>Dynamic</td>
</tr>
<tr>
<td>ID</td>
<td>Name</td>
<td>Description</td>
<td>Category</td>
<td>Static/Dynamic</td>
</tr>
<tr>
<td>-----</td>
<td>-------------------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------------------</td>
<td>----------------</td>
</tr>
<tr>
<td>008</td>
<td>No &quot;undo&quot; after property closure</td>
<td>After a property has closed, any protests from buyer or seller will go to an arbitration board, set up at the mutual agreement of our agency, the buyer and the seller. No protests after closure will be handled any other way.</td>
<td>Structural fact</td>
<td>Static</td>
</tr>
<tr>
<td>009</td>
<td>Buyer may use outside loan provider</td>
<td>The buyer may use a loan provider outside the system. This will not affect the fees charged to the customer, other than requiring less time with the agent.</td>
<td>Structural fact</td>
<td>Dynamic</td>
</tr>
</tbody>
</table>
Appendix B

UML SPECIFICATIONS OF JOIN POINTS

3.XX Join Point

3.XX.1 Semantics

A join point is a connection point between an aspect and a component of a system, where the aspect has a cross-cutting relationship with the component in an aspect-oriented programming model. The join point does not belong to any package but serves as a connection point between an aspect package and a package containing components of the main system. The name of a join point represents a unique description of where the point appears in a program. The semantics of the description depends on the target programming language used.

3.XX.2 Notation

A join point is drawn as a circle with a cross inside. The name of the join point is written in brackets below the circle.
3.XX.3 Presentation Options

In a package diagram, a join point is drawn between two packages, with dependency association lines drawn from the packages to the join point.

In a class diagram in an aspect package, a join point is drawn as a connection to a class.

In a sequence or collaboration diagram, a join point is drawn on the message line where the join point is found in the system. A join point can also be modeled in a sequence or collaboration diagram in an aspect package as an element which initiates a message.

In a statechart diagram, the join point is drawn on the association line that connects the object to an aspect.

3.XX.4 Style Guidelines

The name of the join point is written in a small font in brackets.
3.XX.5 Example

Figure 3.XX: Join points connecting aspects to a message in a sequence diagram (left) and connecting an aspect package to a component package (right).

3.XX.6 Mapping

A join point maps into a line of code in a component that represents the point at which an aspect cross-cuts the component.

In an aspect, a join point maps to the aspect's reference to the component's join point.
Appendix C

UML SPECIFICATIONS OF ASPECTS

3.XX Aspects

3.XX.1 Semantics

An aspect represents a separate concern that cross-cuts with component concerns in the main system. The functionality of the aspect is modeled in its own package and the name of the aspect package represents the name of the aspect.

3.XX.2 Notation

An aspect is modeled as a rectangle divided into a top and bottom compartment. The top compartment contains the name of the aspect. The bottom compartment is filled by an “X.”
3.XX.3 Presentation Options

An aspect is used in a sequence or collaboration diagram or a statechart diagram to display its interaction with a component of the main system. The aspect is connected by an association line to a join point that represents the cross-cutting point between the aspect and the component.

3.XX.4 Style Guidelines

The name of the aspect is written in a small font.

3.XX.5 Example

Figure 3.XX: Aspects connected to a message in a sequence diagram.
3.XX.6 Mapping

An aspect maps to an aspect package that is modeled in the overall system.
VITA

Mark Alan Basch has a Bachelor of Science in Journalism degree from Northwestern University, 1982, a Master of Business Administration degree from the University of North Florida, 1996, and expects to receive a Master of Science in Computer and Information Sciences from the University of North Florida in December 2002. Dr. Arturo J. Sanchez of the University of North Florida is serving as Mark’s thesis adviser.

Mark has an ongoing interest in software engineering and the software development process. His academic work includes using tools and procedures of the software development process, as well as programming experience in Java, C, C++, COBOL, Visual BASIC and SQL.

Mark has been a journalist for 20 years and is currently employed as a business writer for The Florida Times-Union in Jacksonville, Fla. Mark is originally from Somerset, N.J., but has resided in the Jacksonville area for the past 15 years.