The Object Primer: Introduction to Techniques for Agile Modeling

A Ronin International White Paper

Scott W. Ambler
President, Ronin International

Portions of this white paper have been modified from Scott W. Ambler’s book, 
*The Object Primer 2nd Edition*

This Version: June 22, 2001

Copyright 2001 Ronin International
Visit [http://www.ronin-intl.com](http://www.ronin-intl.com) for more development white papers
**Table Of Contents**

1. YOU HAVE MULTIPLE MODELS IN YOUR DEVELOPMENT TOOLKIT ........................................ 2

2. POTENTIAL TECHNIQUES FOR AGILE MODELING .............................................................. 3
   2.1 ACTIVITY DIAGRAM (UML) ............................................................................................. 3
   2.2 BUSINESS RULES ........................................................................................................... 4
   2.3 CHANGE CASES ................................................................................................................ 5
   2.4 CLASS MODEL (UML) ....................................................................................................... 6
   2.5 CLASS RESPONSIBILITY COLLABORATOR (CRC) CARDS ........................................... 10
   2.6 COLLABORATION DIAGRAM (UML) ............................................................................. 11
   2.7 COMPONENT DIAGRAM (UML) ....................................................................................... 13
   2.8 CONSTRAINTS .................................................................................................................. 15
   2.9 DATA/PERSISTENCE MODEL ......................................................................................... 16
   2.10 DEPLOYMENT DIAGRAM (UML) ................................................................................ 18
   2.11 ESSENTIAL USER INTERFACE PROTOTYPES .............................................................. 20
   2.12 NON-FUNCTIONAL REQUIREMENTS ........................................................................... 24
   2.13 SEQUENCE DIAGRAMS (UML) ................................................................................... 24
   2.14 STATE CHART DIAGRAM (UML) .................................................................................. 27
   2.15 USE CASE DIAGRAM (UML) ........................................................................................ 29
       2.15.1 Reuse in Use-Case Diagrams: <<extend>>, <<include>>, and Inheritance ............. 30
   2.16 USE CASES: ESSENTIAL AND SYSTEM ..................................................................... 31
   2.17 USER INTERFACE FLOW DIAGRAMS ......................................................................... 35
   2.18 USER INTERFACE PROTOTYPE .................................................................................. 36

3. SUMMARY .................................................................................................................................. 38

4. BLATANT ADVERTISING ........................................................................................................... 38

5. REFERENCES AND RECOMMENDED READING ............................................................... 39
   5.1 WEB-BASED RESOURCES ............................................................................................. 43

6. ABOUT THE AUTHOR ............................................................................................................ 44

7. INDEX ....................................................................................................................................... 45
The purpose of this white paper is to present a brief overview of possible modeling techniques\(^1\) for object-oriented and component-based development, techniques that you could apply as an agile modeler. Although you will not become an expert at any of these techniques by reading this paper, my hope is that you will at least gain an understanding of what techniques you have available to you and how they fit together.

There are several important messages that you want to take away from this paper:

1. **The Unified Modeling Language (UML) is not yet sufficient for the development of business software.** As you see in Figure 1-1, modified from *The Object Primer 2\(^{nd}\) Edition*, [http://www.ambysoft.com/theObjectPrimer.html](http://www.ambysoft.com/theObjectPrimer.html), you have a wide range of models available to you. Although this list is not complete, for example Robustness Diagrams or simple documentation tables are not indicated, you see that there are many non-UML artifacts indicated.

2. **You don’t need to apply all these modeling techniques on every project.** I'm a firm believer that you should have a wide range of techniques in your development toolkit so you can apply the right technique to the job at hand. A good analogy is that of a home-repair job. Some home repairs require me to use a hammer and screwdriver whereas others require a screwdriver, wrench, and soldering iron. Different projects, different tool needs, therefore to be successful at home repairs I would need a wide range of tools in my toolkit. Similarly, with software development projects sometimes I need to apply CRC cards, user stories, and physical data models (persistence models) whereas on other projects I need to apply user interface prototypes, UML class diagrams, and physical data models. Different projects, different modeling needs.

3. **You should take an agile approach to modeling.** I am currently working on a methodology called Agile Modeling (AM), the details for which are posted at [www.agilemodeling.com](http://www.agilemodeling.com). Simply put, AM is a collection of values, principles, and practices for modeling software that can be applied on a software development project in an effective and light-weight manner. AM recognizes that the values of agile development, as promoted by the Agile Alliance, are key to your being effective. These values are the promotion of 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. I invite you to get involved with the AM mailing list, visit [www.agilemodeling.com/feedback.htm](http://www.agilemodeling.com/feedback.htm) for details.

---

\(^1\) The techniques are described in detail in *The Object Primer 2/e* (Ambler, 2001).
Figure 1-1. Suggested models for developing business software.
1. You Have Multiple Models in Your Development Toolkit

AM’s principle multiple models tells you that you have many modeling artifacts at your disposal – change cases, user stories, business rules, UML activity diagrams, UML class diagrams, data models, and business rules – to name a few. Figure 1-1 shows that you have a wide range of modeling options open to you, a box represents an artifact that you may choose to create during a software project. Lines indicate major “iteration” relationships between the artifacts (more on this in a bit). An interesting observation is that you have far more than just the diagrams of the UML at your disposal, and when you consider the number of non-UML models depicted in Figure 1-1 you quickly realize that the UML very likely isn’t yet sufficient for business application development. By the way, Figure 1-1 isn’t complete itself – it doesn’t include robustness diagrams, XP’s user stories, or structure charts – so please don’t assume that these are the only models at your disposal.

Naturally, with a wide range of models at your disposal you will want to follow AM’s practice of apply the right artifact(s) to be successful. For example, you wouldn’t develop a use case model to describe your database schema, nor would you develop an data model to describe your user interface design.

Agile modelers will iterate to another artifact whenever they become stuck working on their current model. This immediately gets the modeler moving again, improving their short-term productivity. Hopefully the change of perspective will provide insight into whatever it was that got them stuck in the first place because it gives them an opportunity to go at the same problem from another direction. But what model to iterate to? That’s where the lines of Figure 1-1 come in, they provide a good indication what other artifact you should consider jumping to. For example, if you’re working on a component diagram and find you’re not making progress consider working on your deployment model to explore how you will distribute the software components across your hardware, working on a design class model to understand the inner workings of one or more components, or a collaboration diagram to explore how the components will interoperate.

An interesting result of this practice is that you often find you are more productive following the AM practice create several models simultaneously than you are by focusing on the creation of a single artifact. Because each type of model has its strengths and weaknesses no single model is sufficient for your modeling needs. For example, when you are exploring requirements you may need to develop some essential use cases, some essential UI prototypes, and some business rules. By working on these artifacts simultaneously you can quickly evolve your understanding of the requirements for your system, by working them one at a time you can only evolve your understanding of a single aspect of the requirements. The implication is that you should question the viability of modeling sessions that focus on a single artifact, such as a use-case model, a class model, or a data model. The RUP certainly doesn’t prevent such an approach, but the near-serial flow in the activity diagrams presented for each major modeling activity doesn’t communicate this concept well.

Agile Modeling’s principles of iterate to another artifact and create several models simultaneously can be tough ones for experienced modelers to adopt. Traditional modeling techniques often promoted a single artifact approach, such as use-case modeling or user-interface prototyping sessions, and often even modelers that focused, for example data modelers. These concepts were great in theory, focusing on a single artifact at a time should have allowed the modelers to get it right quickly, but unfortunately practice shows this not to be the case. A good way to ease into these practices is instead of use-case modeling sessions instead run requirements modeling sessions where you work on use cases, Class Responsibility Collaborator (CRC) cards, business rules, and user interface prototypes simultaneously. Similarly, hold analysis sessions where you are use case modeling, sequence diagramming, user interface prototyping, and class modeling may make sense, and design sessions where you are class modeling, state chart modeling, data modeling, component modeling, user interface prototyping, and hopefully even developing business code. Once you are comfortable with these practices the next step is to then merge your
modeling efforts in with your implementation efforts, applying multiple artifacts including all your potential models, source code, and test cases as needed – truly iterative development.

2. Potential Techniques for Agile Modeling

The following techniques are often used for object-oriented (OO) and/or component-based development (CBD). The techniques are presented in alphabetical order.

2.1 Activity Diagram (UML)

UML activity diagrams are used to document the logic of a single operation/method, a single use case, or the flow of logic of a business process. In many ways activity diagrams are the object-oriented equivalent of flow charts and data-flow diagrams (DFDs) from structured development. The activity diagram of Figure 2-1 depicts the business logic for the how someone new to the university would enroll for the first time.

Figure 2-1. A UML activity diagram for enrolling in school for the first time.

The filled circle represents the starting point of the activity diagram, effectively a placeholder and the filled circle with a border represents the ending point. The rounded rectangles represent processes or activities that are performed. For the diagram of Figure 2-1 the activities map reasonably closely to use cases, although you will notice that the "Enroll in Seminar(s)" activity would be the invocation of the
"Enroll in Seminar" use case several times. Activities can also be much finer grained, particularly if I had chosen to document the logic of a method instead of a high-level business process. The diamonds represent decision points, although in this example the decision point had only two possible outcomes it could just as easily had many more. The arrows represent transitions between activities, modeling the flow order between the various activities. The text on the arrows represent conditions that must be fulfilled to proceed along the transition and are always described using the format "[condition]". The thick bars represent the start and end of potentially parallel processes – after you are successfully enrolled in the university you must attend the mandatory overview presentation as well as enroll in at least one seminar and pay at least some of your tuition.

**Agile Modeling Practice – Apply the Right Artifact(s)**

Each artifact has its own specific applications. For example, a UML activity diagram is useful for describing a business process, whereas the static structure of your database is better represented by a physical data or persistence model. Very often a diagram is a better choice than source code – if a picture is worth a thousand words then a model is often worth 1024 lines of code when applied in the right circumstances (a term borrowed from Karl Wieger’s *Software Requirements*) because you can often explore design alternatives more effectively by drawing a couple diagrams on whiteboards with your peers than you can by sitting down and developing code samples. The implication is that you need to know the strengths and weaknesses of each type of artifact, something that is very difficult considering you have multiple models available to you.

### 2.2 Business Rules

A business rule is effectively an operating principle or policy that your software must satisfy. Business rules often focus on access control issues, for example professors are allowed to input and modify the marks of the students taking the seminars that they instruct but not the marks of students in other seminars. Business rules may also pertain to business calculations, for example how to convert a percentage mark (e.g. 76%) that a student receives into a seminar into a letter grade (e.g. A-). Some business rules focus on the policies of your organization, perhaps the university expels for a year anyone who fails more than two courses in the same semester. Figure 2-2 summarizes several examples of business rules and Figure 2-3 presents a fully documented version of BR123.

BR123 Tenured professors may administer student grades.

BR124 Teaching assistants that have been granted authority by a tenured professor may administer student grades.

BR177 Table to convert between numeric grades and letter grades.

BR245 All Master degree programs must include the development of a thesis.

Figure 2-2. Example business rules (summarized).

**Agile Modeling Principle – Know Your Models**

Because you have multiple models that you can apply you need to know their strengths and weaknesses to be effective in their use.
Name: Tenured professors may administer student grades
Identifier: BR123
Description: Only tenured professors are granted the ability to initially input, modify, and delete grades that students receive in the seminars that they and they only instruct. They may do so only during the period that a seminar is active.
Example: Dr. Bruce Banner, instructor of "Biology 301 Advanced Uses of Gamma Radiation," may administer the marks of all students enrolled in that seminar but not those enrolled in "Biology 302 Effects of Radiation on Arachnids" which is taught by Dr. Peter Parker.
Source: University Policies and Procedures
Doc ID: U1701
Publication date: August 14, 2000
Related rules: BR12 Qualifying For Tenure
BR65 Active Period for Seminars
BR200 Modifying Final Student Grades
Revision History: Defined March 2, 2001 by Diana Prince
Updated October 10, 2001 by Gwen Stacy to reference related rule BR200.

Figure 2-3. A fully documented business rule.

Agile Modeling Practice – Collective Ownership

Everyone can work on any model, and in fact any artifact on the project, if they need to.

2.3 Change Cases

Change cases are used to describe new potential requirements for a system or modifications to existing requirements. Change cases are similar in concept to use cases: where use cases describe behavioral requirements for your system, change cases describe a potential requirement that your system may need to support in the future. It is important to note that change cases can be defined pertaining to both behavioral and non-behavioral requirements.

Change cases can be developed as part of your overall modeling efforts, particularly during requirements gathering but also during analysis and design. Change cases are often the result of brainstorming by your SMEs, where questions such as “How can the business change?” “What technology can change?” “What legislation can change?” “What is your competition doing?” “What systems will we need to interact with?” And “Who else might use the system and how?” are explored. In many ways the change cases for a system is a well-documented risk assessment for changing requirements.

Change cases are documented in a simple manner. You describe the potential change to your existing requirements, indicate the likeliness of that change occurring, and indicate the potential impact of that change. Figure 2-4 presents two change cases, one potential change that is motivated by technical innovation, in this case the use of the Internet, and a second by a change in your business environment. Notice how both change cases are short and to the point, making them easy to understand. The name of a change case should describe the potential change itself, as you can see in the figure.

Copyright 2001 Ronin International
Visit http://www.ronin-intl.com for more development white papers
**Change case:** Registration will occur completely via the Internet.

**Likelihood:** Medium likelihood within 2 to 3 years, very likely within 10 years

**Impact:** Unknown. Although registration will be available online starting in September, we currently expect less than one quarter of registrations to be made via the Internet this year. Response time will be an issue during the peak usage periods, which are the two weeks prior to the beginning of classes each term as well as the first week of classes.

**Change case:** The University will open a new campus.

**Likelihood:** Certain. It has been announced that a new campus will be opened in two years across town.

**Impact:** Large. Students will be able to register in classes at either campus. Some instructors will teach at both campuses. Some departments, such as the Computer Science and Philosophy departments are slated to move their entire programs to the new campus. It is likely that most students will want to schedule courses at only one of the two campuses so we will need to make this easy to support.

**Figure 2-4. Examples of change cases for the university.**

An important thing to understand about change cases is that just because you have identified them that doesn’t mean you need to overbuild your system to support the potential changes. I like to consider the potential changes when making architecture and design decisions – when all things are equal I will pick a design alternative that fulfills one or more potential changes over one that does not (note the use of the word “equal”).

---

**Agile Modeling Practice – Model in Small Increments**

Incremental development in which you organize a larger effort into smaller portions that you release over time, hopefully in increments of several weeks or a month or two, increases your agility by enabling you to deliver software into the hands of your users faster.

---

### 2.4 Class Model (UML)

Class models are the mainstay of object-oriented analysis and design, and before the UML most methodologies called them object models instead of class models. Class models show the classes of the system, their inter-relationships (including inheritance, aggregation, and association), and the operations and attributes of the classes. Figure 2-5 and Figure 2-6 depict simple versions of class models, the first one of which I would consider an analysis class model due to the lack of technical details such as visibility and the second one a design class model. Figure 2-7 provides examples of a wider range of the notation that you may use in a UML class model.
**Agile Modeling Principle – Content is More Important Than Representation**

Any given model could have several ways to represent it. For example, a UI specification could be created using Post-It notes on a large sheet of paper (an essential or low-fidelity prototype), as a sketch on paper or a whiteboard, as a "traditional" prototype built using a prototyping tool or programming language, or as a formal document including both a visual representation as well as a textual description of the UI. An interesting implication is that a model does not need to be a document. Even a complex set of diagrams created using a CASE tool may not become part of a document, instead they are used as inputs into other artifacts, very likely source code, but never formalized as official documentation. The point is that you take advantage of the benefits of modeling without incurring the costs of creating and maintaining documentation.
Many developers worry about whether their artifacts -- such as models, source code, or documents -- are detailed enough or if they are too detailed, or similarly if they are sufficiently accurate. What they’re not doing is stepping back and asking why they’re creating the artifact in the first place and who they are creating it for. With respect to modeling, perhaps you need to understand an aspect of your software better, perhaps you need to communicate your approach to senior management to justify your project, or perhaps you need to create documentation that describes your system to the people who will be operating and/or maintaining/evolving it over time. If you cannot identify why and for whom you are creating a model then why are you bothering to work on it all? Your first step is to identify a valid purpose for creating a model and the audience for that model, then based on that purpose and audience develop it to the point where it is both sufficiently accurate and sufficiently detailed. Once a model has fulfilled its goals you’re finished with it for now and should move on to something else, such as writing some code to show that the model works. This principle also applies to a change to an existing model: if you are making a change, perhaps applying a known pattern, then you should have a valid reason to make that change (perhaps to support a new requirement or to refactor your work to something cleaner). An important implication of this principle is that you need to know your audience, even when that audience is yourself. For example, if you are creating a model for maintenance developers, what do they really need? Do they need a 500 page comprehensive document or would a 10 page overview of how everything works be sufficient? Don’t know? Go talk to them and find out.

---

**Student**

- name: string
- phoneNumber: PhoneNumber
- emailAddress: EmailAddress
- studentNumber: StudentNumber
- averageMark: long

+ isEligible(name: string, studentNumber: StudentNumber): boolean
+ Student(studentNumber: StudentNumber): Student <<constructor>>
+ getSeminarsTaken(): Vector
+ purchaseParkingPass()
+ getAverageMark(): long
- setAverageMark(newAverageMark: long)

**StudentNumber**

- number: int
- nextStudentNumber: int

+ StudentNumber(): StudentNumber <<constructor>>

---

**Agile Modeling Principle – Model With A Purpose**

Figure 2-6. A design class model.
Agile Modeling Practice – Create Several Models in Parallel

Because each type of model has its strengths and weaknesses no single model is sufficient for your modeling needs. For example when you are exploring requirements you may need to develop some essential use cases or user stories, an essential UI prototype, and some business rules. In combination with the practice of iterating to another artifact agile modelers will often discover that they are far more productive working on several models simultaneously than if they are only focusing on one at any given time.

Figure 2-7. More class modeling notation.
2.5 Class Responsibility Collaborator (CRC) Cards

A Class Responsibility Collaborator (CRC) model is a collection of standard index cards that have been divided into three sections, as depicted in Figure 2-8. A class represents a collection of similar objects, a responsibility is something that a class knows or does, and a collaborator is another class that a class interacts with to fulfill its responsibilities.

<table>
<thead>
<tr>
<th>Class Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Responsibilities</td>
</tr>
</tbody>
</table>

Figure 2-8. The layout of a CRC card.

A class represents a collection of similar objects. An object is a person, place, thing, event, or concept that is relevant to the system at hand. For example, in a university system there would classes that represent students, tenured professors, and seminars. The name of the class appears across the top of a CRC card and is typically a singular noun or singular noun phrase, such as “Student,” “Tenured Professor,” and “Seminar.” A responsibility is anything that a class knows or does. For example, students have names, addresses, and phone numbers. These are the things that a student knows. Students also enroll in seminars, drop seminars, and request transcripts. These are the things that a student does. Sometimes a class will have a responsibility to fulfill, but will not have enough information to do it. For example, you see in Figure 2-9 that students enroll in seminars. To do this, a student needs to know if there is a spot available in the seminar, and if so he then needs to be added to the seminar. However, students only have information about themselves (their name, . . .), and not about seminars. What the student needs to do is collaborate/interact with the card labeled “Seminar” in order to sign up for a seminar. Therefore, “Seminar” is included in the list of collaborators of “Student.”

<table>
<thead>
<tr>
<th>Student</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student number</td>
</tr>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Address</td>
</tr>
<tr>
<td>Phone number</td>
</tr>
<tr>
<td>Enroll in a seminar</td>
</tr>
<tr>
<td>Drop a seminar</td>
</tr>
<tr>
<td>Request transcripts</td>
</tr>
</tbody>
</table>

Figure 2-9. An example CRC card.
Although CRC cards were originally introduced as a technique for teaching object-oriented concepts they have also been successfully used by both developers and users to understand an OO application throughout its entire system development lifecycle. It is my experience that CRC models are an incredibly effective tool for domain modeling, the modeling of fundamental concepts within your domain, so that is the context in which I will discuss them. It is possible to use CRC models for design purposes, for example they are a one of the fundamental techniques employed for design in the Extreme Programming (XP) software process (Beck, 2000). However, my experience is that CRC models are very effective tools for working with your users and subject matter experts (SMEs) while you are gathering requirements, whereas UML class diagrams are a better choice for domain modeling during analysis.

**Agile Modeling Practice – Active Stakeholder Participation**

An expansion of XP's On-Site Customer which describes the need to have on-site access to users that have the authority and ability to provide information pertaining to the system being built and to make pertinent and timely decisions regarding the requirements, and prioritization thereof. AM expands XP's On-Site Customer practice to have project stakeholders -- including direct users, their management, senior management, operations staff, and support (help desk) staff -- actively involved in the project. This includes making timely resourcing decisions by senior management, public and private support for the project by senior management, active participation of operations and support staff in the development of requirements and models pertaining to their respective areas.

**2.6 Collaboration Diagram (UML)**

What happens when you need to show the behavior of several objects collaborating together to fulfill a common purpose? This is what UML collaboration diagrams can be used for, to provide a birds-eye view of a collection of collaborating objects.

Collaboration diagrams show the message flow between objects in an OO application, and also imply the basic associations (relationships) between classes. Figure 2-10 presents a simplified collaboration diagram for displaying a seminar details screen or page. The rectangles represent the various objects involved that make up the application, and the lines between the classes represent the relationships (associations, aggregation, composition, dependencies, or inheritance) between them. The same notation for classes and objects used on UML sequence diagrams (Section 2.13) are used on UML collaboration diagrams, another example of the consistency of the UML. The details of your associations, such as their multiplicities, are not modeled as this information is contained on your UML class diagrams: remember, each UML diagram has its own specific purpose and no single diagram is sufficient on its own. Messages are depicted as a labeled arrow that indicates the direction of the message, using a notation similar to that used on sequence diagrams. You may optionally indicate the sequence number in which the message is sent, indicate an optional return value, and indicate the method name and the parameters (if any) passed to it.
Figure 2-10. A collaboration diagram for a simple university.

**Agile Modeling Practice** – Discard Temporary Models

The vast majority of the models that you create are temporary/working models -- design sketches, low fidelity prototypes, index cards, potential architecture/design alternatives, and so on -- models that have fulfilled their purpose but no longer add value now that they have done so. Models quickly become out of sync with the code, and there is nothing wrong with that. You must then make the decision to synchronize the models if doing so adds value to your project or to simply discard them because the investment to update the models won't be recouped by the value of having done so (there's negative payback).
2.7 Component Diagram (UML)

Component-based development (CBD) and object-oriented development go hand-in-hand, and it is generally recognized that object technology is the preferred foundation from which to build components. The Unified Modeling Language (UML) includes a component diagram that can be used to both analyze and design your component-based software. Figure 2-11 presents an example component model for the university system. Components are modeled as rectangles with two smaller rectangles jutting out from the left-hand side. Components implement one or more interfaces, modeled using the same “lollipop” notation that UML class diagrams use. Components have dependencies on the interfaces of other components, modeled using the standard UML dependency notation.

The goal of component modeling is to distribute the classes of your system into larger-scale, cohesive components. In The Object Primer 2/e I describe the five steps that are typically performed in an iterative manner to componentize your object design:

1. Handle non-business/domain classes
2. Define class contracts
3. Simplify inheritance and aggregation hierarchies
4. Identify domain components
5. Define domain-component contracts

Blatant Advertising – Agile Modeling (AM) Workshop

How do you successfully model the complexities of modern-day software without getting bogged-down in mountains of paper work? How do you effectively engineer the requirements for your system? What techniques can you apply to analyze those requirements? To the design your software? This workshop is a straightforward, easy to understand introduction to object-oriented (OO), component-based, and essential modeling techniques for developing requirements, analysis, and design models. It includes the industry-standard techniques of the Unified Modeling Language (UML) but goes beyond them to be sufficient for the real-world development of modern business applications. While objects and components are often used to develop complex systems, learning how to work with object-oriented techniques does not need to be complicated, nor do you need to develop complex documentation to be successful using them. Visit http://www.ronin-intl.com/services/agileModeling.html for details.
Figure 2-11. An example UML component diagram for the university.
Agile Modeling Practice – Display Models Publicly

You should display your models publicly, often on something called a "modeling wall" or a "wall of wonder." This supports Open and Honest Communication on your team because all of the current models are quickly accessible to them, as well as with your project stakeholders because you aren’t hiding anything from them. Your modeling wall is where you post your models for everyone to see; the modeling wall should be accessible to your development team and other project stakeholders. Your modeling wall may be physical, perhaps a designated whiteboard for your architecture diagram(s) or a place where you tape a printout of your physical data model. Modeling walls can be virtual, such as an internal Web page that is updated with scanned images.

2.8 Constraints

A constraint is a restriction on the degree of freedom you have in providing a solution. Constraints are effectively global requirements, such as limited development resources or a decision by senior management that restricts the way that you develop a system. Constraints can be economic, political, technical, or environmental and pertain to your project resources, schedule, target environment, or to the system itself. Figure 2-12 presents several potential constraints for the university system. Like business rules and non-functional requirements, constraints are documented in a similar manner.

C23 The user interface will work on browsers that support HTML v2.0 or better.
C24 The system will work on Sun Solaris servers.
C52 The system will be written using J2EE technologies.
C56 The system will use the data contained the existing Oracle database.
C73 The system will be delivered before June 1, 2002.
C76 The system will be developed by unionized software professionals.

Figure 2-12. Potential constraints for the university system.

Agile Modeling Practice – Iterate To Another Artifact

When you are working on a development artifact -- such as a use case, CRC card, sequence diagram, or even source code -- and find that you are stuck then you should consider working on another artifact for the time being. Each artifact has its strengths and weaknesses, each artifact is good for a certain type of job. Whenever you find you are having difficulties working on one artifact, perhaps you are working on a use case and find that you are struggling to describe the business logic, then that’s a sign that you should iterate to another artifact. An interesting implication of Figure 1-1 is that the links from the artifact that you are currently working on provide a good indication of candidate artifacts that you may want to jump to. For example, if you are working on an essential use case then you may want to consider changing focus to start working on an essential UI prototype, a CRC model, a business rule, a system use case, or a change case. By iterating to another artifact you immediately become "unstuck" because you are making progress working on that other artifact. Furthermore, by changing your point of view you often discover that you address whatever it was that was causing you to be stuck in the first place.
2.9 **Data/Persistence Model**

If you are one of the lucky few to work in an organization using an object database (ODB) you do not need to read this section. However, if you are among the majority of developers and are using a relational database (RDB) then you need to pay careful attention to this section. Relational databases are often used as the mechanism to make your objects persistent. Because relational databases do not completely support object-oriented concepts the design of your database is often different than the design of your class diagram. Persistence models, also called data models or entity-relationship (ER) models, are used to communicate the design of a database, usually a relational database, to both your users and to other developers. At the time of this writing, the UML does not support persistence models.

In Figure 2-13 you see an example of a persistence model for the design of a simple human resources system. In the model there are four data entities – “Position,” “Employee,” “Task,” and “Benefit” – which in many ways are simply classes that have data but no functionality. The entities are connected by relationships, and although not shown in Figure 2-13 it is possible to model associations, inheritance, aggregation, and composition on persistence models just as you would in a class model (Section 2.4).

Persistence models are used to design the schema of your database. You typically need to draw a persistence model whenever you are using a relational database to store your objects in. The strength of persistence models is that data entities are conceptually the same as the tables of a relational database and that attributes are the same as table columns.

### Agile Modeling Principle – Multiple Models

You need to use multiple models to develop software because each model describes a single aspect of your software. Figure 1-1 depicts the latest incarnation of the solution to the detailed modeling process pattern, a pattern which addresses the question “What models are potentially required to build modern-day business applications?” These models are in addition to XP's user stories, an artifact used to define user requirements for a system. An important point is that you don't need to develop all of these models for any given system, but that depending on the exact nature of the software you are developing you will require at least a subset of the models. Different systems, different subsets. Just like every fixit job at home doesn't require you to use every tool available to you in your toolbox, overtime the variety of jobs you perform will require you to use each tool at some point. Just like you use some tools more than others, you will use some types of models more than others.
Figure 2-13. A persistence model for a simple human resources database.
Agile Modeling Practice – Formalize Contract Models

Contract models are often required when an external group controls an information resource that your system requires, such as a database, legacy application or information service. A contract model is something that both parties should mutually agree to and mutually change over time if required. Examples of contract models include the detailed documentation of an application programming interface (API), a file layout description, an XML DTD or a physical data model describing a shared database. As with a legal contract, a contract model often requires you to invest significant resources to develop and maintain the contract to ensure that it’s accurate and sufficiently detailed. Your goal is to minimize the number of contract models for your system to conform to the XP principle of traveling light. Note that you will almost always use an electronic tool to develop a contract model because the model must be maintained over time.

2.10 Deployment Diagram (UML)

A UML deployment diagram depicts a static view of the run-time configuration of processing nodes and the components that run on those nodes. In other words, deployment diagrams show the hardware for your system, the software that is installed on that hardware, and the middleware used to connect the disparate machines to one another. You want to create a deployment model for applications that are deployed to several machines. A point-of-sales application running on a thin-client network computer directly accessing a centralized database server would be a good candidate for a deployment model. So would a customer service system deployed using a distributed object architecture such as CORBA (Common Object Request Broker Architecture). Deployment models are also needed for the design of embedded systems, showing how the hardware and software components work together. In short, all but the most trivial of systems will require a deployment model.

Figure 2-14 presents an example of a UML deployment diagram for the student administration application. The three-dimensional boxes represent nodes such as computers or switches and connections between nodes are represented with simple lines. As you would expect software components, interfaces, and dependencies are indicated using the standard UML notations. In Figure 2-14 stereotypes indicate that the connection between the browser and the application server uses the Internet’s standard HTTP protocol and that Java’s Remote Method Invocation (RMI) protocol is used across the connection between the application server and the data server. The components have the same type of stereotypes as they did on the UML component diagram of Figure 2-11.
Figure 2-14. A deployment diagram for the university information system.
Agile Modeling Practice – Model With Others

When you Model With a Purpose you often find that you are modeling to understand something, that you are modeling to communicate your ideas to others, or you are seeking to develop a common vision on your project. This is a group activity, one in which you want the input of several people working together effectively. You will often find that your development team needs to work together to create the core set of models critical to your project. For example, to develop the metaphor or architecture for your system, you will often need to model with a group of people to develop a solution everyone agrees on as well as one that is as simple as possible. Most of the time the best way to do this is to talk the issue through with one or more people.

2.11 Essential User Interface Prototypes

The user interface (UI) is the portion of software that a user directly interacts with. An essential user interface prototype is a low-fidelity model, or prototype, of the UI for your system – it represents the general ideas behind the UI but not the exact details. Essential UI prototypes represent user interface requirements in a technology independent manner, just as essential use case models do for behavioral requirements. An essential user interface prototype is effectively the initial state, the beginning point, of the user interface prototype for your system. It models user interface requirements, requirements which are evolved through analysis and design to result in the final user interface design for your system.

There are two basic differences between essential user interface prototyping and traditional UI prototyping. First, your goal is to focus on your users and their usage of the system, not system features. This is one of the reasons why you want to perform essential use case modeling and essential user interface prototyping in tandem: they each focus on usage. Second, your prototyping tools are very simple, including white boards, flip chart paper, and sticky notes. The minute that you introduce electronic technology to your prototyping efforts you have made a design decision about the implementation technology. If you use an HTML development tool to build a user interface prototype then you have immediately narrowed your design space to the functionality supported within browsers. If you choose a Java development environment then you have narrowed your design space to Java, and if you choose a Windows-based prototyping tool, you narrow your design space to whatever is supported on the Windows platform. Understand the problem first, then solve it.

Let's work through an example. Figure 2-15 depicts an essential user interface prototype for enrolling in a seminar. The outside box represents the flip chart paper; you typically use one piece of flip-chart paper per major user interface element. The name of the major UI element, in this case “Enroll Student in Seminar,” is typically written in one of the corners of the flip chart paper. Notice how there are three containers, the three largest rectangles, none of which are marked as containers. Also notice how some minor UI elements are input fields, such as “Student Number” and “Student Name,” whereas others are display only. “Student Name” is interesting because it is a bit of a cheat, listing four separate data elements on the one sticky note. I will often do this when I know that some thing always come in a group and when I think I will need the room, which as you can see in Figure 2-15 I do. The “Professor” and “Enrollment Requester” hang off the edge of their container, something I did on purpose to show you that you do not have to get your essential user interface prototypes perfect. There are several “Requester” UI elements, for example “Help Requester” and “Search Requester,” indicating user interface elements that are often implemented as push buttons, function keys, or “hot key” combinations such as CTRL-SHIFT-S. Many of the minor UI elements include detailed notes for how they are used, particularly the elements that support input and the requesters. All of the lists support selection of the information within them, for lists that do not support selection you should indicate this information.
Figure 2-15. An essential user interface prototype to enroll a student in a seminar.
Figure 2-16 depicts a second example of an essential user interface prototype of a major UI element, in this case the student transcript. It is comprised of several display-only minor UI elements, such as the name and address of the student. It also contains several elements pertaining to the current status of the student, including basic financial information, which could have been grouped together had the SMEs felt it necessary. Normally when you build the actual transcript, perhaps as a printed report, you would use all of the space available to you, although it is interesting to note that there is significant whitespace left on the prototype. For example, the list of seminars likely would horizontally span the entire printed page, whereas this is not indicated in Figure 2-16 – sticky notes only come in a certain sizes and are only meant to model the relative size and positioning of the minor UI elements. In short, I got it close enough, which was exactly my goal. Notice how the seminar list in Figure 2-15 contains different information than the seminar list in Figure 2-16. This is because each list is used for a different purpose, in Figure 2-15 to determine which seminars are available and in Figure 2-16 to indicate which seminars have been taken by a student. Occasionally, the result will be a different informational content and a different purpose.

**Agile Modeling Principle – Open & Honest Communication**

People need to be free, and to perceive that they are free, to offer suggestions. This includes ideas pertaining to one or more models, perhaps someone has a new way to approach a portion of the design or has a new insight regarding a requirement; the delivery of bad news such as being behind schedule; or simply the current status of their work. Open and honest communication enables people to make better decisions because the quality of the information that they are basing them on is more accurate.

**Blatant Advertising – Purchase The Object Primer, 2nd Edition!**

*The Object Primer 2/e* is a straightforward, easy to understand introduction to object-oriented concepts, requirements, analysis, and design techniques applying the techniques of the Unified Modeling Language (UML). The Object Primer goes further to show you how to move from object modeling to object-oriented programming, showing how to implement your designs using Java and relational database. The book also describes the techniques of the Full Lifecycle Object-Oriented Testing (FLOOT) methodology to enable you to test all of your development artifacts. Finally, *The Object Primer 2/e* summarizes the leading software processes, including the enhanced version of the Rational Unified Process (RUP) and eXtreme Programming (XP). Visit www.ambysoft.com/theObjectPrimer.html for more details.
Figure 2-16. An essential user interface prototype for a student transcript.
2.12 Non-Functional Requirements

A non-functional requirement pertains to the technical aspects that your system must fulfill, such as performance-related issues, reliability issues, and availability issues. Non-functional requirements are often referred to as technical requirements. Examples of non-functional requirements are presented in Figure 2-17. You document non-functional requirements in the same manner as business rules (Section 2.2), potentially including a description, an example, a source, references to related technical requirements, and a revision history.

TR34 The system shall be unavailable for no less than 2 minutes in a 24-hour period.
TR78 A seminar search will occur within less than 3 seconds 95% of the time and no more than 10 seconds 99% of the time.

Figure 2-17. Potential non-functional requirements for the university system.

Agile Modeling Practice – Create Simple Content

You should keep the actual content of your models – your requirements, your analysis, your architecture, or your design – as simple as you possibly can while still fulfilling the needs of your project stakeholders. The implication is that you should not add additional aspects to your models unless they are justifiable -- if you do not have a requirement to add system auditing features then don't add that features to your models. Have the courage to trust that you can in fact add this feature when, and if, it is ever asked of you. This is along the lines of XP’s practice of Simple Design.

2.13 Sequence Diagrams (UML)

Sequence diagrams are used to model the logic of usage scenarios. A usage scenario is exactly what its name indicates – the description of a potential way that your system is used. The logic of a usage scenario may be part of a use case, perhaps an alternate course. It may also be one entire pass through a use case, such as the logic described by the basic course of action or a portion of the basic course of action plus one or more alternate scenarios. It may also be a pass through the logic contained in several use cases, for example a student enrolls in the university then immediately enrolls in three seminars. For example, Figure 2-18 models the basic course of action for the "Enroll in Seminar" use case. Sequence diagrams model the flow of logic within your system in a visual manner, enabling you to both document and validate your logic, and are commonly used for both analysis and design purposes.

The boxes across the top of the diagram represent classifiers or their instances, typically use cases, objects, classes, or actors. Because you can send messages to both objects and classes, objects respond to messages through the invocation of an operation and classes do so through the invocation of static operations, it makes sense to include both on sequence diagrams. Because actors initiate and take an active part in usage scenarios they are also included in sequence diagrams. Objects have labels in the standard UML format “name: ClassName” where “name” is optional (objects that have not been given a name on the diagram are called anonymous objects). Classes have labels in the format "ClassName," and actors have names in the format "Actor Name" – both UML standards as well. For example, in Figure 2-18 you see that the Student actor has the name "A Student" and is labeled with the stereotype <<actor>>. The instance of the major UI element representing "UI32 Seminar Selection Screen", is an
anonymous object with the name "SeminarSelector" and the stereotype <<UI>>. The Student class is indicated on the diagram, the box with the name "Student," because the static message "isEligible(name, studentNumber)" is sent to it. The instance of “Student” was given a name "theStudent" because it is used in several places as a parameter in a message, whereas the instance of the “StudentsFees” class did not need to be referenced anywhere else in the diagram and thus could be anonymous. The dashed lines hanging from the boxes are called object lifelines, representing the life span of the object during the scenario being modeled. The long, thin boxes on the lifelines are method-invocation boxes that indicate that processing is being performed by the target object/class to fulfill a message. The "X" at the bottom of a method-invocation box is a UML convention to indicate that an object has been removed from memory, typically the result of receiving a message with the stereotype of <<destroy>>. Messages are indicated as labeled arrows, when the source and target of a message is an object or class the label is the signature of the method invoked in response to the message. However, if either the source or target is a human actor then the message is labeled with brief text describing the information being communicated.

**Agile Modeling Principle – Travel Light**

Every artifact that you create, and then decide to keep, will need to be maintained over time. If you decide to keep seven models, then whenever a change occurs (a new/updated requirement, a new approach is taken by your team, a new technology is adopted, ...) you will need to consider the impact of that change on all seven models and then act accordingly. If you decide to keep only three models then you clearly have less work to perform to support the same change, making you more agile because you are traveling lighter. Similarly, the more complex/detailed your models are, the more likely it is that any given change will be harder to accomplish (the individual model is "heavier" and is therefore more of a burden to maintain). Every time you decide to keep a model you trade-off agility for the convenience of having that information available to your team in an abstract manner (hence potentially enhancing communication within your team as well as with project stakeholders). Never underestimate the seriousness of this trade-off. Someone trekking across the desert will benefit from a map, a hat, good boots, and a canteen of water they likely won’t make it if they burden themselves with hundreds of gallons of water, a pack full of every piece of survival gear imaginable, and a collection of books about the desert. Similarly, a development team that decides to develop and maintain a detailed requirements document, a detailed collection of analysis models, a detailed collection of architectural models, and a detailed collection of design models will quickly discover they are spending the majority of their time updating documents instead of writing source code.
Figure 2-18. A UML sequence diagram for the basic course of action of the use case of Figure 2-22.

**Agile Modeling Practice** – Depict Models Simply

When you consider the potential diagrams that you could apply (UML diagrams, user interface diagrams, data models, and so on) you quickly realize that the majority of the time you only require a subset of the diagramming notation available to you. A simple model that shows the key features that you are trying to understand, perhaps a class model depicting the primary responsibilities of classes and the relationships between them, often proves to be sufficient. Yes, you could model all the scaffolding code that you will need to write, all the getter and setter operations that your coding standards tell you to use, but what value would that add? Very little.
2.14 State Chart Diagram (UML)

Objects have both behavior and state, or in other words they do things and they know things. Some objects do and know more things, or at least more complicated things, than other objects. Some objects are incredibly complicated, so complex that developers can have difficulties understanding them. To better understand complex classes, particularly those that act in different manners depending on their state, you should develop one or more UML state chart diagrams describing how their instances work.

First, some basic terminology for state modeling. UML state chart diagrams depict the various states that an object may be in and the transitions between those states. In fact, in other modeling languages it is common for this type of a diagram to be called a state-transition diagram of even simply a state diagram. A state represents a stage in the behaviour pattern of an object, and like UML activity diagrams (Section 2.1) it is possible to have initial states and final states. An initial state, also called a creation state, is the one that an object is in when it is first created whereas a final state is one in which no transitions lead out of. A transition is a progression from one state to another and will be triggered by an event that is either internal or external to the object.

Figure 2-19 presents an example state chart diagram for the Seminar class during registration. The rounded rectangles represent states: You see that instances of Seminar can be in the “Proposed,” “Scheduled,” “Open For Enrollment,” “Full,” and “Closed to Enrollment” states. An object starts in an initial state, represented by the closed circle, and can end up in a final state, represented by the bordered circle. This is the exact same notation used by UML activity diagrams, a perfect example of the consistency of the UML.

Blatant Advertising – Hire Ronin International Consultants today!

Ronin International is a software consulting company that specializes in providing high-end object development services such as Agile Modeling (AM), Enterprise JavaBeans (EJB) development, software process mentoring and tailoring software architecture consulting (including reviews), and transitioning existing staff to object technologies and techniques. I am President of the company and work with clients around the world as a senior consultant. Visit www.ronin-intl.com for details.
Figure 2-19. A UML State Chart diagram for the Seminar class during registration.

The arrows in Figure 2-19 represent transitions, progressions from one state to another. For example, when a seminar is in the “Scheduled” state it can either be opened for enrollment or cancelled. Transitions can also have guards on them, conditions that must be true for the transition to be triggered. An example of a guard in Figure 2-19 is shown on the transition from the “Open For Enrollment” to the “Closed To Enrollment” state, the UML notation for which is in the format “[guard description]” where the text is free-form. It is also possible to indicate the invocation of methods on your transition, and example of which is shown on the same transition as the guard, the format being “methodName1(); methodName2();…” with the order in the listing implying the order in which they are invoked.

States are represented by the values of the attributes of an object. For example, a seminar is in the “Open For Enrollment” state when it has been flagged as open and there are seats available to be filled. It is possible to indicate the invocation of methods within a state, for example upon entry into the “Closed To Enrollment” state the method “notifyInstructor()” is invoked.
2.15 Use Case Diagram (UML)

The use-case diagram in Figure 2-20, provides an example, and depicts a collection of use cases, actors, their associations, a system boundary box (optional), and packages (optional). A use case describes a sequence of actions that provide a measurable value to an actor and is drawn as a horizontal ellipse. An actor is a person, organization, or external system that plays a role in one or more interactions with your system (actors are drawn as stick figures). Relationships between actors and classes are indicated in use-case diagrams, a relationship exists whenever an actor is involved with an interaction described by a use case. Relationships can also exist between use cases, a topic discussed below. Associations are modeled as lines connecting use cases and actors to one another, with an optional arrowhead on one end of the line indicating the direction of the initial invocation of the relationship. The rectangle around the use cases is called the system boundary box and as the name suggests it indicates the scope of your system – the use cases inside the rectangle represent the functionality that you intend to implement.

Figure 2-20. A use-case diagram for a simple university.

In the example depicted in Figure 2-20, students are enrolling in courses with the potential help of registrars. Professors input the marks that students earn on assignments and registrars authorize the distribution of transcripts (report cards) to students. Note how for some use cases there is more than one actor involved. Moreover, note how some associations have arrowheads – any given use-case association will have a zero or one arrowhead. The association between Student and “Enroll in seminar” indicates that this use case is initially invoked by a student and not by a registrar (the Registrar actor is also involved with this use case). It is important to understand that associations do not represent flows of information, they merely indicate that an actor is involved with a use case somehow. Yes, there is information flowing back and forth between the actor and the use case, for example students would need to indicate which seminars they wish to enroll in and the system would need to indicate to the student whether or not they have been enrolled. However, use-case diagrams do not model this sort of information. Information flow can be modeled using UML activity diagrams, covered in Section 2.1. The line between the “Enroll in seminar” use case and the Registrar actor has no arrowhead, indicating that it is not clear how the interaction between the system and registrars start. Perhaps a registrar may notice that a student needs help and offers aid whereas other times the student may request help from the registrar, important information that would be documented in the description of the use case. Actors are always involved with at least one use case and are always drawn on the outside edges of a use-case diagram.
2.15.1 Reuse in Use-Case Diagrams: <<extend>>, <<include>>, and Inheritance

One of your goals during analysis is to identify potential opportunities for reuse, a goal that you can work towards as you are developing your use case model. Potential reuse can be model through four generalization relationships supported by UML use case models: extend relationships between use cases, include relationships between use cases, inheritance between use cases, and inheritance between actors.

An extend association, formerly called an extends relationship in UML v1.2 and earlier, is a generalization relationship where an extending use case continues the behavior of a base use case. The extending use case accomplishes this by conceptually inserting additional action sequences into the base use case sequence. This allows an extending use case to continue the activity sequence of a base use case when the appropriate extension point is reached in the base use case and the extension condition is fulfilled. When the extending use case activity sequence is completed, the base use case continues. In Figure 2-21 you see that the use case "Enroll International Student in University" extends the use case "Enroll in University", the notation for doing so is simply a normal use case association with the stereotype of <<extend>>. In this case "Enroll in University" is the base use case and "Enroll International Student in University" is the extending use case.

A second way to indicate potential reuse within use case models exists in the form of include associations. An include association, formerly known as a uses relationship in UML v1.2 and earlier, is a generalization relationship denoting the inclusion of the behavior described by another use case. The best way to think of an include association is that it is the invocation of a use case by another one. In Figure

![Use-Case Diagram](image-url)
2-21 you see that the use case "Enroll in University" includes the use case "Enroll in Seminar", the notation for doing so is simply a normal use case association with the stereotype of <<include>>.

Use cases can inherit from other use cases, offering a third opportunity to indicate potential reuse. Figure 2-21 depicts an example of this, showing that "Enroll Family Member in University" inherits from the "Enroll In University" use case. Inheritance between use cases is not as common as either the use of extend or include associations but it is still possible. The inheriting use case would completely replace one or more of the courses of action of the inherited use case.

---

**Agile Modeling Practice – Apply Patterns Gently**

Effective modelers learn and then appropriately apply common architectural, design and analysis patterns in your models. However, as Martin Fowler points out in Is Design Dead? developers should consider easing into the application of a pattern, to apply it gently. This reflects the value of simplicity. In other words, if you SUSPECT that a pattern applies you should model it in such a way as to implement the minimal amount you need today but that makes it easy to refactor it later it is clear that applying the full-fledged pattern is in fact the simplest approach possible. In other words, don't over model. For example, you may recognize a good spot in your design to apply the GoF's Strategy pattern, but at the current moment you only have two algorithms to implement. The simplest approach might be to encapsulate each strategy in its own class and build an operation that chooses them appropriately and passes them the appropriate input. This is a partial implementation of Strategy that leaves you in a position to refactor your design if more algorithms need to be implemented, yet does not require you to build all the scaffolding that Strategy requires -- an approach that enables you to ease into application of the pattern.

---

**2.16 Use Cases: Essential and System**

A use case is a sequence of actions that provide a measurable value to an actor. Another way to look at it is that a use case describes a way in which a real-world actor interacts with the system. An essential use-case, sometimes called a business use case, is a simplified, abstract, generalized use case that captures the intentions of a user in a technology and implementation independent manner. An essential use case is a structured narrative, expressed in the language of the application domain and of users, comprising a simplified, generalized, abstract, technology free and implementation independent description of one task or interaction. An essential use case is complete, meaningful, and well designed from the point of view of users in some role or roles in relation to a system and that embodies the purpose or intentions underlying the interaction.

Consider the two versions of the use case “Enroll in seminar” presented in Figure 2-22, which presents a simplified system use case (also called a traditional or concrete use case), and in Figure 2-23 which presents it as an essential use case. The first thing to notice is that the “traditional” system use case has many implementation details embedded within it. For example, the concept of registrar disappeared and been replaced with the term system, indicating that a decision has been made to automate many of the mundane aspects of enrollment. Don’t worry, the concept of a registrar isn’t likely to completely go away, in fact this actor may reappear when the alternate course(s) of action are identified for the use case. The writer of system use cases is analyzing and describing requirements imposed by the problem intermingled with implicit decisions about what the user interface is going to be like, whereas the writer of an essential use case does not. The second thing to notice is that the system use case makes references to screen and reports, for example “UI23 Security Login Screen” and “UI89 Enrollment Summary Report,” and the essential use case does not. Once again this reflects implementation details, someone has decided that the system will be implemented as screens, as opposed to HTML pages perhaps, and printed reports. However, the essential use case could just as easily have referred to major user interface elements, the essential version of screens and reports, and to tell you the truth this is a practice that I
recommend. I did not include references to UI elements in Figure 2-23 to provide you with an example where this has not been done. Essential user interface prototyping is described in Section 2.11. Third, both versions of the use reference business rule definitions, such as “BR129 Determine Eligibility to Enroll,” because business rules reflect essential characteristics of your domain that your system must implement. Fourth, the system use case has more steps than the essential use case version. This in fact reflects my style of writing use cases, I believe that each use case step should reflect one step and one step only. There are several advantages to this approach: the use case becomes easier to test because each statement is easier to understand and to validate; alternate courses are easier to write because it is easier to branch from a statement when it does one thing only. Fifth, the use case steps are written in the active voice. For example, the statement "The registrar informs the student of the fees" is in active voice whereas "The student is informed of the fees by the registrar" is in passive voice. Writing in the active voice leads to sentences that are succinct. Finally, another style issue is that I like to end the basic course of action within a use case with a statement such as “The use case ends” or “The use case ends when…” to indicate that the logic for the course of action has been completely defined.

**Agile Modeling Practice – Use the Simplest Tools**

The vast majority of models can be drawn on a whiteboard, on paper or even the back of a napkin. Whenever you want to save one of these diagrams you can take a picture of it with a digital camera, or even simply transcribe it onto paper. This works because most diagrams are throwaways; their true value comes from drawing them to think through an issue, and once the issue is resolved the diagram doesn’t offer much value. As a result a whiteboard and markers are often your best modeling tool alternative: Use a drawing tool to create diagrams to present to important project stakeholders and occasionally use a modeling tool if and only if they provide value to your programming efforts such as the generation of code. Think of it like this: If you’re creating simple models, often models that are throwaways because if you are modeling to understand you likely don’t need to keep the model(s) around any more once you do understand the issue, then you likely don’t need to apply a complex modeling tool.
Name: Enroll in Seminar

Description:
Enroll an existing student in a seminar for which they are eligible.

Preconditions:
The Student is registered at the University.

Postconditions:
The Student will be enrolled in the course they want if they are eligible and there is room available.

Basic Course of Action:
1. A student wishes to enroll in a seminar.
2. The student inputs their name and student number into the system via “UI23 Security Login Screen.”
3. The system verifies that the student is eligible to enroll in seminars at the university according to business rule “BR129 Determine Eligibility to Enroll.”
4. The system displays “UI32 Seminar Selection Screen” which indicates the list of available seminars.
5. The student indicates the seminar that they wish to enroll in.
6. The system validates that the student is eligible to enroll in the seminar according to the business rule “BR130 Determine Student Eligibility to Enroll in a Seminar.”
7. The system validates that the seminar fits into the existing schedule of the student according to the business rule “BR143 Validate Student Seminar Schedule.”
8. The system calculates the fees for the seminar based on the fee published in the course catalog, applicable student fees, and applicable taxes. Apply business rules “BR 180 Calculate Student Fees” and “BR45 Calculate Taxes for Seminar.”
9. The system displays the fees via “UI33 Display Seminar Fees Screen.”
10. The system asks the student whether they would still like to enroll in the seminar.
11. The student indicates that they wish to enroll in the seminar.
12. The system enrolls the student in the seminar.
13. The system informs the student that the enrollment was successful via “UI88 Seminar Enrollment Summary Screen.”
14. The system bills the student for the seminar according to business rule ‘BR100 Bill Student for Seminar.”
15. The system asks the student if they would like a printed statement of the enrollment.
16. The student indicates they would like a printed statement.
17. The system prints the enrollment statement “UI89 Enrollment Summary Report.”
18. The use case ends when the student takes the printed statement.

Figure 2-22. “Enroll in seminar” as a “traditional” system use case.
Name: Enroll in Seminar

Description:
Enroll an existing student in a seminar for which they are eligible.

Preconditions:
The Student is registered at the University.

Postconditions:
The Student will be enrolled in the course they want if they are eligible and there is room available.

Basic Course of Action:
1. A student wishes to enroll in a seminar.
2. The student submits their name and student number to the registrar.
3. The registrar verifies that the student is eligible to enroll in seminars at the university according to business rule “BR129 Determine Eligibility to Enroll.”
4. The student indicates, from the list of available seminars, the seminar that they wish to enroll in.
5. The registrar validates that the student is eligible to enroll in the seminar according to the business rule “BR130 Determine Student Eligibility to Enroll in a Seminar.”
6. The registrar validates that the seminar fits into the existing schedule of the student according to the business rule “BR143 Validate Student Seminar Schedule.”
7. The registrar calculates the fees for the seminar based on the fee published in the course catalog, applicable student fees, and applicable taxes. Apply business rules “BR 180 Calculate Student Fees” and “BR45 Calculate Taxes for Seminar.”
8. The registrar informs the student of the fees.
9. The registrar verifies that the student still wishes to enroll in the seminar.
10. The student indicates they wish to enroll in the seminar.
11. The registrar enrolls the student in the seminar.
12. The registrar adds the appropriate fees to the student’s bill according to business rule “BR100 Bill Student for Seminar.”
13. The registrar provides the student with a confirmation that they are enrolled.
14. The use case ends.

Figure 2-23. “Enroll in seminar” as an essential use case.

Agile Modeling Practice – Update Only When It Hurts

You should update a model only when you absolutely need to, when not having the model updated is more painful than the effort of updating it. With this approach you discover that you update a smaller number of models than you would have in the past because the reality is that your models don't have to be perfect to provide value. The street map that I have to my town is over five years old, I know this because my own street doesn't appear on it and it's been in place for a little over five years, yet the map is still useful to me. Yes, I could purchase an updated map, one comes out every year, but why bother? Missing a few streets isn't painful enough to warrant this investment, it simply doesn't make sense to spend the money to purchase a new map every year when the one that I have is good enough. Too much time and money is wasted trying to keep models and documents in sync with source code, an impossible task to begin with, time and money that could be better spent developing new software.
2.17 User Interface Flow Diagrams

To your users, the user interface is the system. It is as simple as that. Does it not make sense that you should have some sort of diagram to help you model the user interface for your system? Essential user interface prototypes are an excellent means of documenting the requirements for your user interface, and you will see in Section 2.18 that user interface prototypes are great artifacts to develop your user interface design. The problem with both of these techniques is that you can quickly be bogged down in the details of the user interface and not see the bigger picture. Consequently, you often miss high-level relationships and interactions between the user interface elements of your application. User interface-flow diagrams – also called interface-flow diagrams, windows navigation diagrams, and context-navigation maps – enable you to model the high-level relationships between major user interface elements.

In Figure 2-24, you see the start at a user interface-flow diagram for the university system. Although the Unified Modeling Language (UML) does not yet support user interface-flow diagrams, I have applied a combination of the notations for UML activity diagrams (Section 2.1) and UML collaboration diagrams (Section 2.6) in the example. The boxes represent major user interface elements, modeled as you would objects, and the arrows represent the possible flow between them, modeled as you would transitions in activity diagrams. For example, when you are on the main menu screen you can use the “Enrollment Requestor” to take you to the “Enroll in Seminar” UI element. Once you are there you can either go back to the main menu (going back is always assumed), go to the “Professor Information” UI element or to the “Seminar Information” UI element. The addition of these two major UI elements was the result of the SMEs decision to enable students to find out more information about the instructors teaching a seminar and the detailed information regarding prerequisites to a seminar.

Figure 2-24. Initial user interface-flow diagram for the university system.
User interface-flow diagrams are typically used for one of two purposes. First, they are used to model the interactions that users have with your software as defined in a single use case. For example, the use case described in Figure 2-22 refers to several user interface elements and provides insight into how they are used. Based on this information you can develop a user interface-flow diagram that reflects the behavioral view of the single use case. Second, as you see in Figure 2-24 they enable you to gain a high-level overview of the user interface for your application. This overview is effectively the combination of all of the behavioral views derived from your use cases, the result being called the architectural view of your user interface. I prefer to take the high-level overview approach, also referred to as the architectural approach, because it enables me to understand the complete user interface for a system.

**Agile Modeling Practice – Apply Modeling Standards**

This practice is renamed from XP's Coding Standards, the basic idea is that developers should agree to and follow a common set of modeling standards on a software project. Just like there is value in following common coding conventions, clean code that follows your chosen coding guidelines is easier to understand and evolve than code that doesn't, there is similar value in following common modeling conventions. There is a wide variety of common modeling standards available to you, including the Object Management Group’s Unified Modeling Language (UML) which defines the notation and semantics for common object-oriented models. The UML provides a good start but it isn't sufficient – as you saw earlier not all possible modeling artifacts are encompassed by the UML. Furthermore, it says nothing about modeling style guidelines to create clean-looking diagrams. What is the difference between a style guideline and a standards. WRT to source code, a standard would be to name attributes in the format `attributeName` whereas a style guideline is to indent the code within a control structure (an if statement, a loop, ...) by one unit. WRT to models, a standard would be to use a square rectangle to model a class on a class diagram and a style guideline would be to have subclasses placed on diagrams to the south of their superclasses.

### 2.18 User Interface Prototype

User interface (UI) prototyping is an iterative analysis technique in which users are actively involved in the mocking-up of the UI for a system. UI prototyping has two purposes: first, it is an analysis technique because it enables you to explore the problem space that your system addresses. Second, UI prototyping enables you to explore the solution space of your system, at least from the point of view of its users, and provides a vehicle for you to communicate the possible UI design(s) of your system. As you see in the activity diagram depicted in Figure 2-25 there are four high-level steps to the UI prototyping process:
- Determine the needs of your users
- Build the prototype
- Evaluate the prototype
- Determine if you are finished
Figure 2-25. The iterative steps of prototyping.

Constantine and Lockwood (1999) provide valuable insight into the process of user interface prototyping. First, you cannot make everything simple – sometimes your software will be difficult to use because the problem that it addresses is inherently difficult. Your goal is to make your user interface as easy as possible to use, not simplistic. Second, they differentiate between the concepts of WYSIWYG, "what you see is what you get," and WYSIWYN, "what you see is what you need." Their point is that a good user interface fulfills the needs of the people that work with it, it isn't loaded with a lot of interesting by unnecessary features. Third, consistency is important in your user interface. Inconsistent user interfaces lead to less usable software, more programming, and greater support and training costs. Fourth, small details can make or break your user interface. Have you ever used some software, then discarded it for the product of a competitor, because you didn't like the way that it prints, saves files, or some other feature that you simply found too annoying to live with? I have. Although the rest of the software may have been great, that vendor lost my business because a portion of their product's user interface was deficient.
3. Summary

Object-oriented and component-based development is hard. You have a wide range of techniques, summarized briefly in this white paper, that you may apply when you are modeling. The UML is not complete, at least not for the development of business software, and to be successful you want to apply many of the techniques of agile modeling (http://www.agilemodeling.com) to make your modeling efforts successful.

4. Blatant Advertising

If you are interested in learning more about the techniques that I have described I invite you to pick up a copy of The Object Primer 2/e (http://www.ambysoft.com/theObjectPrimer.html) which describes each technique in detail and provides step-by-step instructions for developing each model effectively. If you'd like training in these techniques then Ronin International’s “Agile Modeling (AM) Workshop” (http://www.ronin-intl.com/services/agileModeling.html) is probably for you. It’s currently one of the few OOA&D courses available to you that includes both UML and non-UML material and it is the only one that focuses on the techniques of agile modeling (they’re brought to you from the source, Ronin International).
5. References and Recommended Reading


Copyright 2001 Ronin International
Visit http://www.ronin-intl.com for more development white papers


Copyright 2001 Ronin International
Visit [http://www.ronin-intl.com](http://www.ronin-intl.com) for more development white papers


### 5.1 Web-Based Resources


*Scott Ambler's Online Writings*. [http://www.ambysoft.com/onlineWritings.html](http://www.ambysoft.com/onlineWritings.html)
6. About the Author

7. Index

A
Activity diagrams ............................................3
Agile Alliance ..............................................1
Agile Modeling (AM) ......................................1
mailing list ................................................1
Author contacting ........................................44

B
Blatant Advertising ...................................22, 27, 38
Business rule ...............................................4

C
Change cases ................................................5
Class models ...............................................6
Class Responsibility Collaborator (CRC) model ........................................10
Collaboration diagrams ................................11
Component diagrams ....................................13
Constraints ..................................................15

D
Data Models ...............................................16
Deployment diagrams ..................................18

E
Essential modeling ......................................20
Essential use cases ......................................31
Essential user interface prototypes ...........20

F
FLOOT .....................................................22
Full Lifecycle Object-Oriented Testing ...........22

M
Mailing list ...............................................1

N
Non-functional requirements ......................24

O
Object Management Group (OMG) .............36
On-Site Customer .......................................11

P
Patterns .....................................................31
Persistence Models ....................................16

Practice
Active Stakeholder Participation .................11
Apply Modeling Standards .............................36
Apply Patterns Gently ..................................31
Apply The Right Artifact(s) .........................2, 4
Collective Ownership ..................................5
Create Several Models in Parallel .................9
Create Several Models Simultaneously ..........2
Create Simple Content ................................24
Depict Models Simply ................................26
Discard Temporary Models .........................12
Display Models Publicly ................................15
Formalize Contract Models .........................18
Iterate To Another Artifact .........................2, 15
Model In Small Increments .........................6
Model With Others .......................................20
Update Only When It Hurts ..........................34
Use The Simplest Tools ................................32

Principle
Content Is More Important Than
Representation ..............................................7
Know Your Models ....................................4
Model With A Purpose ..................................8
Multiple Models .........................................2, 16
Open & Honest Communication ..................22
Travel Light ..................................................25

R
Ronin International .....................................27
RUP .........................................................22

S
Sequence diagrams ......................................24
State chart diagrams ...................................27
System use cases .......................................31

T
The Object Primer 2/e ................................13, 22

U
UML
Activity diagrams .......................................3
Class models .............................................6
Collaboration diagrams ...............................11
Component diagrams ..................................13
Deployment diagrams ..................................18
insufficiency of ..........................................1
Sequence diagrams .....................................24
State chart diagrams ..................................27
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use-case diagrams</td>
<td>28</td>
</tr>
<tr>
<td>Unified Process</td>
<td>22</td>
</tr>
<tr>
<td>Use cases</td>
<td>31</td>
</tr>
<tr>
<td>Use-case diagrams</td>
<td>28</td>
</tr>
<tr>
<td>User interface prototype</td>
<td>20, 36</td>
</tr>
<tr>
<td>User interface prototypes</td>
<td>35</td>
</tr>
<tr>
<td>User interface-flow diagrams</td>
<td>35</td>
</tr>
<tr>
<td>WYSIWYG</td>
<td>37</td>
</tr>
<tr>
<td>WYSIWYN</td>
<td>37</td>
</tr>
<tr>
<td>XP</td>
<td></td>
</tr>
<tr>
<td>XP</td>
<td>2, 11, 16, 18, 22, 24, 36</td>
</tr>
</tbody>
</table>