Chapter 2

Development Process

As I've already mentioned, the UML grew out of a bunch of OO analysis and design methods. To some extent, all of them mixed a graphical modeling language with a process that described how to go about developing software.

Interestingly, as the UML was formed, the various players discovered that although they could agree on a modeling language, they most certainly could not agree on a process. As a result, they agreed to leave any agreement on process until later and to confine the UML to being a modeling language.

The title of this book is *UML Distilled*, so I could have safely ignored process. However, I don't believe that modeling techniques make any sense without knowing how they fit into a process. The way you use the UML depends a lot on the style of process you use.

As a result, I think that it's important to talk about process first so that you can see the context for using the UML. I'm not going to go into great detail on any particular process; I simply want to give you enough information to see this context and pointers to where you can find out more.

When you hear people discuss the UML, you often hear them talk about the Rational Unified Process (RUP). RUP is one process—or, more strictly, a process framework—that you can use with the UML. But other than the common involvement of various people from Rational and the name "unified," it doesn't have any special relationship to the UML. The UML can be used with any process. RUP is a popular approach and is discussed on page 25.

Iterative and Waterfall Processes

One of the biggest debates about process is that between waterfall and iterative styles. The terms often get misused, particularly as iterative is seen as fashionable, while the waterfall process seems to wear plaid trousers. As a result, many projects claim to do iterative development but are really doing waterfall.

The essential difference between the two is how you break up a project into smaller chunks. If you have a project that you think will take a year, few people are comfortable telling the team to go away for a year and to come back when done. Some breakdown is needed so that people can approach the problem and track progress.

The **waterfall** style breaks down a project based on activity. To build software, you have to do certain activities: requirements analysis, design, coding, and testing. Our 1-year project might thus have a 2-month analysis phase, followed by a 4-month design phase, followed by a 3-month coding phase, followed by a 3-month testing phase.

The **iterative** style breaks down a project by subsets of functionality. You might take a year and break it into 3-month iterations. In the first iteration, you'd take a quarter of the requirements and do the complete software life cycle for that quarter: analysis, design, code, and test. At the end of the first iteration, you'd have a system that does a quarter of the needed functionality. Then you'd do a second iteration so that at the end of 6 months, you'd have a system that does half the functionality.

Of course, the above is a simplified description, but it is the essence of the difference. In practice, of course, some impurities leak into the process.

With waterfall development, there is usually some form of formal handoff between each phase, but there are often backflows. During coding, something may come up that causes you to revisit the analysis and design. You certainly should not assume that all design is finished when coding begins. It's inevitable that analysis and design decisions will have to be revisited in later phases. However, these backflows are exceptions and should be minimized as much as possible.

With iteration, you usually see some form of exploration activity before the true iterations begin. At the very least, this will get a high-level view of the requirements: at least enough to break the requirements down into the iterations that will follow. Some high-level design decisions may occur during exploration too. At the other end, although each iteration should produce production-ready integrated software, it often doesn't quite get to that point and needs a stabilization period to iron out the last bugs. Also, some activities, such as user training, are left to the end.

You may well not put the system into production at the end of each iteration, but the system should be of production quality. Often, however, you can put the system into production at regular intervals; this is good because you get value from the system earlier and you get better-quality feedback. In this situation, you often hear of a project having multiple **releases**, each of which is broken down into several **iterations**.

Iterative development has come under many names: incremental, spiral, evolutionary, and jacuzzi spring to mind. Various people make distinctions among them, but the distinctions are neither widely agreed on nor that important compared to the iterative/waterfall dichotomy.

You can have hybrid approaches. [McConnell] describes the **staged delivery** life cycle whereby analysis and high-level design are done first, in a waterfall style, and then the coding and testing are divided up into iterations. Such a project might have 4 months of analysis and design followed by four 2-month iterative builds of the system.

Most writers on software process in the past few years, especially in the object-oriented community, dislike the waterfall approach. Of the many reasons for this, the most fundamental is that it's very difficult to tell whether the project is truly on track with a waterfall process. It's too easy to declare victory with early phases and hide a schedule slip. Usually, the only way you can really tell whether you are on track is to produce tested, integrated software. By doing this repeatedly, an iterative style gives you better warning if something is going awry.

For that reason alone, I strongly recommend that projects do not use a pure waterfall approach. You should at least use staged delivery, if not a more pure iterative technique.

The OO community has long been in favor of iterative development, and it's safe to say that pretty much everyone involved in building the UML is in favor of at least some form of iterative development. My sense of industrial practice is that waterfall development is still the more common approach, however. One reason for this is what I refer to as pseudoiterative development: People claim to be doing iterative development but are in fact doing waterfall. Common symptoms of this are:

- "We are doing one analysis iteration followed by two design iterations. . . ."
- "This iteration's code is very buggy, but we'll clean it up at the end."

It is particularly important that each iteration produces tested, integrated code that is as close to production quality as possible. Testing and integration are the hardest activities to estimate, so it's important not to have an open-ended activity like that at the end of the project. The test should be that any iteration that's not scheduled to be released could be released without substantial extra development work.

A common technique with iterations is to use **time boxing**. This forces an iteration to be a fixed length of time. If it appears that you can't build all you intended to build during an iteration, you must decide to slip some functionality

from the iteration; you must not slip the date of the iteration. Most projects that use iterative development use the same iteration length throughout the project; that way, you get a regular rhythm of builds.

I like time boxing because people usually have difficulty slipping functionality. By practicing slipping function regularly, they are in a better position to make an intelligent choice at a big release between slipping a date and slipping function. Slipping function during iterations is also effective at helping people learn what the real requirements priorities are.

One of the most common concerns about iterative development is the issue of rework. Iterative development explicitly assumes that you will be reworking and deleting existing code during the later iterations of a project. In many domains, such as manufacturing, rework is seen as a waste. But software isn't like manufacturing; as a result, it often is more efficient to rework existing code than to patch around code that was poorly designed. A number of technical practices can greatly help make rework be more efficient.

- **Automated regression tests** help by allowing you to quickly detect any defects that may have been introduced when you are changing things. The xUnit family of testing frameworks is a particularly valuable tool for building automated unit tests. Starting with the original JUnit **http://junit.org**, there are now ports to almost every language imaginable (see **http://www.xprogramming.com/software.htm**). A good rule of thumb is that the size of your unit test code should be about the same size as your production code.

- **Refactoring** is a disciplined technique for changing existing software [Fowler, refactoring]. Refactoring works by using a series of small behavior-preserving transformations to the code base. Many of these transformations can be automated (see **http://www.refactoring.com**).

- **Continuous integration** keeps a team in sync to avoid painful integration cycles [Fowler and Foemmel]. At the heart of this lies a fully automated build process that can be kicked off automatically whenever any member of the team checks code into the code base. Developers are expected to check in daily, so automated builds are done many times a day. The build process includes running a large block of automated regression tests so that any inconsistencies are caught quickly so they can be fixed easily.

All these technical practices have been popularized recently by Extreme Programming [Beck], although they were used before and can, and should, be used whether or not you use XP or any other agile process.

Predictive and Adaptive Planning

One reason that the waterfall endures is the desire for predictability in software development. Nothing is more frustrating than not having a clear idea how much it will cost to build some software and how long it will take to build it.

A predictive approach looks to do work early in the project in order to yield a greater understanding of what has to be done later. This way, you can reach a point where the latter part of the project can be estimated with a reasonable degree of accuracy. With **predictive planning,** a project has two stages. The first stage comes up with plans and is difficult to predict, but the second stage is much more predictable because the plans are in place.

This isn't necessarily a black-and-white affair. As the project goes on, you gradually get more predictability. And even once you have a predictive plan, things will go wrong. You simply expect that the deviations become less significant once a solid plan is in place.

However, there is a considerable debate about whether many software projects can ever be predictable. At the heart of this question is requirements analysis. One of the unique sources of complexity in software projects is the difficulty in understanding the requirements for a software system. The majority of software projects experience significant **requirements churn**: changes in requirements in the later stages of the project. These changes shatter the foundations of a predictive plan. You can combat these changes by freezing the requirements early on and not permitting changes, but this runs the risk of delivering a system that no longer meets the needs of its users.

This problem leads to two very different reactions. One route is to put more effort into the requirements process itself. This way, you may get a more accurate set of requirements, which will reduce the churn.

Another school contends that requirements churn is unavoidable, that it's too difficult for many projects to stabilize requirements sufficiently to use a predictive plan. This may be either owing to the sheer difficulty of envisioning what software can do or because market conditions force unpredictable changes. This school of thought advocates **adaptive planning,** whereby predictivity is seen as an illusion. Instead of fooling ourselves with illusory predictability, we should face the reality of constant change and use a planning approach that treats change as a constant in a software project. This change is controlled so that the project delivers the best software it can; but although the project is controllable, it is not predictable.

The difference between a predictive project and an adaptive project surfaces in many ways that people talk about how the project goes. When people talk about

a project that's doing well because it's going according to plan, that's a predictive form of thinking. You can't say "according to plan" in an adaptive environment, because the plan is always changing. This doesn't mean that adaptive projects don't plan; they usually plan a lot, but the plan is treated as a baseline to assess the consequences of change rather than as a prediction of the future.

With a predictive plan, you can develop a fixed-price/fixed-scope contract. Such a contract says exactly what should be built, how much it will cost, and when it will be delivered. Such fixing isn't possible with an adaptive plan. You can fix a budget and a time for delivery, but you can't fix what functionality will be delivered. An adaptive contract assumes that the users will collaborate with the development team to regularly reassess what functionality needs to be built and will cancel the project if progress ends up being too slow. As such, an adaptive planning process can be fixed price/variable scope.

Naturally, the adaptive approach is less desirable, as anyone would prefer greater predictability in a software project. However, predictability depends on a precise, accurate, and stable set of requirements. If you cannot stabilize your requirements, the predictive plan is based on sand and the chances are high that the project goes off course. This leads to two important pieces of advice.

1. Don't make a predictive plan until you have precise and accurate requirements and are confident that they won't significantly change.
2. If you can't get precise, accurate, and stable requirements, use an adaptive planning style.

Predictivity and adaptivity feed into the choice of life cycle. An adaptive plan absolutely requires an iterative process. Predictive planning can be done either way, although it's easier to see how it works with waterfall or a staged delivery approach.

Agile Processes

In the past few years, there's been a lot of interest in agile software processes. *Agile* is an umbrella term that covers many processes that share a common set of values and principles as defined by the Manifesto of Agile Software Development (**http://agileManifesto.org**). Examples of these processes are Extreme Programming (XP), Scrum, Feature Driven Development (FDD), Crystal, and DSDM (Dynamic Systems Development Method).

In terms of our discussion, agile processes are strongly adaptive in their nature. They are also very much people-oriented processes. Agile approaches

assume that the most important factor in a project's success is the quality of the people on the project and how well they work together in human terms. Which process they use and which tools they use are strictly second-order effects.

Agile methods tend to use short, time-boxed iterations, most often of a month or less. Because they don't attach much weight to documents, agile approaches disdain using the UML in blueprint mode. Most use the UML in sketch mode, with a few advocating using it as a programming language.

Agile processes tend to be low in **ceremony**. A high-ceremony, or heavyweight, process has a lot of documents and control points during the project. Agile processes consider that ceremony makes it harder to make changes and works against the grain of talented people. As a result, agile processes are often characterized as **lightweight**. It's important to realize that the lack of ceremony is a consequence of adaptivity and people orientation rather than a fundamental property.

Rational Unified Process

Although the Rational Unified Process (RUP) is independent of the UML, the two are often talked about together. So I think it's worth saying a few things about it here.

Although RUP is called a process, it actually is a process framework, providing a vocabulary and loose structure to talk about processes. When you use RUP, the first thing you need to do is choose a **development case**: the process you are going to use in the project. Development cases can vary widely, so don't assume that your development case will look that much like any other development case. Choosing a development case needs someone early on who is very familiar with RUP: someone who can tailor RUP for a particular project's needs. Alternatively, there is a growing body of packaged development cases to start from.

Whatever the development case, RUP is essentially an iterative process. A waterfall style isn't compatible with the philosophy of RUP, although sadly it's not uncommon to run into projects that use a waterfall-style process and dress it up in RUP's clothes.

All RUP projects should follow four phases.

1. **Inception** makes an initial evaluation of a project. Typically in inception, you decide whether to commit enough funds to do an elaboration phase.

2. **Elaboration** identifies the primary use cases of the project and builds software in iterations in order to shake out the architecture of the system. At

the end of elaboration, you should have a good sense of the requirements and a skeletal working system that acts as the seed of development. In particular, you should have found and resolved the major risks to the project.

3. **Construction** continues the building process, developing enough functionality to release.

4. **Transition** includes various late-stage activities that you don't do iteratively. These may include deployment into the data center, user training, and the like.

There's a fair amount of fuzziness between the phases, especially between elaboration and construction. For some, the shift to construction is the point at which you can move into a predictive planning mode. For others, it merely indicates the point at which you have a broad vision of requirements and an architecture that you think is going to last the rest of the project.

Sometimes, RUP is referred to as the Unified Process (UP). This is usually done by organizations that wish to use the terminology and overall style of RUP without using the licensed products of Rational Software. You can think of RUP as Rational's product offering based on the UP, or you can think of RUP and UP as the same thing. Either way, you'll find people who agree with you.

Fitting a Process to a Project

Software projects differ greatly from one another. The way you go about software development depends on many factors: the kind of system you're building, the technology you're using, the size and distribution of the team, the nature of the risks, the consequences of failure, the working styles of the team, and the culture of the organization. As a result, you should never expect there to be a one-size-fits-all process that will work for all projects.

Consequently, you always have to adapt a process to fit your particular environment. One of the first things you need to do is look at your project and consider which processes seem close to a fit. This should give you a short list of processes to consider.

You should then consider what adaptations you need to make to fit them to your project. You have to be somewhat careful with this. Many processes are difficult to fully appreciate until you've worked with them. In these cases, it's often worth using the process out of the box for a couple of iterations until you learn how it works. Then you can start modifying the process. If from the beginning you are more familiar with how a process works, you can modify it

Patterns

The UML tells you how to express an object-oriented design. Patterns look, instead, at the results of the process: example designs.

Many people have commented that projects have problems because the people involved were not aware of designs that are well known to those with more experience. Patterns describe common ways of doing things and are collected by people who spot repeating themes in designs. These people take each theme and describe it so that other people can read the pattern and see how to apply it.

Let's look at an example. Say that you have some objects running in a process on your desktop and that they need to communicate with other objects running in another process. Perhaps this process is also on your desktop; perhaps it resides elsewhere. You don't want the objects in your system to have to worry about finding other objects on the network or executing remote procedure calls.

What you can do is create a proxy object within your local process for the remote object. The proxy has the same interface as the remote object. Your local objects talk to the proxy, using the usual in-process message sends. The proxy then is responsible for passing any messages on to the real object, wherever it might reside.

Proxies are a common technique used in networks and elsewhere. People have a lot of experience using proxies, knowing how they can be used, what advantages they can bring, their limitations, and how to implement them. Methods books like this one don't discuss this knowledge; all they discuss is how you can diagram a proxy, although useful, is not as useful as discussing the experience involving proxies.

In the early 1990s, some people began to capture this experience. They formed a community interested in writing patterns. These people sponsor conferences and have produced several books.

The most famous patterns book to emerge from this group is [Gang of Four], which discusses 23 design patterns in detail. If you want to know about proxies, this book spends ten pages on the subject, giving details about how the objects work together, the benefits and limitations of the pattern, common variations, and implementation tips.

A pattern is much more than a model. A pattern must also include the reason why it is the way it is. It is often said that a pattern is a solution to a problem. The pattern must identify the problem clearly, explain why

it solves the problem, and also explain the circumstances under which the pattern works and doesn't work.

Patterns are important because they are the next stage beyond understanding the basics of a language or a modeling technique. Patterns give you a series of solutions and also show you what makes a good model and how you go about constructing a model. Patterns teach by example.

When I started out, I wondered why I had to invent things from scratch. Why didn't I have handbooks to show me how to do common things? The patterns community is trying to build these handbooks.

There are now many patterns books out there, and they vary greatly in quality. My favorites are [Gang of Four], [POSA1], [POSA2], [Core J2EE Patterns], [Pont], and with suitable immodesty [Fowler, AP] and [Fowler, P of EAA]. You can also take a look at the patterns home page: **http://www.hillside.net/patterns.**

from the beginning. Remember that it's usually easier to start with too little and add things than it is to start with too much and take things away.

However confident you are with your process when you begin, it's essential to learn as you go along. Indeed, one of the great benefits of iterative development is that it supports frequent process improvement.

At the end of each iteration, conduct an **iteration retrospective**, whereby the team assembles to consider how things went and how they can be improved. A couple of hours is plenty if your iterations are short. A good way to do this is to make a list with three categories:

1. *Keep:* things that worked well that you want to ensure you continue to do
2. *Problems:* areas that aren't working well
3. *Try:* changes to your process to improve it

You can start each iteration retrospective after the first by reviewing the items from the previous session and seeing how things have changed. Don't forget the list of things to keep; it's important to keep track of things that are working. If you don't do that, you can lose a sense of perspective on the project and potentially stop paying attention to winning practices.

At the end of a project or at a major release, you may want to consider a more formal **project retrospective** that will last a couple of days; see **http://www.retrospectives.com/** and [Kerth] for more details. One of my biggest irri-

tations is how organizations consistently fail to learn from their own experience and end up making expensive mistakes time and time again.

Fitting the UML into a Process

When they look at graphical modeling languages, people usually think of them in the context of a waterfall process. A waterfall process usually has documents that act as the handoffs between analysis, design, and coding phases. Graphical models can often form a major part of these documents. Indeed, many of the structured methods from the 1970s and 1980s talk a lot about analysis and design models like this.

Whether or not you use a waterfall approach, you still do the activities of analysis, design, coding, and testing. You can run an iterative project with 1-week iterations, with each week a miniwaterfall.

Using the UML doesn't necessarily imply developing documents or feeding a complex CASE tool. Many people draw UML diagrams on whiteboards only during a meeting to help communicate their ideas.

Requirements Analysis

The activity of requirements analysis involves trying to figure out what the users and customers of a software effort want the system to do. A number of UML techniques can come in handy here:

- Use cases, which describe how people interact with the system.
- A class diagram drawn from the conceptual perspective, which can be a good way of building up a rigorous vocabulary of the domain.
- An activity diagram, which can show the work flow of the organization, showing how software and human activities interact. An activity diagram can show the context for use cases and also the details of how a complicated use case works.
- A state diagram, which can be useful if a concept has an interesting life cycle, with various states and events that change that state.

When working in requirements analysis, remember that the most important thing is communication with your users and customers. Usually, they are not software people and will be unfamiliar with the UML or any other technique.

Even so, I've had success using these techniques with nontechnical people. To do this, remember that it's important to keep the notation to a minimum. Don't introduce anything that specific to the software implementation.

Be prepared to break the rules of the UML at any time if it helps you communicate better. The biggest risk with using the UML in analysis is that you draw diagrams that the domain experts don't fully understand. A diagram that isn't understood by the people who know the domain is worse than useless; all it does is breed a false sense of confidence for the development team.

Design

When you are doing design, you can get more technical with your diagrams. You can use more notation and be more precise about your notation. Some useful techniques are

- Class diagrams from a software perspective. These show the classes in the software and how they interrelate.
- Sequence diagrams for common scenarios. A valuable approach is to pick the most important and interesting scenarios from the use cases and use CRC cards or sequence diagrams to figure out what happens in the software.
- Package diagrams to show the large-scale organization of the software.
- State diagrams for classes with complex life histories.
- Deployment diagrams to show the physical layout of the software.

Many of these same techniques can be used to document software once it's been written. This may help people find their way around the software if they have to work on it and are not familiar with the code.

With a waterfall life cycle, you would do these diagrams and activities as part of the phases. The end-of-phase documents usually include the appropriate UML diagrams for that activity. A waterfall style usually implies that the UML is used as a blueprint.

In an iterative style, the UML diagrams can be used in either a blueprint or a sketch style. With a blueprint, the analysis diagrams will usually be built in the iteration prior to the one that builds the functionality. Each iteration doesn't start from scratch; rather, it modifies the existing body of documents, highlighting the changes in the new iteration.

Blueprint designs are usually done early in the iteration and may be done in pieces for different bits of functionality that are targeted for the iteration.

Again, iteration implies making changes to an existing model rather than building a new model each time.

Using the UML in sketch mode implies a more fluid process. One approach is to spend a couple of days at the beginning of an iteration, sketching out the design for that iteration. You can also do short design sessions at any point during the iteration, setting up a quick meeting for half an hour whenever a developer starts to tackle a nontrivial function.

With a blueprint, you expect the code implementation to follow the diagrams. A change from the blueprint is a deviation that needs review from the designers who did the blueprint. A sketch is usually treated more as a first cut at the design; if, during coding, people find that the sketch isn't exactly right, they should feel free to change the design. The implementors have to use their judgment as to whether the change needs a wider discussion to understand the full ramifications.

One of my concerns with blueprints is my own observation that it's very hard to get them right, even for a good designer. I often find that my own designs do not survive contact with coding intact. I still find UML sketches useful, but I don't find that they can be treated as absolutes.

In both modes, it makes sense to explore a number of design alternatives. It's usually best to explore alternatives in sketch mode so that you can quickly generate and change the alternatives. Once you pick a design to run with, you can either use that sketch or detail it into a blueprint.

Documentation

Once you have built the software, you can use the UML to help document what you have done. For this, I find UML diagrams useful for getting an overall understanding of a system. In doing this, however, I should stress that I do not believe in producing detailed diagrams of the whole system. To quote Ward Cunningham [Cunningham]:

> *Carefully selected and well-written memos can easily substitute for traditional comprehensive design documentation. The latter rarely shines except in isolated spots. Elevate those spots . . . and forget about the rest. (p. 384)*

I believe that detailed documentation should be generated from the code—like, for instance, JavaDoc. You should write additional documentation to highlight important concepts. Think of these as comprising a first step for the reader before he or she goes into the code-based details. I like to structure these as prose documents, short enough to read over a cup of coffee, using UML diagrams to

help illustrate the discussion. I prefer the diagrams as sketches that highlight the most important parts of the system. Obviously, the writer of the document needs to decide what is important and what isn't, but the writer is much better equipped than the reader to do that.

A package diagram makes a good logical road map of the system. This diagram helps me understand the logical pieces of the system and see the dependencies and keep them under control. A deployment diagram (see Chapter 8), which shows the high-level physical picture, may also prove useful at this stage.

Within each package, I like to see a class diagram. I don't show every operation on every class. I show only the important features that help me understand what is in there. This class diagram acts as a graphical table of contents.

The class diagram should be supported by a handful of interaction diagrams that show the most important interactions in the system. Again, selectivity is important here; remember that, in this kind of document, comprehensiveness is the enemy of comprehensibility.

If a class has complex life-cycle behavior, I draw a state machine diagram (see Chapter 10) to describe it. I do this only if the behavior is sufficiently complex, which I find doesn't happen often.

I'll often include some important code, written in a literate program style. If a particularly complex algorithm is involved, I'll consider using an activity diagram (see Chapter 11) but only if it gives me more understanding than the code alone.

If I find concepts that are coming up repeatedly, I use patterns (page 27) to capture the basic ideas.

One of the most important things to document is the design alternatives you didn't take and why you didn't do them. That's often the most forgotten but most useful piece of external documentation you can provide.

Understanding Legacy Code

The UML can help you figure out a gnarly bunch of unfamiliar code in a couple of ways. Building a sketch of key facts can act as a graphical note-taking mechanism that helps you capture important information as you learn about it. Sketches of key classes in a package and their key interactions can help clarify what's going on.

With modern tools, you can generate detailed diagrams for key parts of a system. Don't use these tools to generate big paper reports; instead, use them to drill into key areas as you are exploring the code itself. A particularly nice capability is that of generating a sequence diagram to see how multiple objects collaborate in handling a complex method.

Choosing a Development Process

I'm strongly in favor of iterative development processes. As I've said in this book before: You should use iterative development only on projects that you want to succeed.

Perhaps that's a bit glib, but as I get older, I get more aggressive about using iterative development. Done well, it is an essential technique, one you can use to expose risk early and to obtain better control over development. It is not the same as having no management, although to be fair, I should point out that some have used it that way. It does need to be well planned. But it is a solid approach, and every OO development book encourages using it—for good reason.

You should not be surprised to hear that as one the authors of the Manifesto for Agile Software Development, I'm very much a fan of agile approaches. I've also had a lot of positive experiences with Extreme Programming, and certainly you should consider its practices very seriously.

Where to Find Out More

Books on software process have always been common, and the rise of agile software development has led to many new books. Overall, my favorite book on process in general is [McConnell]. He gives a broad and practical coverage of many of the issues involved in software development and a long list of useful practices.

From the agile community, [Cockburn, agile] and [Highsmith] provide a good overview. For a lot of good advice about applying the UML in an agile way, see [Ambler].

One of the most popular agile methods is Extreme Programming (XP), which you can delve into via such Web sites as **http://xprogramming.com** and **http://www.extremeprogramming.org.** XP has spawned many books, which is why I now refer to it as the formerly lightweight methodology. The usual starting point is [Beck].

Although it's written for XP, [Beck and Fowler] gives more details on planning an iterative project. Much of this is also covered by the other XP books, but if you're interested only in the planning aspect, this would be a good choice.

For more information on the Rational Unified Process, my favorite introduction is [Kruchten].

blank page