13 Iterative Models

When to use iterative development? You should use iterative development only on projects that you want to succeed.

—Martin Fowler

Control is for beginners.

—Deborah Mills-Scofield

Iteration is truly the mother of invention.

—Mary Brodie

WHAT YOU WILL LEARN IN THIS CHAPTER:

- ➤ Differences between predictive, iterative, incremental, and agile approaches
- ➤ Benefits of prototyping and kinds of prototypes
- ➤ Spiral, Unified Process (plus variations), and Cleanroom development models

Predictive software development has some big advantages. It's predictable, encourages a lot of up-front design (hence the nickname big design up front or BDUF), and gives a certain inevitability to a project.

Unfortunately, that inevitably can lead to either success or failure. If the design is correct and everything stays on track, the project is like a luxury train coasting majestically into Grand Central Station. However, if something goes wrong, the project is more like a train engulfed in flames and speeding toward a dynamited bridge.

In recent years, software organizations have spent a lot of effort developing techniques that help keep projects headed in the right direction. Lumping all those models and techniques together would make for a bloated chapter, so I decided to split them up a bit.

This chapter discusses one of the techniques that is easiest to apply to any other development model: iteration. In an iterative model, you build the final application incrementally. You start with a minimally working program and then add pieces to it, making it better and better, until you decide the application is as good as it can be (or at least good enough).

I actually snuck a little iteration into the preceding chapter with the incremental waterfall model. This chapter focuses on other iterative variations. The next chapter introduces a bunch of other techniques and approaches that can help keep software engineering efforts on track.

ITERATIVE VERSUS PREDICTIVE

The problem with predictive models is that they are ill-suited to handle unexpected change. They can deal with small changes (such as customers deciding they want combo boxes instead of list boxes on their forms), but they don't handle big changes well (such as customers deciding they want 20 extra reports that are all viewable on a desktop computer, tablet, or smartphone).

Predictive projects spend a lot of effort at the beginning figuring out exactly what they will do. After you gather requirements and commit to a schedule, it's hard to change course. In theory you could stop the project at any point and head in a new direction. In practice that rarely happens. Changing direction in a big way would essentially mean starting over. You would have to go through the requirements and design phases again. You would also have to abandon all the work you did in those phases the first time around. If the changes are big enough, this is practically the same as declaring the project a failure (no one wants to do that) and starting a new one.

Even if you manage to point the project in a new direction, there's no guarantee that you won't need to do it all over again. Actually, because your customers have already shown that they're willing to put you through all that pain and inconvenience, further changes seem if anything more likely.

One of the biggest reasons why those changes are so painful is the size of the commitment you've made. If you're three years into a five-year schedule, you've already invested a lot of effort in the project and you have a lot to lose. But what if you were only three months into a five-month plan? Scrapping the work-to-date and starting over would still be annoying, but it would be much less painful. (You might even get to keep your job.)

Predictive models also don't handle fuzzy requirements well. Unless you nail down the requirements precisely at the beginning of the project, it's impossible to create a solid schedule. How can you plan to build something with unknown requirements? (Imagine what would happen if you went to a real estate developer and said, "We want to build a new shopping mall. We don't know exactly where it will be or what it will look like, but start building it and we'll work out the details later.")

Iterative models address those problems by building the application incrementally. They start by building the smallest program that is reasonably useful. Then they use a series of *increments* to add more features to the program until it's finished (if it is ever finished).

Because each increment has a relatively small duration (compared to a predictive project), you're committed to a smaller amount of work. If you decide that the project is heading in the wrong direction, you need to stop only the most recent increment and start a new one instead of canceling the whole project and starting over.

Better still, because you can reorient the project before each new increment, you're less likely to need to cancel anything.

You may have trouble foreseeing the direction a predictive project should take four years from now. It's much easier to guess where you should be headed four months from now. Even if you decide that you need to adjust course, you can probably finish the current iteration and make the adjustments in the next one.

Iterative models also handle fuzzy requirements reasonably well. If you're unsure of some of the application's requirements, you can start building the parts you do understand and figure out the rest later. Sometimes, you'll learn things building the first part of the system that will make the rest of the requirements clear. Other times the requirements clarify themselves over time.

The preceding chapter described an iterative waterfall model that uses a series of waterfall-style projects to incrementally refine an application. Figure 13-1 shows the stages in an iterative waterfall project.

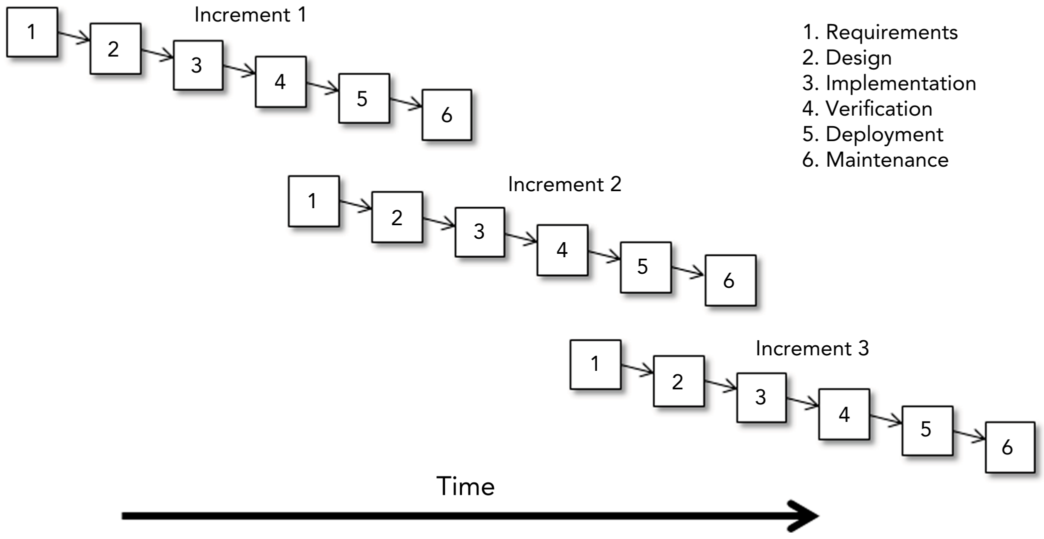

FIGURE 13-1: Iterative models use a series of development efforts to incrementally refine an application.

Other models also use iterative approaches. Many of the models described in the next chapter use iterative techniques to help stay on the right track.

You can even use iterative methods for just one part of a project. For example, you can use iterative prototyping to refine a project's requirements. After that, you could use waterfall, sashimi, or some other model to finish development.

The following sections describe some common iterative techniques that you can use to improve a project's chance of success.

ITERATIVE VERSUS INCREMENTAL

Normally, you think of an iterative project as running through several cycles, each of which provides an incremental improvement over the preceding version. Technically, however, that's not necessarily the case, so an iterative project might not be incremental.

For example, suppose in version 1 of a project you produce a usable application, but its code doesn't follow good programming standards. In version 2 you rewrite the code to make the project more maintainable. Version 2 doesn't add any new features to the application, so in some sense you might not think of it as an incremental improvement over version 1. The process is iterative but not incremental.

Karl Scotland provides an interesting perspective on this issue at `availagility.co.uk/2009/12/22/fidelity-the-lost-dimension-of-the-iron-triangle`. He argues that the difference between iterative and incremental development is clear if you consider the fidelity of a project's features. By *fidelity* he means the completeness of the feature. For example, a low-fidelity real-estate search screen might let you search for houses by price. A high-fidelity version would let you search by price, square feet, number of bedrooms, number of bathrooms, availability of high-speed Internet, and distance to the nearest ice cream store. (Karl explicitly avoids the term "quality" for this because he doesn't want to get into an argument about releasing low-quality code. All the code is assumed to be high quality. It's just some versions of a feature might do more than others.)

Now suppose you're working on a project that provides three features. Here's how you might use fidelity to describe different development approaches:

- **Predictive—**Provides all three features at the same time with full fidelity.
- **Iterative—**Initially provides all three features at a low (but usable) fidelity. Later iterations provide higher and higher fidelity until all the features are provided with full fidelity.
- **Incremental—**Initially provides the fewest possible features for a usable application, but all the features present are provided with full fidelity. Later versions add more features, always at full fidelity.
- **Agile—**Initially provides the fewest possible features at low fidelity. Later versions improve the fidelity of existing features and add new features. Eventually all the features are provided at full fidelity. (The next chapter says more about agile development models.)

Figure 13-2 shows a representation of these approaches. The rectangles represent the application's features. In the predictive model, the users don't receive any program until the application is completely finished. The iterative model starts with low-fidelity versions of every feature and then improves them over time. The incremental model starts with nothing and then adds new features with full fidelity. Finally, the agile model starts with some features of low fidelity and over time improves fidelity and adds more features.

All four of those approaches end with an application that includes all the features at full fidelity. It's the routes they take to get to their final solutions that differ.

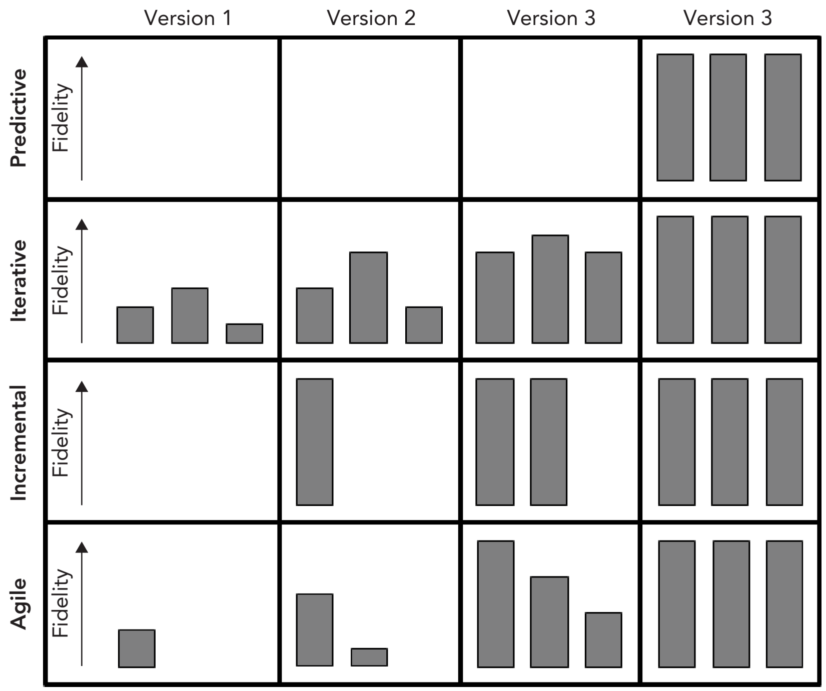

FIGURE 13-2: Different development approaches add features and increase fidelity in different ways.

The difference is somewhat pedantic, but it's probably worth knowing just so you're not confused when a senior executive starts throwing around terms he doesn't really understand.

PROTOTYPES

The section "Prototypes" in Chapter 4 briefly described prototypes. This section provides some additional details and focuses on how prototypes can be useful in iterative development.

A *prototype* is a simplified model that demonstrates some behavior that you want to study. Typically, a software prototype is a program that mimics part of the application you want to build.

Two important facts about prototypes are that they don't need to work the same way the final application will, and they don't need to implement all the features of the final application. They just give you a glimpse of a piece of the final application.

For example, suppose you're building a product ordering system. You enter some parameters such as a date range or a customer ID number, and click a List button to make the system display a list of matching orders. You can then double-click to view an order's details. From the order detail, you can click links to jump to detailed information about the items in the order or to see the customer's contact information.

During the requirements gathering phase, you might build a prototype to let the customers see what the finished application will look like. When you click the List button, the prototype displays a predefined list of orders. When you double-click an order, the prototype doesn't display information for that order. Instead it displays information about a preselected order. Finally, if you click a link on the order, you can see information about the preselected customer.

This prototype doesn't work exactly the same way the finished application will work. If it did, it would be the actual application and not just a prototype. However, it lets the customers see what the application will look like. It lets them click some buttons and links so that they can try out the program's method for interacting with users.

After the customers experiment with the prototype, they can give you feedback to help refine the requirements. For example, when they see the order list, they may think of other fields they want to display. After they try double-clicking to open an order's detail information, they may decide they would rather check boxes next to one or more orders and then click a Detail button to open detail pages for all the selected orders. When they view a customer's information, they may decide they want a quick way to see that customer's order history.

Although software prototypes are often programs, you can make other kind of prototypes using less sophisticated techniques. For example, you might mock up some screens using pieces of paper to show customers what the application will look like as they navigate through the system. (A slightly more high-tech version might use a PowerPoint slide show instead of pieces of paper.) If an application includes special hardware, such as a fingerprint scanner, you could tape a cardboard version onto a laptop to show customers what it will look like.

Sometimes, prototypes don't even have a user interface. For example, suppose you're writing a billing application that will process customer charges to generate invoices. You could write a prototype that fetches a particular customer's data, calculates outstanding charges, and prints an invoice. That would let programmers study how the data processing code works before they try to do the same thing for the entire customer database.

HORIZONTAL AND VERTICAL PROTOTYPES

A *horizontal prototype* is one that demonstrates a lot of the application's features but with little depth. For example, the prototype described earlier that lets a user pretend to navigate through customer orders is a horizontal prototype. Horizontal prototypes are often user interface prototypes that let customers see what the finished application will look like.

In contrast, a *vertical prototype* is one that has little breadth but great depth. The example described earlier that has no user interface and generates an invoice for a single customer is a vertical prototype.

The following sections explain some additional details about prototypes.

Types of Prototypes

There are several ways you can use a prototype. Chapter 4 mentioned two important types of prototypes: throwaway prototypes and evolutionary prototypes. In a throwaway prototype, you use the prototype to study some aspect of the system and then you throw it away and write code from scratch.

In an evolutionary prototype, the prototype demonstrates some of the application's features. As the project progresses, you refine those features and add new ones until the prototype morphs into the finished application.

A third kind of prototyping is incremental prototyping. In *incremental prototyping*, you build a collection of prototypes of that separately demonstrate the finished application's features. You then combine the prototypes (or at least their code) to build the finished application. As is the case with an evolutionary prototype, the prototype code becomes part of the final application, so you need to use good programming techniques for all the prototypes. That means coding is slower than it is with a throwaway prototype. Because the pieces of the system are built from separate prototypes, it may be easier for different programmers or teams to work on the pieces at the same time. That may let you finish the combined application sooner—although, you do need to allow time to integrate the pieces.

Pros and Cons

The following list summarizes prototyping's main benefits:

- **Improved requirements—**Prototypes allow customers to see what the finished application will look like. That lets them provide feedback to modify the requirements early in the project. Often customers can spot problems and request changes earlier so the finished result is more useful to users.
- **Common vision—**Prototypes let the customers and developers see the same preview of the finished application, so they are more likely to have a common vision of what the application should do and what it should look like.
- **Better design—**Prototypes (particularly vertical prototypes) let the developers quickly explore specific pieces of the application to learn what they involve. Prototypes also let developers test different approaches to see which one is best. The developers can use what they learn from the prototypes to improve the design and make the final code more elegant and robust.

Programming, like the rest of life, follows the rule of TANSTAAFL: There ain't no such thing as a free lunch. Prototyping comes with some disadvantages to go with its advantages:

- **Narrowing vision—**People tend to focus on a prototype's specific approach rather than on the problem it addresses. When you show customers (and developers) a prototype, they'll be less likely to think about other solutions that might do a better job.

  To avoid this problem, either don't build a prototype until you've considered possible alternatives, or build several prototypes to choose from.
- **Customer impatience—**A good prototype can make customers think that the finished application is just around the corner. They'll say things like, "The prototype looks good. Can't you just add a little error handling and a few extra features and call it done?"

  To avoid this, make sure customers realize that the prototype isn't anywhere close to the finished application. It's like a realistically painted cruise ship made out of papier-mâché. It may look ready to set sail, but you wouldn't want to put it in the ocean and climb aboard.

- **Schedule pressure—**This goes with the preceding issue. If customers see a prototype that they think is mostly done, they may not understand that you need another year to finish and may pressure you to shorten the schedule.

  To avoid this problem, the project manager, executive champion, and other management types need to manage customer expectations so that they know what will be ready and when.

- **Raised expectations—**Sometimes, a prototype may demonstrate features that won't be included in the application. For example, those features might turn out to be too hard. Sometimes, features are included to assess their value to users, and the features are dropped if they don't have enough benefit. Other times a feature may be present just to show a possible future direction. In those cases, users may be disappointed when their favorite features are missing from the finished application. This can be a particular problem with projects that release a series of versions of the application and someone's pet feature isn't included in release 1.0.

  To avoid this, make sure customers understand which features will be included and when.

- **Attachment to code—**Sometimes, developers become attached to the prototype's code. That can make them follow the methods used by that code (or even reuse the code wholesale) even if a better design exists. This can be a particularly bad problem with throwaway prototypes where the initial code might have low quality.

  To avoid this, make sure developers understand that the code should change if a better design is available. Hold design reviews and code reviews to make sure no one is stuck following a prototype approach if there's a better alternative.

- **Never-ending prototypes—**Throwaway prototypes are supposed to be built relatively quickly to provide fast feedback. Sometimes, developers spend far too much time refining a prototype to make it look better and include more features that aren't actually necessary.

  To avoid this, make sure the prototype doesn't include any more than is necessary to give customers a feel for how the program will work. Don't waste time making the prototype more flexible than necessary. Do the least amount of work you can get away with. For example, a prototype rarely needs to use a database. Usually you can just hard-wire data into the program to get a feel for how the final program will look. If customers decide they need to see more, they can say so.

Prototypes are great for helping you decide on the direction you should take. They can help define the user interface and other features, make sure customers and developers are on the same page, and let developers explore different solutions.

SPIRAL

The spiral model (not to be confused with a "death spiral" or "circling the drain") was first described in 1986 by Barry Boehm. He describes it as a "process model generator." It uses a risk-driven approach to help project teams decide on what development approach to take for various parts of the project. For example, if you don't understand all the requirements, then you might use an iterative approach for developing them.

The general spiral approach shown in Figure 13-3 consists of four basic phases.

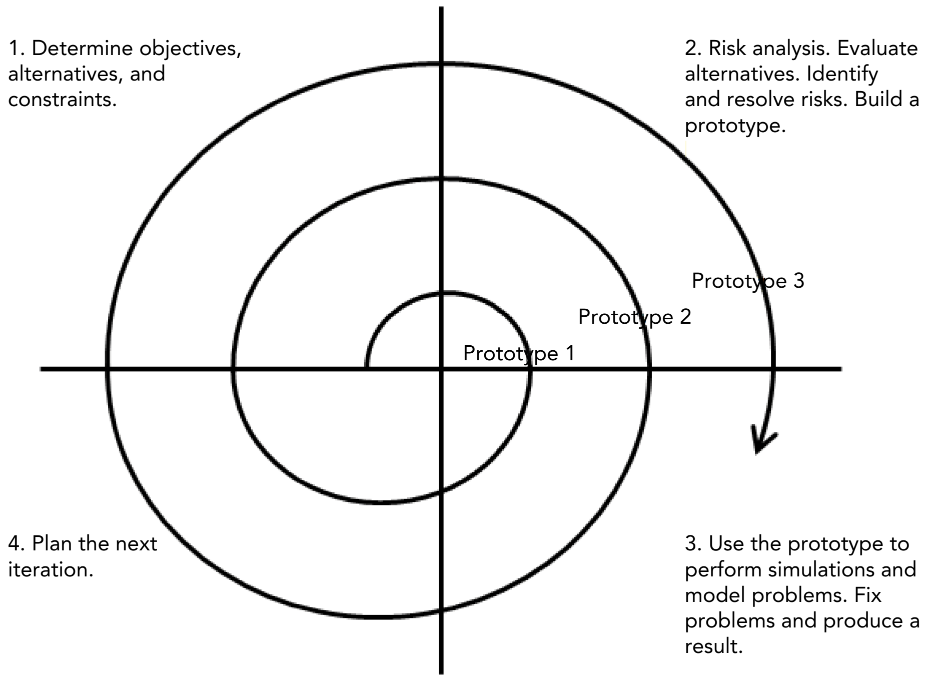
1. Determine objectives, alternatives, and constraints.
2. Risk analysis. Evaluate alternatives. Identify and resolve risks. Build a prototype.
Prototype 3
Prototype 2
Prototype 1
4. Plan the next iteration.
3. Use the prototype to perform simulations and model problems. Fix problems and produce a result.

FIGURE 13-3: The spiral process uses four phases.

In the first phase (which some call the planning phase), you determine the objectives of the current cycle. You define any alternatives and constraints on the objectives.

In the second phase (which some call the risk analysis phase), you perform a risk analysis to determine what the biggest risk factors are that could prevent you from achieving this cycle's objectives. You resolve the risks and build a prototype to achieve your objectives. (Note that this may not be a program. For example, if the goal of the current cycle is to build requirements, then this will be a set of prototype requirements.)

In the third phase (which some call the engineering phase), you use the prototype you just built to evaluate your solution. You perform simulations and model specific problems to see if you're on the right track. (For example, you might run through a bunch of operational scenarios to see if your prototype requirements can handle them.) You use what you learn to achieve the original objectives. After this phase, you should have something concrete to show for your efforts.

In the fourth phase (which some call the evaluation phase), you evaluate your progress so far and make sure the project's major stakeholders agree that the solution you came up with is correct and that the project should continue. If they decide you've made a mistake, you run another lap around the spiral to fix whatever problems remain. (You identify the missed objectives, evaluate alternatives, identify and resolve risks, and produce another prototype.) After you're sure you're on the right track, you plan the next trip around the spiral.

For a concrete example, consider Figure 13-4. The first trip around the spiral builds the project requirements. The team examines alternatives, identifies the largest risks (perhaps the customers' performance requirements are unclear), resolves the risks, and builds a prototype set of requirements. Team members then use the prototype to analyze the requirements and verify that they are correct. At that point, the verified requirements become the actual requirements.

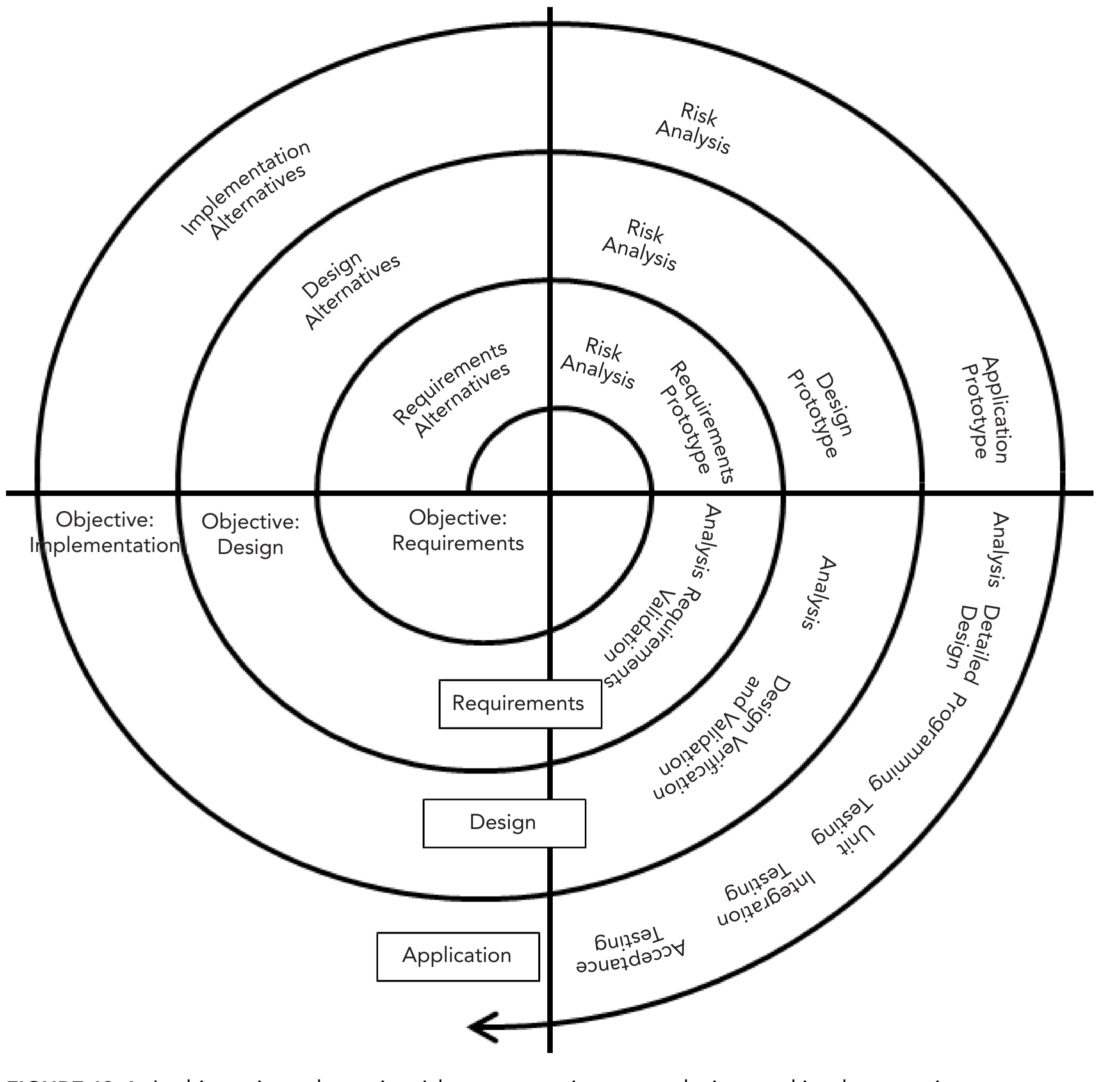

FIGURE 13-4: In this project, the major risks were requirements, design, and implementation.

The next trip around the spiral builds the application design. The team evaluates design alternatives, identifies and resolves the major risks, and builds a prototype design. Team members analyze the design and verify that it makes sense. The prototype design then becomes the design.

The final trip around the spiral drives the application's implementation. The team evaluates implementation alternatives (although they're probably used to a particular development approach already). They identify risks (perhaps previous projects have had maintenance issues) and resolve them (they decide to have more code reviews). The team then builds an operational prototype that shows how the program will work. They use the prototype to verify that everything is on track, and then they actually build the application. In this cycle, the implementation steps are broken down into detailed design, programming, unit testing, integration testing, and acceptance testing.

This example doesn't include every cycle that you might want to perform. For example, you might want to make separate user interface design or database design cycles.

Clarifications

Since he initially described the spiral approach, Boehm has made several clarifications mostly to correct mistakes people made interpreting the method. Those clarifications include the following:

- This is not simply a series of waterfall models drawn in a spiral and executed one after another to form an incremental approach. In fact, you could use multiple spirals to build different versions of an application.
- The activities need not follow a single spiral sequence. For example, you could spin the user interface design and database design off into completely separate spirals that both feed into an overall project cycle.
- You can add items, remove items, or perform items in different orders in a specific spiral model as needed. The activities you need to perform depend on the project.

Boehm further defined six characteristics that spiral development cycles should follow:

1. Define tasks concurrently. There's no need to perform everything sequentially.
2. Perform the following four tasks in each cycle. (Basically they are goals of the four phases.)
    a. Consider the goals of all stakeholders.
    b. Identify and evaluate alternative approaches for satisfying the stakeholders' goals.
    c. Identify and resolve risks in the selected approach.
    d. Make sure the stakeholders agree that the results of the current cycle are correct. Get the stakeholders' approval to continue the project into the next cycle.
3. Use risk to determine the level of effort. For example, perform enough code reviews to minimize the risk of buggy code, but don't perform so many reviews that you risk finishing late.
4. Use risk to determine the level of detail. For example, put enough work into the requirements to minimize the risk of the application not satisfying the customers, but don't over-specify requirements to the point where they restrict the developers' flexibility.
5. Use anchor milestones. Boehm later added the following anchor milestones to track the project's progress.

a. **Life Cycle Objectives** (LCO)—When the stakeholders agree that the project's technical and management approach is defined enough to satisfy all the stakeholders' goals, then it has reached its LCO milestone.

b. **Life Cycle Architecture** (LCA)—When the stakeholders agree that the project's approach can satisfy the goals and all significant risks have been eliminated or mitigated, then it has reached its LCA milestone.

c. **Initial Operational Capability** (IOC)—When there has been sufficient preparation to satisfy everyone's goals, then the project has reached its IOC milestone and should be installed. The preparations include everything needed to make the project a success. For example, the application is ready and tested, the user site is set up, the users have been trained, and the maintenance programmers are ready to take over.

6. Focus on the system and its life cycle rather than on short-term issues such as writing an initial design or writing code. This is intended to help you focus on the big picture.

Pros and Cons

The spiral approach is considered one of the most useful and flexible development approaches. The following list summarizes some of its main advantages:

- Its spiral structure gives stakeholders a lot of points for review and making "go" or "no-go" decisions.
- It emphasizes risk analysis. If you identify and resolve risks correctly, it should lead to eventual success.
- It can accommodate change reasonably well. Simply make any necessary changes and then run through a cycle to identify and resolve any risks they create.
- Estimates such as time and effort required become more accurate over time as cycles are finished and risks are removed from the project.

The following list summarizes some of the spiral approach's biggest disadvantages:

- It's complicated.
- Because it's complicated, it often requires more resources than simpler approaches.
- Risk analysis can be difficult.
- The complication isn't always worth the effort, particularly for low-risk projects.
- Stakeholders must have the time and skills needed to review the project periodically to make sure each cycle is completed satisfactorily.
- Time and effort estimates become more accurate as cycles are finished, but initially those estimates may not be good.
- It doesn't work well with small projects. You could end up spending more time on risk analysis than you'd need to build the entire application with a simpler approach.

For those reasons, the spiral approach is most useful with large high-risk projects and projects with uncertain or changeable requirements.

UNIFIED PROCESS

Despite its name, the *Unified Process* (*UP*) isn't actually a process. Instead it's an iterative and incremental development framework that you can customize to fit your business and projects.

The Unified Process approach is divided into the following four phases:

- **Inception—**During this phase you come up with the project's idea. (Or as in the movie, someone else comes up with the project's idea and makes you think it's yours.) This should be a short phase where you provide a business case, identify risks, provide an initial schedule, and sketch out the project's general goals. It should not include detailed requirements that might restrict the developers.
- **Elaboration—**During this phase you create the project requirements. You build use cases, architectural diagrams, and class hierarchies. You need to specify the system, but you still don't want to restrict developers with unnecessarily detailed requirements. The main goals are to identify and address risks so that the project doesn't fail later. Normally, this phase is divided into several iterations with the first addressing the most important risks.
- **Construction—**During this phase you write, test, and debug the code. This phase is divided into several iterations, each of which ends with a tested, high-quality working executable program that you can release to the users. The iterations implement the most important features first.
- **Transition—**During this phase you transfer the project to customers and the long-term maintenance team. Based on feedback from users, you might make changes and refinements and then release a new version, so this phase can include several iterations. This phase includes all the usual transitioning tasks such as staging, building the user environment (computers, networks, coffee machines, and so forth), user documentation, and user training.

You can add more phases if you like. For example, you might add the following two phases to model the application's life cycle after transition:

- **Production—**During this phase users use the application. The normal Unified Process assumes that the development team doesn't continue producing new versions of the application during this phase.
- **Disposal—**During this phase you remove the application and move users to a replacement system. If you're building the replacement, then this phase overlaps with the new project's transition phase.

Figure 13-5 shows the relative sizes of the Unified Process phases. A rectangle's height represents the resources (mostly the number of people in the development team) required for that phase. A rectangle's width represents the amount of time spent on that phase.

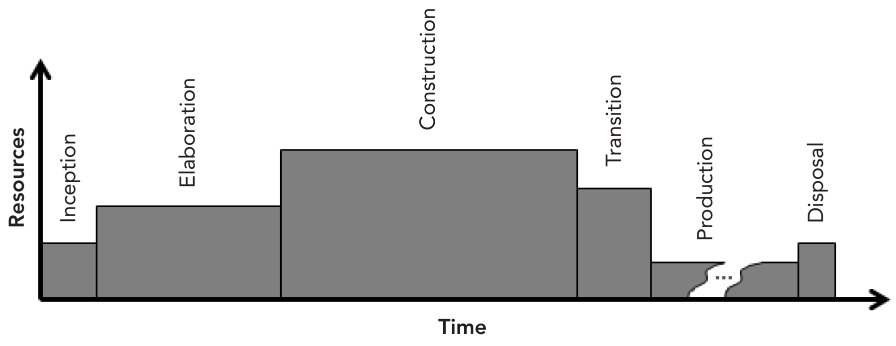

FIGURE 13-5: In the Unified Process, construction takes more time and effort than the other phases.

Figure 13-6 shows the relative amounts of different kinds of work during the project's phases and the iterations within those phases. For example, implementation work (programming) is relatively light during inception and elaboration, picks up during the construction iterations, and then tapers off during transition.

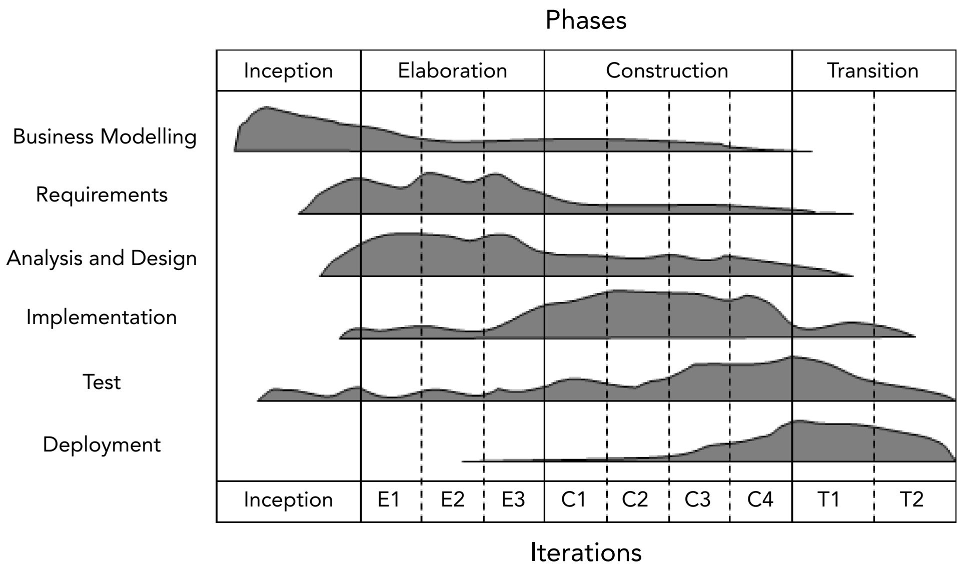

FIGURE 13-6: In the Unified Process, the amounts of different kinds of work grow and shrink during different project phases.

The project shown in Figure 13-6 had three elaboration iterations, four construction iterations, and two transition phases.

Pros and Cons

The following list summarizes some of the main advantages of the Unified Process approach:

➤ The iterative approach to the elaboration, construction, and transition phases enables you to incrementally define the requirements and assemble the application.

- The elaboration iterations focus on risks and risk mitigation to improve the project's chance of success.
- It can accommodate different development models flexibly. For example, you could use a series of waterfalls or an agile approach to the construction phase.
- The inception and elaboration phases generate a lot of documentation that can help new developers join the team later.
- It can enable incremental releases if wanted.

Some of the Unified Process approach's disadvantages are similar to those of the spiral approach. The following list summarizes some of the biggest Unified Process disadvantages:

- It's complicated (although not quite as confusing as the spiral approach).
- Because it's complicated, it often requires more resources than simpler approaches.
- Risk analysis can be difficult.
- The complication isn't always worth the effort, particularly for low-risk projects.
- It doesn't work well with small projects. You could end up spending more time on inception and elaboration than you'd need to build the entire application with a simpler approach.

Like the spiral approach, the Unified Process approach is most useful with large high-risk projects and projects with uncertain or changeable requirements.

Rational Unified Process

The *Rational Unified Process* (*RUP*) is IBM's version of the Unified Process. It uses the same four basic phases defined by UP: inception, elaboration, construction, and transition.

It also uses the same standard engineering disciplines (on the left in Figure 13-6): business modeling, requirements, analysis and design, implementation, test, and deployment. It also adds three "supporting disciplines": configuration and change management, project management, and environment. (Environment refers to customizing the process for the development organization and the current project.)

The main advantages to RUP over UP are the tools provided by IBM that make using the process easier. Those tools include artifact templates, document production and sharing, change request tracking, visual modeling, performance profiling, and more.

ARTIFACTUALLY

In RUP an *artifact* is a final or intermediate result that is produced by the project. The RUP includes documents (such as design documents and deployment plans), models (such as use cases and design models), and model elements (pieces of models such as classes or subsystems).

As you might expect, the advantages and disadvantages of RUP are similar to those for UP.

There are several variations on UP and RUP. For example, the *Open Unified Process* (*OpenUP*) is a tool that makes using the Unified Process easier. To make OpenUP more accessible to its target audience (smallish projects with 3–6 team members working on projects lasting 3–6 months), it omits most of the optional features provided by RUP, so it's easier to use.

OpenUP is part of the open source Eclipse Process Framework. For more information on OpenUP, see `epf.eclipse.org/wikis/openup`. For more information on the Eclipse Process Framework, see `www.eclipse.org/epf`.

The *Agile Unified Process* (*AUP*) is another simplified version of RUP. It brings agile methods such as test-driven development and agile modeling to UP. In 2012, AUP was superseded by *Disciplined Agile Delivery* (*DAD*, not to be confused with your father).

DAD has a structure that's somewhat similar to UP. In particular, it has the three phases: inception, construction, and transition. (Elaboration is divided between inception and construction.) DAD also borrows many techniques from different agile development approaches such as Scrum, Extreme Programming, Kanban, and others. I won't say any more about DAD until the next chapter, after you've learned more about those agile approaches.

CLEANROOM

The Cleanroom model emphasizes defect prevention rather than defect removal. The idea is to build the application in steps that are carefully monitored and tested to prevent anything bad from entering into the application. (The name is inspired by the way a manufacturing clean room prevents dust and other gunk from getting into the manufacturing process.)

The following list summarizes Cleanroom's basic principles:

➤ **Formal methods—**Code is produced using formal mathematical methods that help ensure that the code satisfies the design models. Code reviews also help verify that the code correctly implements the required behavior.

➤ **Statistical quality control—**Code is produced incrementally. Each increment's quality is measured to ensure that the project is making acceptable progress.

➤ **Statistical testing—**Testing uses statistical experiments to estimate the application quality. (This requires some serious statistical analysis so that you can estimate not only the application's quality but so that you can also calculate a level of confidence for that estimate.)

Unfortunately, the mathematics needed to do the statistical analysis is quite intimidating. I like math more than most people (my C# Helper website `www.csharphelper.com` is littered with mathematical examples for C# developers) but even I hesitate when faced with statistical testing models. Some of them are seriously complicated and confusing.

Still, the basic ideas behind Cleanroom are excellent. First, if you don't let bad code sneak into the application, then you won't need to fix it later.

Second, if you evaluate the quality of the application after every iteration, you can track how effective your development effort is. You can also fine-tune development if the number of defects increases in a particular iteration.

Even if you don't have the tools or expertise to follow the Cleanroom process in every detail, it's worth borrowing those two principles.

SUMMARY

Iterated development is one technique for trying to keep a software engineering project on track. It lets you periodically review your progress to ensure that the application is heading toward a result that satisfies the requirements. It also lets you refine and correct the requirements over time if necessary.

Prototyping lets you study pieces of an application so that you can make adjustments. Models such as spiral and Unified Process (and its variants) use iteration to help the development and requirements eventually meet. Some of those models also place an emphasis on risk management to reduce the chances of the project failing.

In addition to keeping a project heading in the right direction, iterative and incremental methods allow you to release partial implementations of the application if they are useful. The next chapter describes other techniques that let you give the users partial functionality as soon as possible.

EXERCISES

1. Suppose your customer wants an application with 10 features and insists that the application is completely useless unless all 10 are implemented with full fidelity. Would there be any benefit to iterative, incremental, or agile approaches?
2. Explain why a throwaway prototype inherently uses an agile approach.
3. How does an incremental prototype differ from an incremental project? What would you need to do to use an incremental prototype in an incremental project?
4. Can you use an evolutionary prototype in a predictive project (such as waterfall)?
5. Can you use an incremental prototype in a predictive project?
6. Look at the project shown in Figure 13-6. Why might the deployment tasks start during the elaboration phase instead of at the beginning of the transition phase? What deployment tasks might you be performing during elaboration? What tasks might you be performing during construction?
7. Look at the project shown in Figure 13-6. The testing tasks begin in the inception phase before the implementation tasks start. What are you testing during inception if there isn't any code yet?
8. Look at the project shown in Figure 13-6. What kinds of code are the team members writing during the elaboration phase? What kinds of tests are they performing during that phase?
9. Add a new row to the bottom of Figure 13-6 that shows the amount of customer interaction required during different phases of development for an in-house project.

10. Draw a diagram showing how the phases of the waterfall model match up with those of Unified Process. What are the main differences?

11. Indicate whether the following items describe the predictive, iterative, incremental, or agile approaches.

    a. Features are released as soon as they are useful. Over time, existing features are improved and new features are added.

    b. Features are released one at a time with full fidelity.

    c. All the application's features are released at the same time with full fidelity.

    d. Every feature is released quickly with low fidelity and then improved over time.

12. Which approach (predictive, iterative, incremental, or agile) gets a working program to users the soonest? Latest? What can you say about the timing of the other two approaches?

13. Suppose you're a real estate developer building a neighborhood containing 100 houses. How would each of the predictive, iterative, incremental, and agile approaches correspond to home sales? Assume the "features" of the project are the houses and "releasing a feature" means allowing people to move into a home. Which of the approaches could work? Which approach do developers actually use?

14. Suppose you're a different real estate developer who specializes in more interesting projects. This time you're building an amusement park. How would each of the predictive, iterative, incremental, and agile approaches correspond to opening the park? The "features" of the project are the rides, snack shops, and games of "chance" (which actually leave little to chance). "Releasing a feature" means allowing people to ride on a ride, buy greasy food at high prices, or use darts to try to pop balloons that seem to be made of Kevlar. Which of the approaches could work?

▶ WHAT YOU LEARNED IN THIS CHAPTER

- Predictive approaches make one big release when everything is done.
- Iterative approaches release every feature with low fidelity and then improve fidelity over time.
- Incremental approaches release features as they are finished with high fidelity.
- Agile approaches combine iterative and incremental approaches. They release features when they are usable. Over time they improve existing features and add new ones.
- A prototype is a simplified model that lets you study the behavior of some part of an application.
- Horizontal prototypes have breadth but little depth. They are typically used to study user interfaces and show customers what the application will look like.
- Vertical prototypes have little breadth and great depth. They are typically used to study architecture and programming issues.
- You don't reuse the code in a throwaway prototype.
- Over time an evolutionary prototype is refined and improved until it becomes the finished application.
- In incremental prototyping, you build separate prototypes of the application's features and then combine them to form the finished application.
- The spiral model uses a sequence of repeating phases (planning, risk analysis, engineering, and evaluation) to identify and neutralize project risks.
- Unified Process is an iterative and incremental approach that uses the phases' inception, elaboration, construction, and transition. The last three phases are iterative.
- Rational Unified Process is IBM's version of Unified Process. Other versions include OpenUP, Agile Unified Process, and Disciplined Agile Delivery.
- Cleanroom uses formal methods, statistical quality control, and statistical testing to prevent defects from entering an application's code. Its two principles, "Don't let bad code into the application," and "Evaluate the quality of the application after each iteration," are worth using in any development approach.
- You shouldn't ride roller coasters that don't have full fidelity.