7

Lifecycle Planning

Contents

Related Topics

Evolutionary delivery: Chapter 20
Evolutionary prototyping: Chapter 21
Staged delivery: Chapter 36
Summary of spiral lifecycle model: Chapter 35
Summary of lifecycle model selection: Chapter 25

EVERY SOFTWARE-DEVELOPMENT EFFORT goes through a "lifecycle," which consists of all the activities between the time that version 1.0 of a system begins life as a gleam in someone's eye and the time that version 6.74b finally takes its last breath on the last customer's machine. A lifecycle model is a prescriptive model of what should happen between first glimmer and last breath.

For our purposes, the main function of a lifecycle model is to establish the order in which a project specifies, prototypes, designs, implements, reviews, tests, and performs its other activities. It establishes the criteria that you use to determine whether to proceed from one task to the next. This chapter

focuses on a limited part of the full lifecycle, *the period between the first glimmer and initial release.* You can direct this focus either to new product development or to maintenance updates of existing software.

The most familiar lifecycle model is the well-known waterfall lifecycle model, which has some equally well-known weaknesses. Other lifecycle models are available, and in many cases they are better choices for rapid development than the waterfall model is. (The waterfall model is described in the next section, "Pure Waterfall.")

By defining the master plan for the project, the lifecycle model you choose has as much influence over your project's success as any other planning decision you make. The appropriate lifecycle model can streamline your project and help ensure that each step moves you closer to your goal. Depending on the lifecycle model you choose, you can improve development speed, improve quality, improve project tracking and control, minimize overhead, minimize risk exposure, or improve client relations. The wrong lifecycle model can be a constant source of slow work, repeated work, unnecessary work, and frustration. Not choosing a lifecycle model can produce the same effects.

Many lifecycle models are available. In the next several sections, I'll describe the models; and in the final section, I'll describe how to pick the one that will work best for your project.

Case Study 7-1. Ineffective Lifecycle Model Selection

The field agents at Giga-Safe were clamoring for an update to Giga-Quote 1.0, both to correct defects and to fix some annoying user-interface glitches. Bill had been reinstated as the project manager for Giga-Quote 1.1 after being removed at the end of Giga-Quote 1.0, and he brought in Randy for some advice. Randy was a high-priced consultant he had met at a sports bar.

"Here's what you should do," Randy said. "You had a lot of schedule problems last time, so this time you need to organize your project for all-out development speed. Prototyping is the fastest approach, so have your team use that." Bill thought that sounded good, so when he met with the team later that day, he told them to use prototyping.

Mike was the technical lead on the project, and he was surprised. "Bill, I don't follow your reasoning," he said. "We've got 6 weeks to fix a bunch of bugs and make some minor changes to the UI. What do you want a prototype for?"

"We need a prototype to speed up the project," Bill said testily. "Prototyping is the newest, fastest approach, and that's what I want you to use. Is there some kind of problem with that?"

(continued)

Case Study 7-1. Ineffective Lifecycie Model Selection, *continued*

Touchy subject, Mike thought. "OK," he said. "We'll develop a prototype, if that's what you want."

Mike and another developer, Sue, went to work on the prototype. Since it was almost identical to their current system, it took only a few days to mock up the whole system.

At the beginning of week 2, they showed the prototype to the field-agents' manager, A.J. "Hell, I can't tell my agents this is all they're getting!" A.J. exclaimed. "It hardly does any more than the program does now! My agents are wailing about how they need something *better*. I've got some ideas for some new reports. Here, I'll show you." Mike and Sue listened patiently, and after the meeting Mike got together with Bill.

"We showed A.J. the prototype. He wants to add some new reports, and he won't take no for an answer. But we've got our hands full with the work we're already supposed to be doing."

"I don't see the problem," Bill said. "He's the field agents' manager. If he says they need some new reports, they need some new reports. You guys will just have to find a way to get them done in time."

"I'll try," Mike said. "But I have to tell you that there is only about a 1-percent chance that we'll finish on time if we add these reports."

"Well, you've got to do them," Bill said. "Maybe now that you're using prototyping, the work will go faster than you expect."

Two days later, A.J. stopped by Mike's cubicle. "I've been looking at that prototype, and I think we need to redesign some of the input screens too. I showed some of my agents your prototype yesterday at our monthly regional meeting. They said they'd call you with some more ideas. I gave them your phone number—I hope you don't mind. Keep up the good work!"

"Thanks," Mike said and slid down into his chair. Later, Mike asked Bill if he would try to talk A.J. out of the changes, but Bill said, "No."

The next day, Mike got phone calls from two agents who had been at the regional meeting. They both wanted more changes to the system. For the next couple of weeks, he got calls every day, and the list of changes piled up.

At the 4-week mark of their 6-week project, Mike and Sue estimated that they had received 6 months' worth of changes that they were supposed to have done in 2 weeks. Mike met with Bill again. "I'm disappointed in you," Bill said. "I've promised A.J. and those field agents that you would make the changes they requested. You're not giving the prototyping approach a chance. Just wait until it kicks in."

(continued)

Case Study 7-1. Ineffective Lifecycle Model Selection, *continued*

It already has kicked in, Mike thought. And I'm the one getting kicked. But Bill was resolute.

At the 8-week mark, Bill started to complain that Mike and Sue weren't working hard enough. At the 10-week mark, Bill began dropping by their cubicles twice a day to check their progress. By the time they hit the 12-week mark, the agents were complaining, and Bill said, "We've got to get something out. Just release what you have now." Since neither the new reports nor the new input screens were completed, Mike and Sue simply stubbed out the code under development and released a version that consisted mainly of bug fixes and corrections to annoying UI glitches—roughly what they had planned to develop and release in the first place, but it took 12 weeks instead of 6.

7.1 Pure Waterfall

The granddaddy of all lifecycle models is the waterfall model. Although it has many problems, it serves as the basis for other, more effective lifecycle models, so it's presented first in this chapter. In the waterfall model, a project progresses through an orderly sequence of steps from the initial software concept through system testing. The project holds a review at the end of each phase to determine whether it is ready to advance to the next phase—for example, from requirements analysis to architectural design. If the review determines that the project isn't ready to move to the next phase, it stays in the current phase until it is ready.

The waterfall model is document driven, which means that the main work products that are carried from phase to phase are documents. In the pure waterfall model, the phases are also discontinuous—they do not overlap. Figure 7-1 shows how the pure waterfall lifecycle model progresses.

The pure waterfall model performs well for product cycles in which you have a stable product definition and when you're working with well-understood technical methodologies. In such cases, the waterfall model helps you to find errors in the early, low-cost stages of a project. It provides the requirements stability that developers crave. If you're building a well-defined maintenance release of an existing product or porting an existing product to a new platform, a waterfall lifecycle might be the right choice for rapid development.

The pure waterfall model helps to minimize planning overhead because you can do all the planning up front. It doesn't provide tangible results in the form of software until the end of the lifecycle, but, to someone who is familiar with it, the documentation it generates provides meaningful indications of progress throughout the lifecycle.

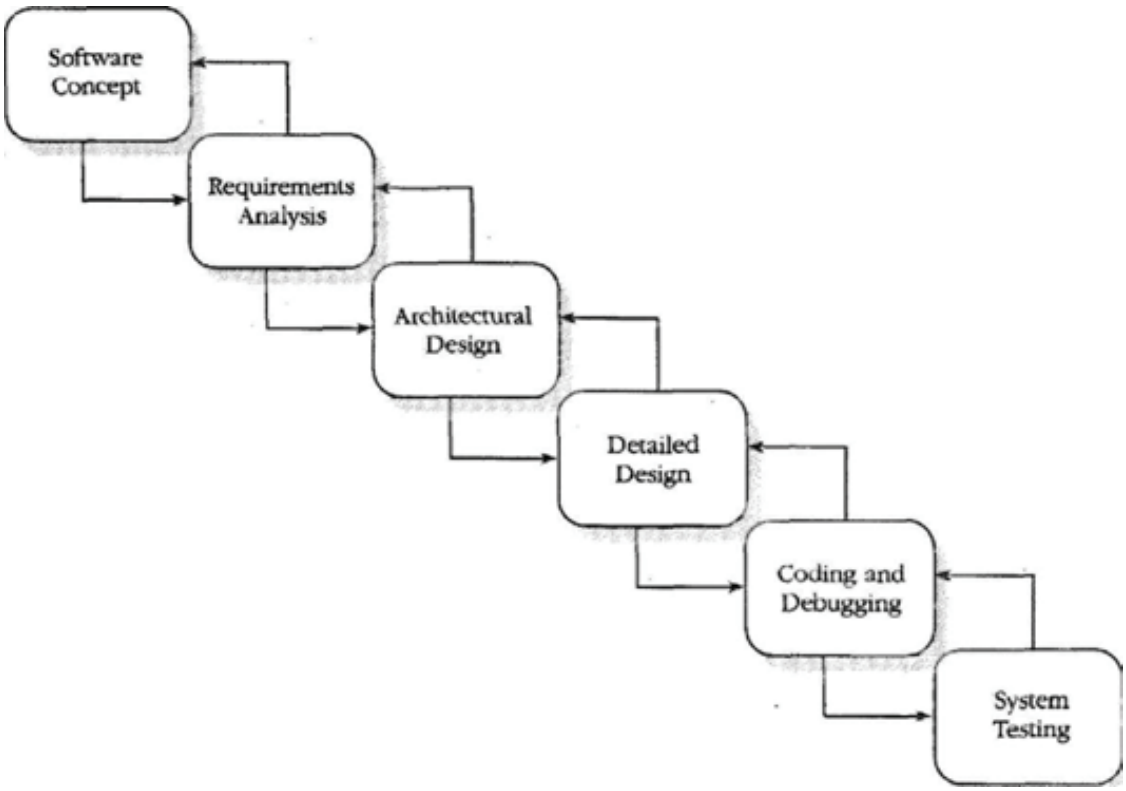


Figure 7-1. *The pure waterfall model. The waterfall model is the most well-known lifecycle model and provides good development speed in some circumstances. Other models, however, often provide greater development speed.*

The waterfall model works well for projects that are well understood but complex, because you can benefit from tackling complexity in an orderly way. It works well when quality requirements dominate cost and schedule requirements. Elimination of midstream changes eliminates a huge and common source of potential errors.

The waterfall model works especially well if you have a technically weak staff or an inexperienced staff because it provides the project with a structure that helps to minimize wasted effort.

The disadvantages of the pure waterfall model arise from the difficulty of fully* specifying requirements at the beginning of the project, before any design work has been done and before any code has been written.

CROSS-REFERENCE For details on problems with traditional specification methods, see "Problems with Traditional Specifications" in Section 14.1.

Developers complain about users who don't know what they want, but suppose the roles were reversed. Imagine trying to specify your car in detail to an automotive engineer. You tell the engineer that you want an engine, body, windows, steering wheel, accelerator pedal, brake pedal, emergency brake, seats, and so on. But can you remember to include everything that an automotive engineer will need to know to build your car?

Suppose you forget to specify that you need back-up lights that turn on when the car goes into reverse. The engineer goes away for 6 months and returns with a car with no back-up lights. You say, "Oh boy, I forgot to specify that the car needs back-up lights that turn on automatically when I shift into reverse."

The engineer goes ballistic. "Do you know what it's going to cost to take the car apart to connect wiring from the transmission to the rear of the car? We have to redesign the rear panel on the car, put in wiring for the brake lights, add another sensor to the transmission—this change will take weeks, if not months! Why didn't you tell me this in the first place?"

You grimace; it seemed like such a simple request...

Understandable mistake, right? A car is a complicated thing for an amateur to specify. A lot of software products are complicated too, and the people who are given the task of specifying software are often not computer experts. They can forget things that seem simple to them until they see the working product. If you're using a waterfall model, forgetting something can be a costly mistake. You don't find out until you get down to system testing that one of the requirements was missing or wrong.

Thus, the first major problem with the waterfall model is that it isn't flexible. You have to fully specify the requirements at the beginning of the project, which may be months or years before you have any working software. This flies in the face of modern business needs, in which the prize often goes to the developers who can implement the most functionality in the latest stage of the project. As Microsoft's Roger Sherman points out, the goal is often not to achieve what you said you would at the beginning of the project, but *to* achieve the maximum possible within the time and resources available (Sherman 1995).

Some people have criticized the waterfall model for not allowing you to back up to correct your mistakes. That's not quite right. As Figure 7-1 suggests, backing up is allowed, but it's difficult. A different view of the waterfall model that might put the matter into better perspective is the salmon lifecycle model, shown in Figure 7-2.

You're allowed to swim upstream, but the effort might kill you! At the end of architectural design, you participated in several major events that declared you were done with that phase. You held a design review, and you signed the official copy of the architecture document. If you discover a flaw in the architecture during coding and debugging, it's awkward to swim upstream and retrofit the architecture.

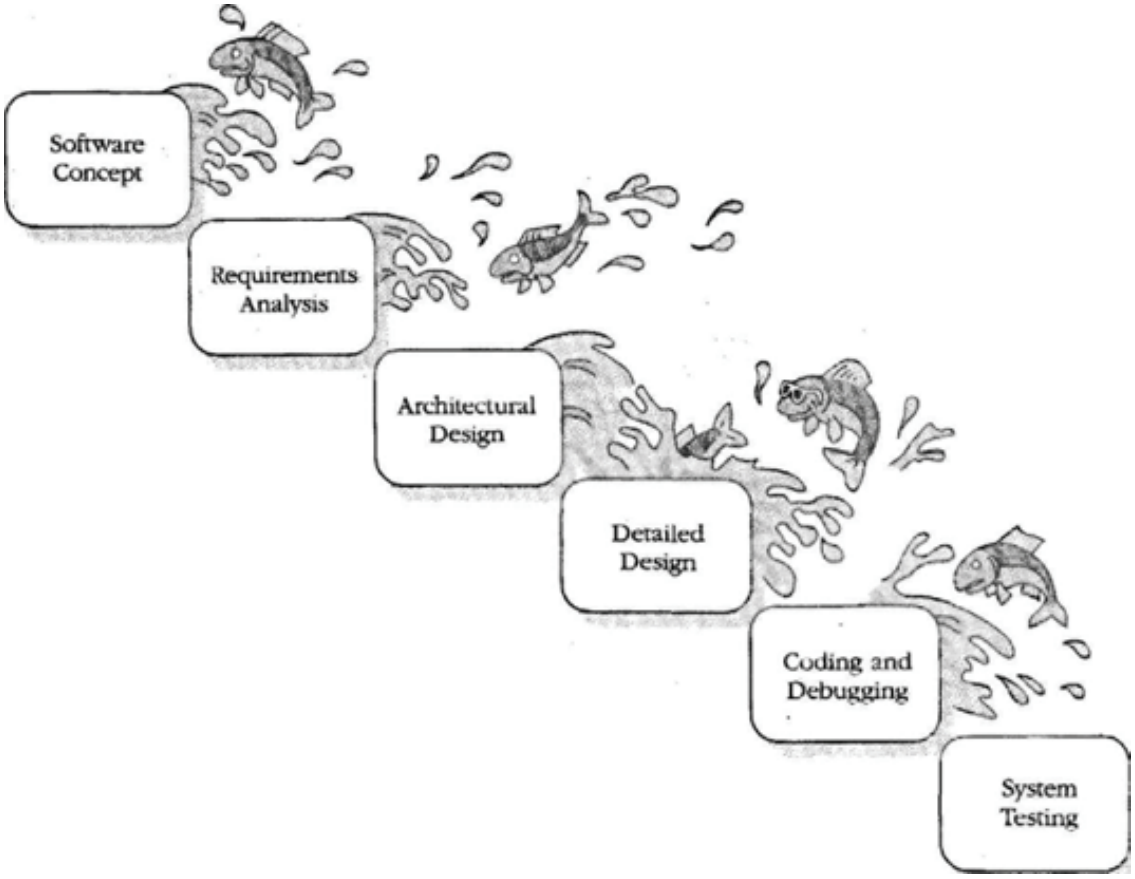


Figure 7-2. *Another depiction of the waterfall model-—the salmon lifecycle model, It inn 't impossible to back up using the waterfall model, just difficult.*

The waterfall lifecycle model has several other weaknesses. Some tools, methods, and activities span waterfall phases; those activities are difficult to accommodate in the waterfall model's disjoint phases. For a rapid-development project, the waterfall model can prescribe an excessive amount of documentation. If you're trying to retain flexibility, updating the specification can become a full-time job. The waterfall model generates few visible signs of progress until the very end. That can create the perception of slow development—even if it isn't true. Customers like to have tangible assurances that their projects will be delivered on time.

In summary, the venerable pure waterfall model's weaknesses often make it poorly suited for a rapid-development project. Even in the cases in which the pure waterfall model's strengths outweigh its weaknesses, modified waterfall models can work better.

7.2 Code-and-Fix

CROSS-REFERENCE
Code-and-fix is commonly combined with commitment-based development approaches. For details, see Section 2.5, "An Alternative Rapid-Development Strategy."

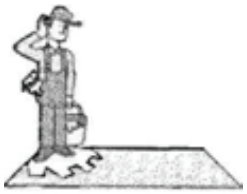
The code-and-fix model is a model that is seldom useful, but it is nonetheless common, so I'd like to discuss it. If you haven't explicitly chosen another lifecycle model, you're probably using code-and-fix by default. If you haven't done much project planning, you're undoubtedly using code-and-fix. Combined with a short schedule, code-and-fix gives rise to the code-like-hell approach described earlier.

When you use the code-and-fix model, you start with a general idea of what you want to build. You might have a formal specification, or you might not. You then use whatever combination of informal design, code, debug, and test methodologies suits you until you have a product that's ready to release. Figure 7-3 illustrates this process.

Figure 7-3. *The code-and-fix model. Code-and-fix is an informal model that's in common use because it's simple, not because it works well.*

The code-and-fix model has two advantages. First, it has no overhead: you don't spend any time on planning, documentation, quality assurance, standards enforcement, or any activities other than pure coding. Since you jump right into coding, you can show signs of progress immediately. Second, it requires little expertise: anyone who has ever written a computer program is familiar with the code-and-fix model. Anyone can use it.

For tiny projects that you intend to throw away shortly after they're built, this model can be useful—for small proof-ofcconcept programs, for short-lived demos, or throwaway prototypes.



CLASSIC MISTAKE

For any kind of project other than a tiny project, this model is dangerous. It might have no overhead, but it also provides no means of assessing progress; you just code until you're done. It provides no means of assessing quality or identifying risks. If you discover three-quarters of the way through coding that your whole design approach is fundamentally flawed, you have no choice but to throw out your work and start over. Other models would set you up to detect such a fundamental mistake earlier, when it would have been less costly to fix.

In the end, this lifecycle model has no place on a rapid-development project, except for the small supporting roles indicated.

7.3 Spiral

At the other end of the sophistication scale from the code-and-fix model is the spiral model. The spiral model is a risk-oriented lifecycle model that breaks a software project up into miniprojects. Each miniproject addresses one or more major risks until all the major risks have been addressed. The concept of "risk" is broadly defined in this context, and it can refer to poorly understood requirements, poorly understood architecture, potential performance problems, problems in the underlying technology, and so on. After the major risks have all been addressed, the spiral model terminates as a waterfall lifecycle model would. Figure 7-4 on the next page illustrates the spiral model, which some people refer to affectionately as "the cinnamon roll."

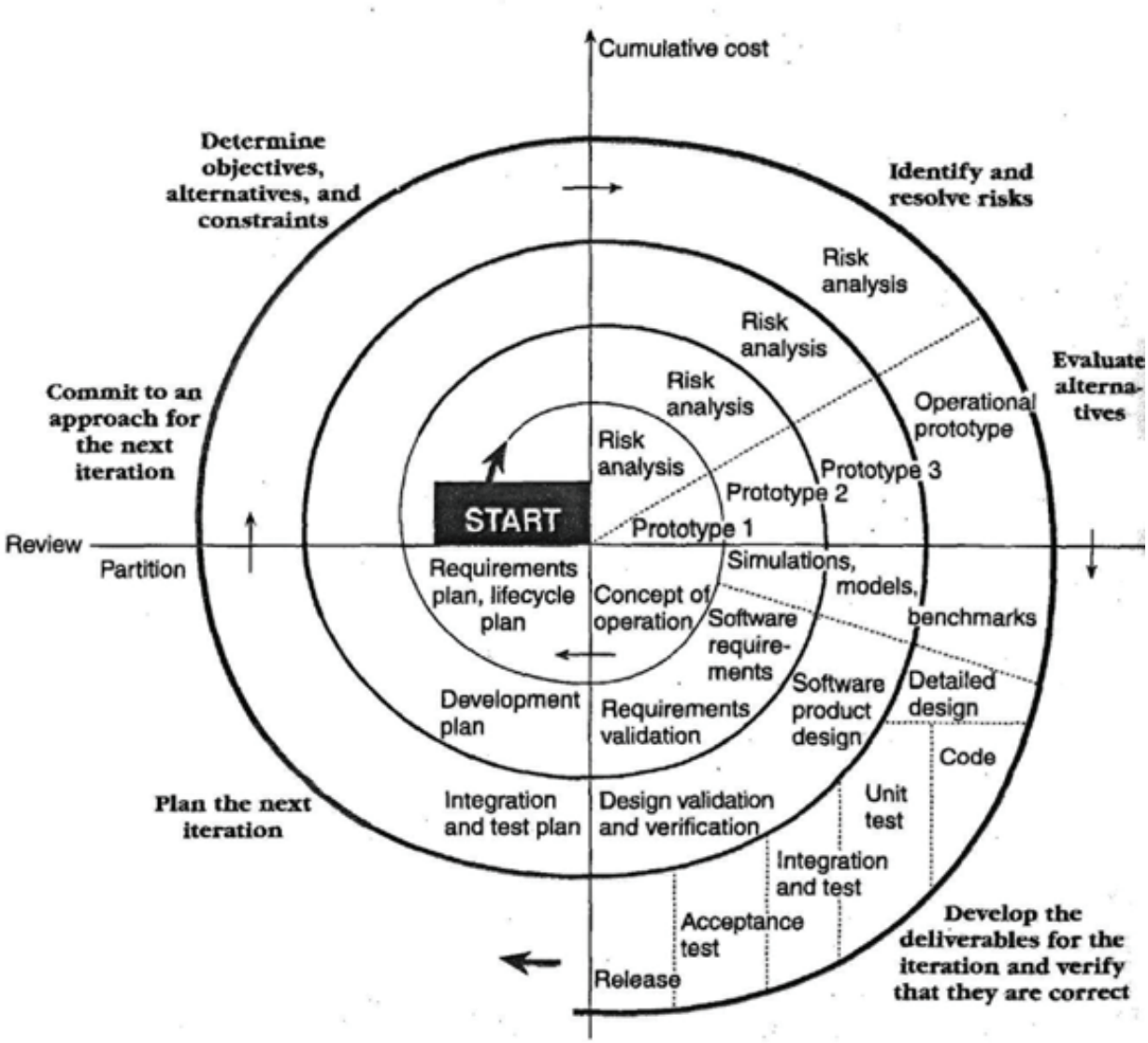
Figure 7-4 is a complicated diagram, and it is worth studying. The basic idea behind the diagram is that you start on a small scale in the middle of the spine, explore the risks, make a plan to handle the risks, and then commit to an approach for the next iteration. Each iteration moves your project to a larger scale. You roll up one layer of the cinnamon roll, check to be sure that it's what you wanted, and then you begin work on the next layer.

Each iteration involves the six steps shown in bold on die outer edges of the spiral:

1. Determine objectives, alternatives, and constraints
2. Identify and resolve risks
3. Evaluate alternatives
4. Develop the deliverables for that iteration, and verify that they are correct
5. Plan the next iteration
6. Commit to an approach for the next iteration (if you decide to have one)

In the spiral model, the early iterations are the cheapest. You spend less developing the concept of operation than you do developing the requirements, and less developing the requirements than you do developing the design, implementing the product, and testing it.

Don't take the diagram more literally than it's meant to be taken. It isn't important that you have exactly four loops around the spiral, and it isn't important that you perform the six steps exactly as indicated, although that's usually a good order to use. You can tailor each iteration of the spiral to suit the needs of your.project.



Source: Adapted from "A Spiral Model of Software Development and Enhancement" (Boehm 1988).

Figure 7-4. *The spiral model. In the spiral model, you start small and expand the scope of the project in increments. You expand the scope only after you've reduced the risks for the next increment to an acceptable level.*

CROSS-REFERENCE
For more on risk management, *see* Chapter 5, "Risk Management,"

You can combine the model with other lifecycle models in a couple different ways. You can begin your project with a series of risk-reduction iterations; after you've reduced risks to an acceptable level, you can conclude the development effort with a waterfall lifecycle or other non-risk-based lifecycle. You can incorporate other lifecycle models as iterations within the spiral model. For example, if one of your risks is that you're not sure that your

performance targets are achievable, you might include a prototyping iteration to investigate whether you can meet the targets.

One of the most important advantages of the spiral model is that as costs increase, risks decrease. The more time and money you spend, the less risk you're taking, which is exactly what you want on a rapid-development project.

The spiral model provides at least as much management control as the traditional waterfall model. You have the checkpoints at the end of each iteration. Because the model is risk oriented, it provides you with early indications of any insurmountable risks. If the project can't be done for technical or other reasons, you'll find out early—and it won't have cost you much.

The only disadvantage of the spiral model is that it's complicated. It requires conscientious, attentive, and knowledgeable management. It can be difficult to define objective, verifiable milestones that indicate whether you're ready to add the next layer to the cinnamon roll. In some cases, the product development is straightforward enough and project risks are modest enough that you don't need the flexibility and risk management provided by the spiral model.

7.4 Modified Waterfalls

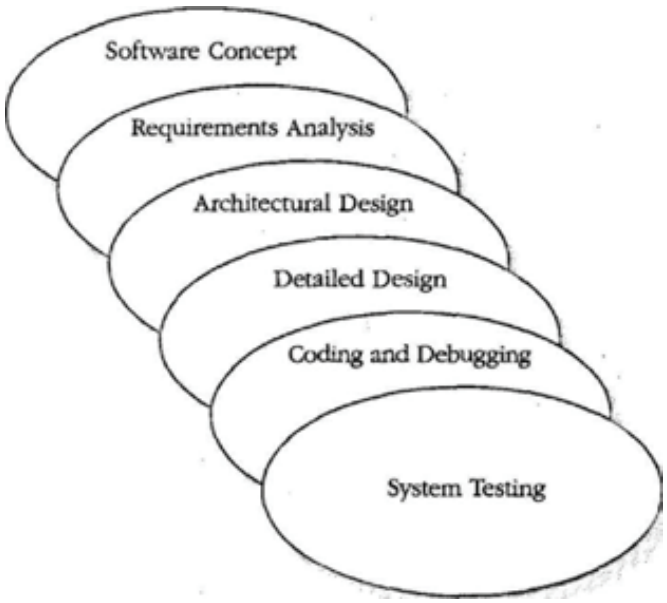
The activities identified in the pure waterfall model are intrinsic to software development. You can't avoid them. You have to come up with a software concept somehow, and you have to get requirements from somewhere. You don't have to use the waterfall lifecycle model to gather requirements, but you do have to use something. Likewise, you can't avoid having an architecture, design, or code.

Most of the weaknesses in the pure waterfall model arise not from problems with these activities but from the treatment of these activities as disjoint, sequential phases. You can, therefore, correct the major weaknesses in the pure waterfall model with relatively minor modifications. You can modify it so that the phases overlap. You can reduce the emphasis on documentation. You can allow for more regression.

Sashimi (Waterfall with Overlapping Phases)

Peter DeGrace describes one of the modifications to the waterfall model as the "sashimi model." The name comes from a Japanese hardware development model (from Fuji-Xerox) and refers to the Japanese style of presenting sliced raw fish, with the slices overlapping each other. (The fact that this

model has to do with fish does not mean that it's related to the salmon lifecycle model.) Figure 7-5 shows my version of what the sashimi model looks like for software.



Source: Adapted from *Wicked Problems, Righteous Solutions* (DeGrace and Stahl 1990).

Figure 7-5. *Tlje sashimi model. You can overcome some of the weaknesses of the waterfall model by overlapping its stages, but the approach creates new problems.*

The traditional waterfall model allows for minimal overlapping between phases at the end-of-phase review. This model suggests a stronger degree of overlap—for example, suggesting that you might be well into architectural design and perhaps partway into detailed design before you consider requirements analysis to be complete. I think this is a reasonable approach for many projects, which tend to gain important insights into what they're doing as they move through their development cycles and which function poorly with strictly sequential development plans.

In the pure waterfall model, the ideal documentation is documentation that one team can hand to a completely separate team between any two phases. The question is, "Why?" If you can provide personnel continuity between software concept, requirements analysis, architectural design, detailed design, and coding and debugging, you don't need as much documentation. You can follow a modified waterfall, model and substantially reduce the documentation needs.

The sashimi model is not without problems. Because there is overlap among phases, milestones are more ambiguous, and it's harder to track progress accurately. Performing activities in parallel can lead to miscommunication,

mistaken assumptions, and inefficiency. If you're working on a small, well-defined project, something close to the pure waterfall model can be the most efficient model available.

Waterfall with Subprojects

Another problem with the pure waterfall model from a rapid-development point of view is that you're supposed to be completely done with architectural design before you begin detailed design, and you're supposed to be completely done with detailed design before you begin coding and debugging. Systems do have some areas that contain design surprises, but they have other areas that we've implemented many times before and that contain no surprises. "Why delay the implementation of the areas that are easy to design just because we're waiting for the design of a difficult area? If the architecture has broken the system into logically independent subsystems, you can spin off separate projects, each of which can proceed at its own pace. Figure 7-6 shows a bird's-eye view of how that might look.

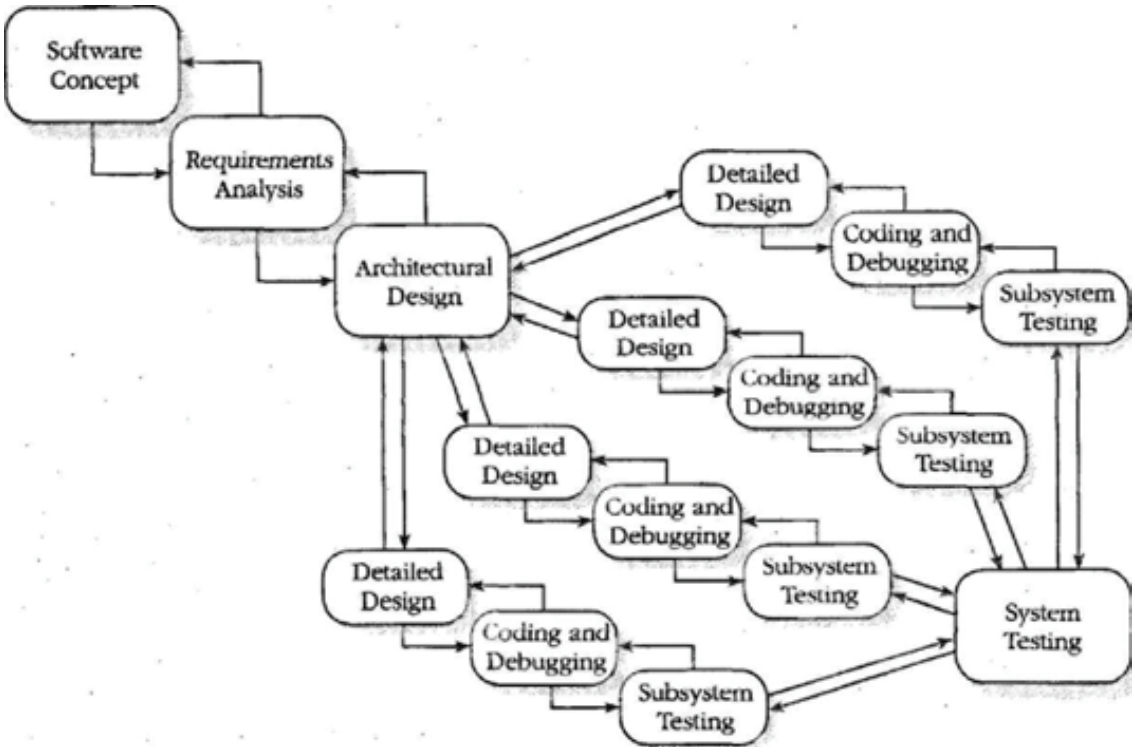


Figure 7-6. *Tlie waterfall model with subprojects. Careful planning can allow you to perform some of the waterfall's tasks in parallel,*

The main risk with this approach is unforeseen interdependences. You can partly account for that by eliminating dependencies at architecture time or waiting until after detailed-design time to break the project into subprojects.

Waterfall with Risk Reduction

Another of the waterfall model's weaknesses is that it requires you to fully define requirements before you begin architectural design, which seems reasonable except that it also requires you to fully understand the requirements before you begin architectural design. Modifying the waterfall—again only slightly—you can put a risk-reduction spiral at the top of the waterfall to address the requirements risk. You can develop a user-interface prototype use system storyboarding, conduct user interviews, videotape users interact ing with an older system, or use any other requirements-gathering practices that you think are appropriate.

Figure 7-7 shows the waterfall model with risk reduction. Requirements analysis and architectural design are shown in gray to indicate that they might be addressed during the risk-reduction phase rather than during the waicrfall phase.

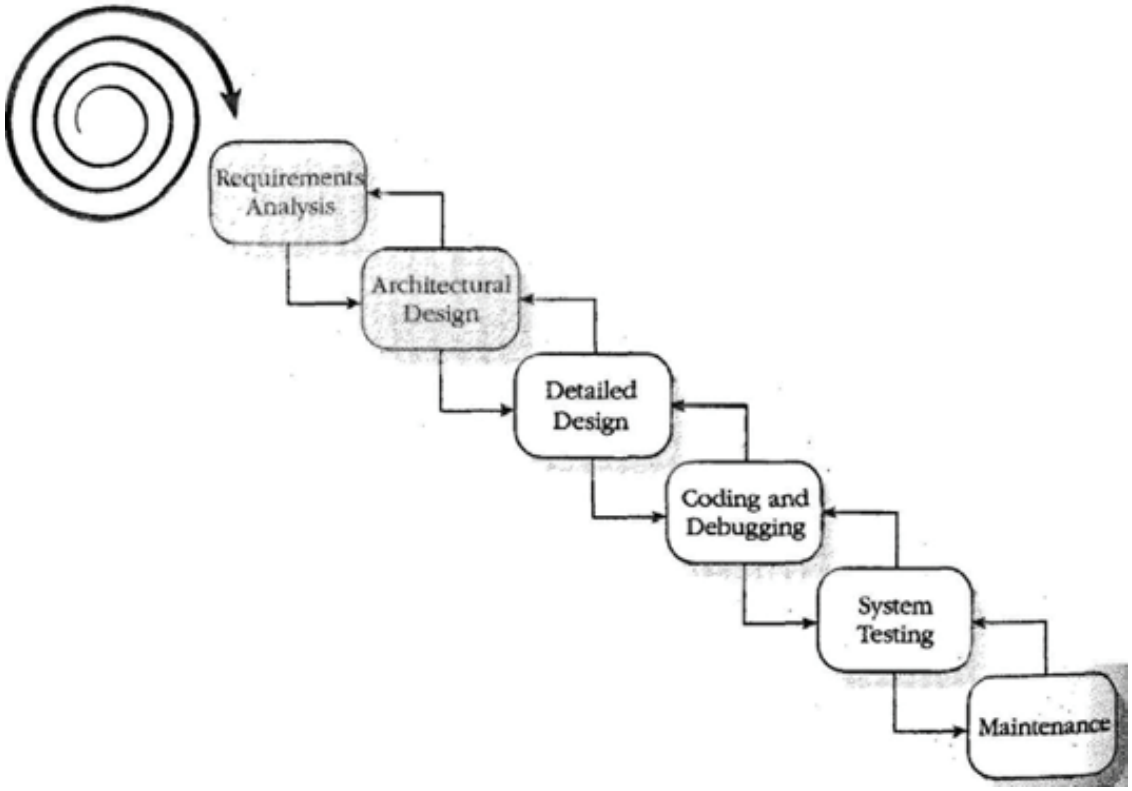


Figure 7-7. *Risk-reduction waterfall model. To overcome problems associated wiht the waterfall model's rigidity, you can precede a waterfall with a risk-reduction spiral for requirements analysis or architectural design.*

The risk-reduction preamble to the waterfall lifecycle isn't limited to requirements. You could use it to reduce architectural risk or any other risk to the

project. If the product depends on developing a high-risk nucleus to the system, you might use a risk-reduction cycle to fully develop the high-risk nucleus before you commit to a full-scale project.

7.5 Evolutionary Prototyping

Evolutionary prototyping is a lifecycle model in which you develop the system concept as you move through the project. Usually you begin by developing the most visible aspects of the system. You demonstrate that part of the system to the customer and then continue to develop die prototype based on the feedback you receive. At some point, you and the customer agree that the prototype is "good enough." At that point, you complete any remaining work on the system and release the prototype as the final product. Figure 7-8 depicts this process graphically.

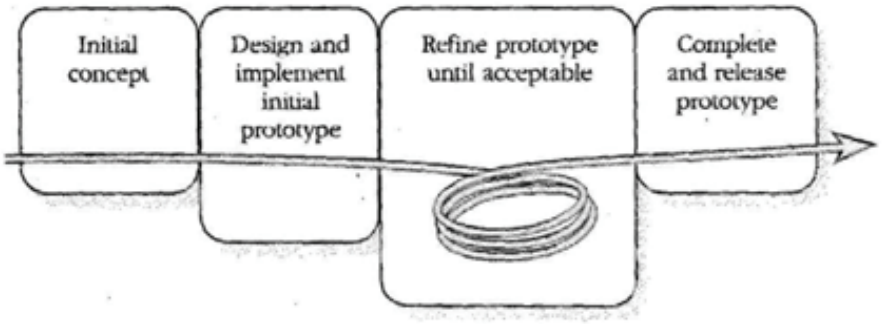


Figure 7-8. *Evolutionary-prototyping model. With evolutionary prototyping, you start by designing and implementing the most prominent parts of the program in a prototype and then adding to and refining the prototvpe until you're done. Toe prototype becomes the software that you eventually release.*

CROSS-REFERENCE
For details on evolutionary Mrtotyping, see Chapter 21, "L.H". ternary Prototyping."

Evolutionary prototyping is especially useful when requirements are changing rapidly, when your customer is reluctant to commit to a set of requirements, or when neither you nor your customer understands the application area well. It is also useful when the developers are unsure of the optimal architecture or algorithms to use. It produces steady, visible signs of progress, which can be especially useful when there is a strong demand for development speed.

The main drawback of this kind of prototyping is that it's impossible to know at the outset of the project how long it will take to create an acceptable product. You don't even know how many iterations you'll have to go through. This drawback is mitigated somewhat by the fact that customers can see steady signs of progress and they therefore tend to be less nervous about

eventually getting a product than with some other approaches. It's also possible to use evolutionary prototyping within a we'll-just-keep-prototyping-until-we-run-out-of-time-or-money-and-then-we'11-declare-ourselves-to-be-done framework.

Another drawback is that this approach can easily become an excuse to do code-and-fix development. Real evolutionary prototyping includes real requirements analysis, real design, and real maintainable code—just in much smaller increments than you'd find with traditional approaches.

7.6 Staged Delivery

The staged-delivery model is another lifecycle model in which you show soft' ware to the customer in successively refined stages. Unlike the evolutionary-prototyping model, when you use staged delivery, you know exactly what you're going to build when you set out to build it. What makes the staged-delivery model distinctive is that you don't deliver the software at the end of the project in one fell swoop. You deliver it in successive stages throughout the project. (This model is also known as "incremental implementation.") Figure 7-9 shows how the model works.

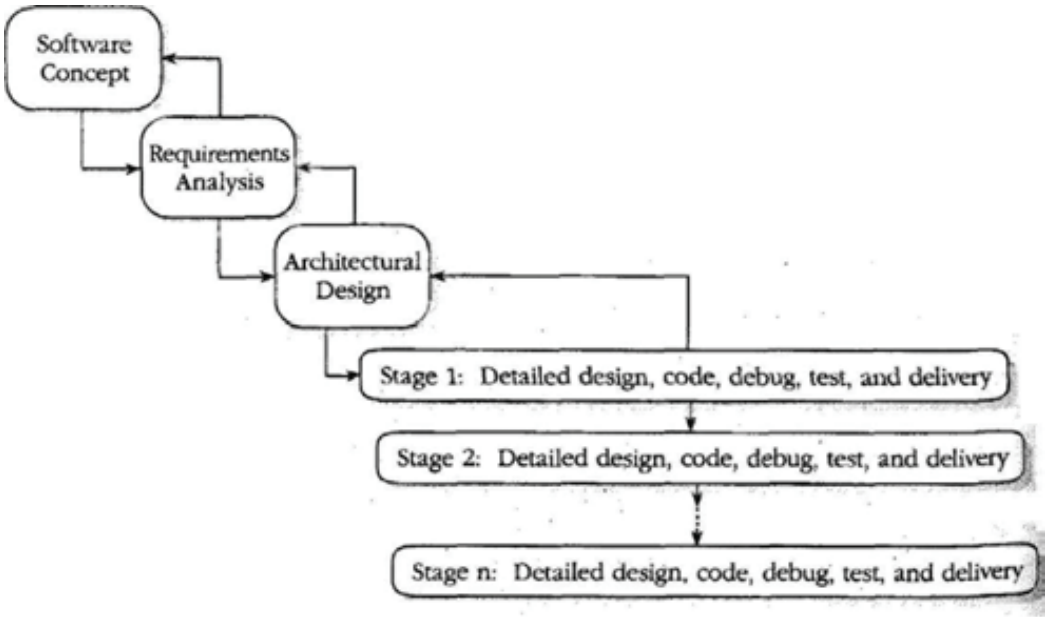


Figure 7-9. *Staged-delivery model. Staged delivery avoids the waterfall model's problem of no part of the system being done until all of it's done. Once you've finished design, you can implement and deliver the system in stages.*

CROSS-REFERENCE
For details on staged delivery, see Chapter 36, "Staged Delivery,"

As Figure 7-9 suggests, with staged delivery you go through the waterfall model steps of defining the software concept, analyzing requirements, arid creating an architectural design for the whole program you intend to build. You then proceed to do detailed design, coding, debugging, and testing within each stage.

The primary advantage of staged delivery is that it allows you to put useful functionality into the hands of your customers earlier than if you delivered 100 percent of the project at the end of the project. If you plan your stages carefully, you may be able to deliver the most important functionality the earliest, and your customers can start using the software at that point.

Staged delivery also provides tangible signs of progress earlier in the project than less incremental approaches do. Such signs of progress can be a valuable ally in keeping schedule pressure to a manageable level.

The main disadvantage of staged delivery is that it won't work without careful planning at both the management and technical levels. At a management level, be sure that the stages you plan are meaningful to the customer and that you distribute work among project personnel in such a way that they can complete their work in time for the stage deadline. At a technical level, be sure that you have accounted for all technical dependencies between different components of the product. A common mistake is to defer development of a component until stage 4 only to find that a component planned for stage 2 can't work without it.

7.7 Design-to-Schedule

The design-to-schedule lifecycle model is similar to the staged-release lifecycle model in that you plan to develop the product in successive stages. The difference is that you don't necessarily know at the outset that you'll ever make it to the last release. You might have five stages planned—but only make it to the third stage because you have an immovable deadline. Figure 7-10 on the next page illustrates this lifecycle model.

This lifecycle model can be a viable strategy for ensuring that you have a product to release by a particular date. If you absolutely must have functioning software in time for a trade show, or by year's end, or by some other immovable date, this strategy guarantees that you will have something. This strategy is particularly useful for parts of the product that you don't want on the critical path. For example, The Microsoft Windows operating system includes several "applets," including WordPad, Paint, and Hearts. Microsoft might use design-to-schedule for those applets to keep them from delaying Windows overall.

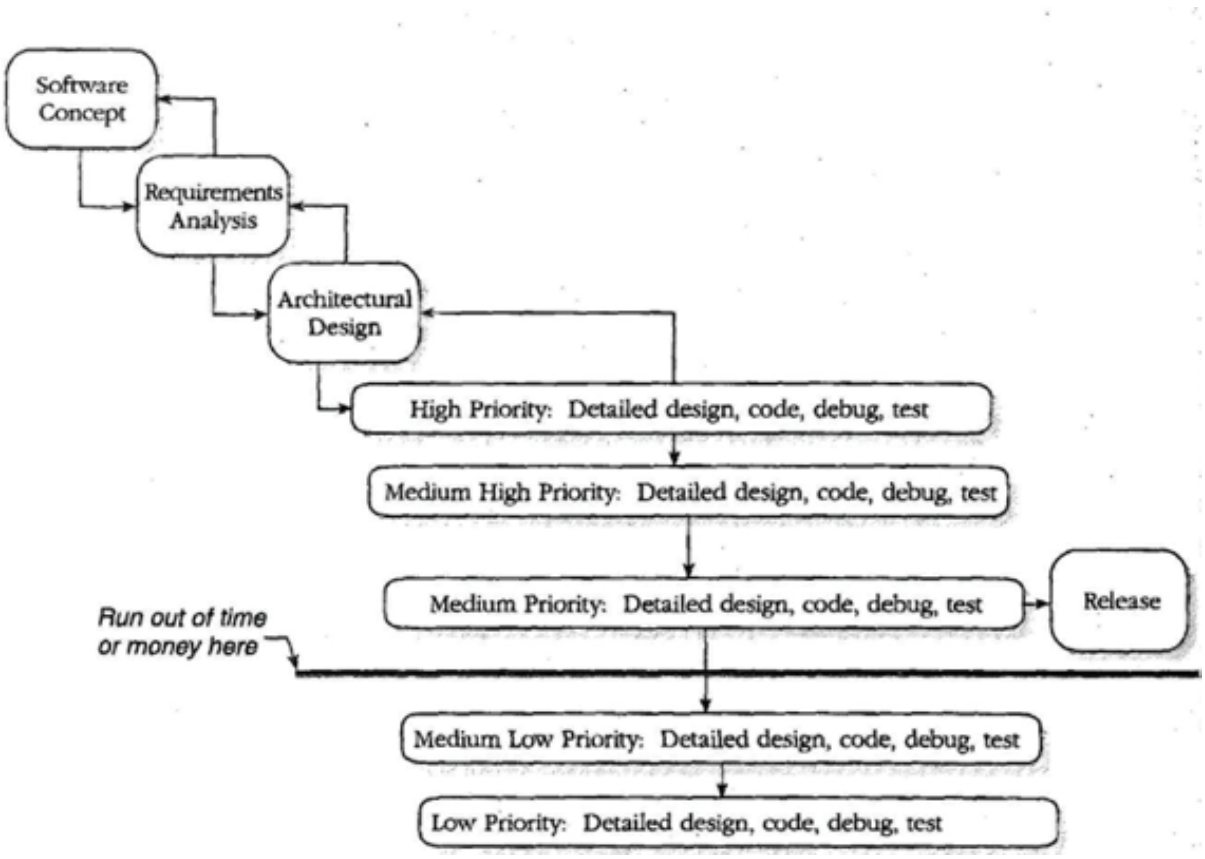


Figure 7-10. *Design-to-schedule model. Design-to-schedule is similar to the staged-release model and is useful when your system has a drop-dead delivery date.*

As Figure 7-10 suggests, one of the critical elements of this lifecycle model is that you prioritize the features and plan your stages so that the early stages contain the highest-priority features. You leave the lower-priority features for later. If the ship date arrives before you've completed all the stages, you don't want to have to leave out critical features because you've spent time implementing less critical ones.

The primary disadvantage of this approach is that if you don't get through all of the stages, you will have wasted time specifying, architecting, and designing features that you don't ship. If you hadn't wasted time on a lot of incomplete features that you didn't ship, you would have had time to squeeze in one or two more complete features.

The decision about whether to use the design-to-schedule model comes down mostly to a question of how much confidence you have in your scheduling ability. If you're highly confident that you can hit your schedule targets, this is an inefficient approach. If you're less confident, this approach just might save your bacon.

7.8 Evolutionary Delivery

Evolutionary delivery is a lifecycle model that straddles the ground between evolutionary prototyping and staged delivery. You develop a version of your product, show it to your customer, and refine the product based on customer feedback. How much evolutionary delivery looks like evolutionary prototyping really depends on the extent to which you plan to accommodate customer requests. If you plan to accommodate most requests, evolutionary delivery will look a lot like evolutionary prototyping. If you plan to accommodate few change requests, evolutionary delivery will look a lot like staged delivery. Figure 7-11 illustrates how the process works.

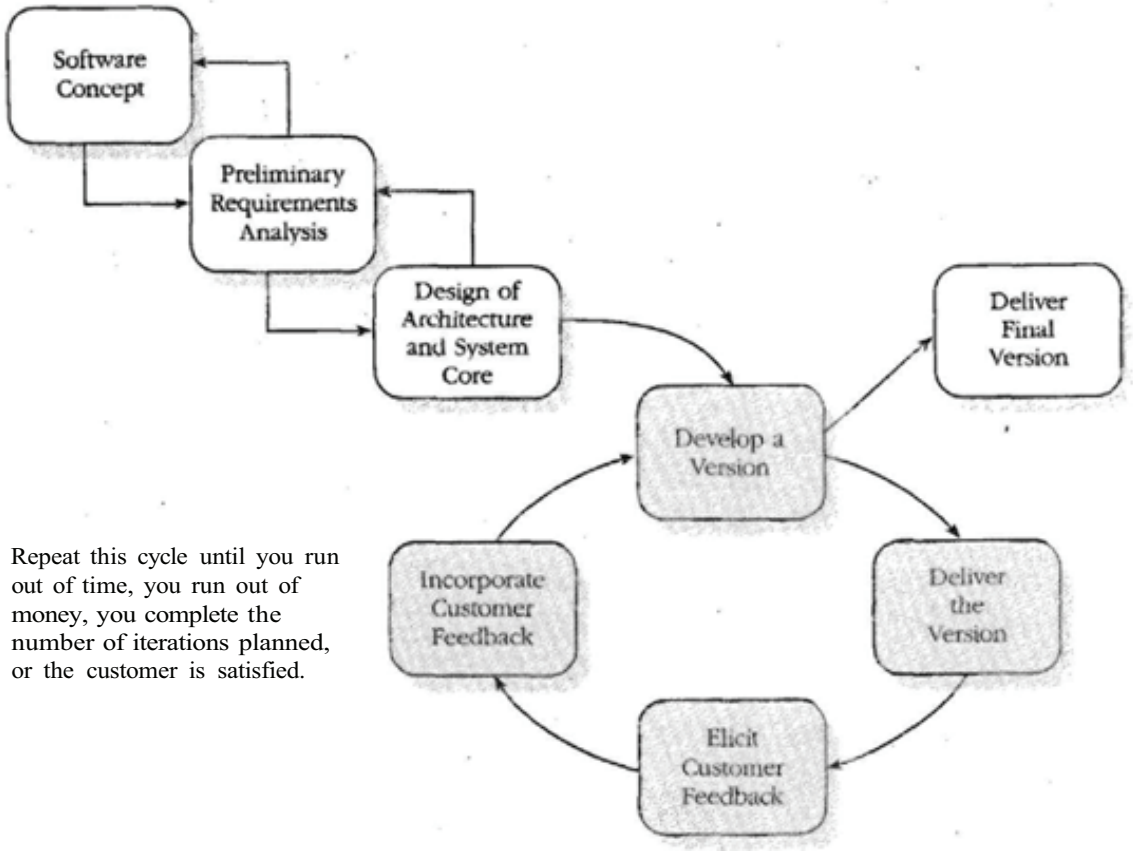


Figure 7-11. *The evolutionary-delivery model. Tins model draws from the control you get with staged delivery and the flexibility you get with evolutionary prototyping. You can tailor it to provide as much control or flexibility as you need.*

CROSS-REFERENCE
For details oh evolutionary delivery, see Chapter 20, "Evolutionary Delivery."

The main differences between evolutionary prototyping and evolutionary delivery are more differences in emphasis than in fundamental approach. In evolutionary prototyping, your initial emphasis is on the visible aspects of the system; you go back later and plug up holes in the system's foundation.

In evolutionary delivery, your initial emphasis is on the core of the system, which consists of lower level system functions that are unlikely to be changed by customer feedback.

Incremental Development Practices

The phrase "incremental development practices" refers to development practices that allow a program to be developed and delivered in stages. Incremental practices reduce risk by breaking the project into a series of small subprojects. Completing small subprojects tends to be easier than completing a single monolithic project. Incremental development practices increase progress visibility by providing finished, operational pieces of a system long before you could make the complete system operational. These practices provide a greater ability to make midcourse changes in direction because the system is brought to a shippable state several times during its development—you can use any of the shippable versions as a jumping-off point rather than needing to wait until the very end.

Lifecycle models that support incremental development include the spiral, evolutionary-prototyping, staged-delivery, and evolutionary-delivery models (discussed earlier in this chapter).

7.9 Design-to-Tools

The design-to-tools lifecycle model is a radical approach that historically has been used only within exceptionally time-sensitive environments. As tools have become more flexible and powerful—complete applications frameworks, visual programming environments, full-featured database programming environments—the number of projects that can consider using design-to-tools has increased.

CROSS-REFERENCE
For more on productivity tools, see Chapter 15, "Productivity Tools;" and Chapter 31, "Rapid-Development Languages (RDLs)."

The idea behind the design-to-tools model is that you put a capability into your product only if it's directly supported by existing software tools. If it isn't supported, you leave it out. By "tools," I mean code and class libraries, code generators, rapid-development languages, and other software tools that dramatically reduce implementation time.

As Figure 7-12 suggests, the result of using this model is inevitably that you won't be able to implement all the functionality you ideally would like to include. But if you choose your tools carefully, you can implement most of the functionality you would like. When time is a constraint, you might actually be able to implement more total functionality than you would have been able to implement with another approach—but it will be the functionality that the tools make easiest to implement, not the functionality you ideally would like.

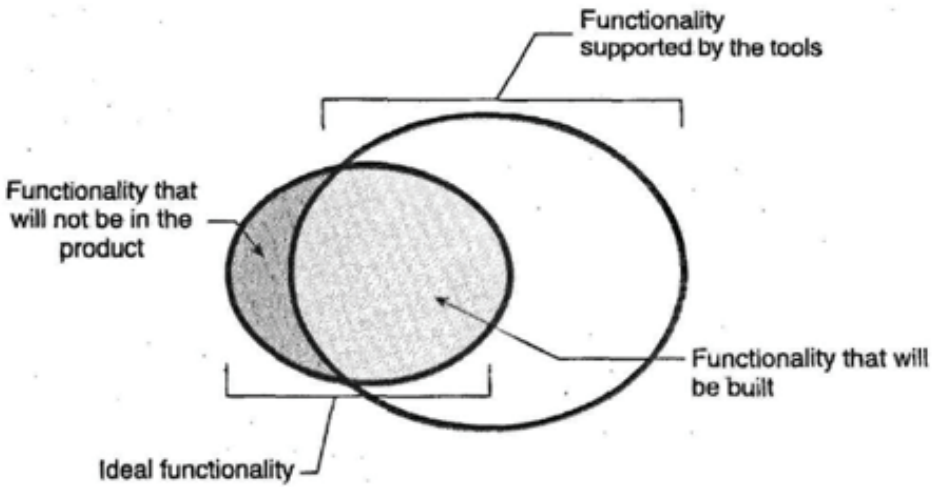


Figure 7-12. *Design-to-tools product concept. Design-to-tools can provide exceptional development speed, but it typically provides less control over your product's functionality than other lifecycle models would provide.*

This model can be combined with the other flexible lifecycle models. You might conduct an initial spiral to identity the capabilities of existing software tools, to identify core requirements, and to determine if the design-to-tools approach is workable. You can use a design-to-tools approach to implement a throwaway prototype, prototyping only the capabilities that can be implemented easily with tools. Then implement the real software using one of the other lifecycle models. You can also combine this model with staged delivery, evolutionary delivery, and design-to-schedule.

The design-to-tools model has a few main disadvantages. You lose a lot of control over your product. You might not be able to implement all the features that you want, and you might not be able to implement other features exactly the way you want. You become more dependent on commercial software producers—on both their product strategies and their financial stabilities. If you're writing small, mostly disposable programs, that might not be much of a problem; but if you're writing programs that you intend to support for a few years, each vendor whose products you use potentially becomes a weak link in the product chain.

7.10 Commercial Off-the-Shelf Software

One alternative that is sometimes overlooked in the excitement surrounding a new system is the option to buy software off the shelf. Off-the-shelf software will rarely satisfy all your needs, but consider the following points.

CROSS-REFERENCE
For details on problems associated with relying on outside vendors for technical products, see Chapter 28, "Outsourcing."

off-the-shelf software is available immediately. In the intervening time between when you can buy off-the-shelf software and when you could release software of your own creation, your users will be provided with at least some valuable capabilities. They can learn to work around the products' limitations by the time you could have provided them with custom software. As time goes by, the commercial software might be revised to suit your needs even more closely.

Custom software probably won't turn out to match your mental vision of the ideal software. Comparisons between custom-built software and off-the-shelf software tend to compare the actual off-the-shelf software to the idealized custom-built software. However, when you actually build your own software, you have to make design, cost, and schedule concessions, and the actual custom-built product will fall short of the ideal you envisioned. If you were to deliver only 75 percent of the ideal product, how would that compare to the off-the-shelf software? (This argument applies to the design-to-tools model, too.)

7.11 Choosing the Most Rapid Lifecycle for Your Project

Different projects have different needs, even if they all need to be developed as quickly as possible. This chapter has described 10 software lifecycle models, which, along with all their variations and combinations, provide you with a full range of choices. Which one is fastest?

There is no such thing as a "rapid-development lifecycle model" because the most effective model depends on the context in which it's used. (See Figure 7-13.) Certain lifecycle models are sometimes touted as being more rapid than others, but each one will be fastest in some situations, slowest in others. A lifecycle model that often works well can work poorly if misapplied (as prototyping was in Case Study 7-1).

To choose the most effective lifecycle model for your project, examine your project and answer several questions:

- How well do my customer and I understand the requirements at the beginning of the project? Is our understanding likely to change significantly as we move through the project?
- How well do I understand the system architecture? Am I likely to need to make major architectural changes midway through the project?
- How much reliability do I need?
- How much do I need to plan ahead and design ahead during this project for future versions?

(continued on page 156)

Dinner Menu

Welcome to le Cafe de Lifecycle Rapide. Bon Appetit!

Entrees

Spiral
Handmade rotini finished with a risk-reduction sauce.
$15.95

Evolutionary Delivery
Mouth-watering melange of staged delivery and evolutionary prototyping.
$15.95

Staged Delivery
A five-course feast. Ask your server for details,
$14.95

Design-to-Schedule
Methodology medley, ideal for quick executive lunches.
$11.95

Pure Waterfall
A classic, still made from the original recipe.
$14.95

Salads

Design-to-Toois
Roast canard generously stuffed with juiienned multi-color beans.
Market Price

Commercial Off-the-Shelf Software
Chef's alchemic fusion of technology du jour. Selection varies daily.
$4.95

Code-and-Fix
Bottomless bowl of spaghetti lightly sprinkled with smoked design and served with reckless abandon.
$5.95

Figure 7-13. *Choosing a lifecvcle model. No one lifecycle model is best for ai projects. The best lifecycle model for any particular project depends on that project's needs.*

- How much risk does this project entail?
- Am I constrained to a predefined schedule?
- Do I need to be able to make midcourse corrections?
- Do I need to provide my customers with visible progress throughout the project?
- Do I need to provide management with visible progress throughout the project?
- How much sophistication do I need to use this lifecycle model successfully?

CROSS-REFERENCE
For more on why a linear, waterfall-like approach is most efficient, see "Wisdom of Stopping Changes Altogether" in Section 14.2.

After you have answered these questions, Table 7-1 should help you decide which lifecycle model to use. In general, the more closely you can stick to a linear, waterfall-like approach—and do it effectively—the more rapid your development will be. Much of what I say throughout this book is based on this premise. But if you have reasons to think that a linear approach won't work, it's safer to choose an approach that's more flexible.

Table 7-1. Lifecycle Model Strengths and Weaknesses

Lifecycle Model Capability	Pure Waterfall	Code-and-Fix	Spiral	Modified Waterfalls	Evolutionary Prototyping
Works with poorly understood requirements	Poor	Poor	Excellent	Fair to excellent	Excellent
Works with poorly understood architecture	Poor	Poor	Excellent	Fair to excellent	Poor to fair
Produces highly reliable system	Excellent	Poor	Excellent	Excellent	Fair
Produces system with large growth envelope	Excellent	Poor to fair	Excellent	Excellent	Excellent
Manages risks	Poor	Poor	Excellent	Fair	Fair
Can be constrained to a predefined schedule	Fair	Poor	Fair	Fair	Poor
Has low overhead	Poor	Excellent	Fair	Excellent	Fair
Allows for midcourse corrections	Poor	Poor to excellent	Fair	Fair	Excellent
Provides customer with progress visibility	Poor	Fair	Excellent	Fair	Excellent
Provides management with progress visibility	Fair	Poor	Excellent	Fair to excellent	Fair
Requires little manager or developer sophistication	Fair	Excellent	'Poor	Poor to fair	Poor

Each rating is either "Poor," "Fair," or "Excellent." Finer distinctions than that wouldn't be meaningful at this level. The ratings in the table are based on the model's best potential. The actual effectiveness of any lifecycle model will depend on how you implement it. It is usually possible to do worse than the table indicates. On the other hand, if you know the model is weak in a particular area, you can address that weakness early in your planning and compensate for it—perhaps by creating a hybrid of one or more of the models described. Of course, many of the table's criteria will also be strongly influenced by development considerations other than your choice of lifecycie models.

Here are detailed descriptions of the lifecycle-model criteria described in Table 7-1:

Works with poorly understood requirements refers to how well the lifecycle model works when either you or your customer understand the system's requirements poorly or when your customer is prone to change requirements. It indicates how well-suited the model is to exploratory software development.

Lifecycle Model Capability	Staged Delivery	Evolutionary Delivery	Design-to-Schedule	Design-to-Tools	Commercial Qff-the-Shelf Software
Works with poorly understood requirements	Poor	Fair to excellent	Poor to fair	Fair	Excellent
Works with poorly understood architecture	Poor	Poor	Poor	Poor to excellent	Poor to excellent
Produces highly reliable system	Excellent	Fair to excellent	Fair	Poor to excellent	Poor to excellent
Produces system with large growth envelope	Excellent	Excellent	Fair to excellent	Poor	N/A
Manages risks	Fair	Fair	Fair to excellent	Poor to fair	N/A
Can be constrained to a predefined schedule	Fair	Fair	Excellent	Excellent	Excellent
Has low overhead	Fair	Fair	Fair	Fair to excellent	Excellent
Allows for midcourse corrections	Poor	Fair to excellent	Poor to fair	Excellent	Poor
Provides customer with progress visibility	Fair	Excellent	Fair	Excellent	N/A
Provides management with progress visibility	Excellent	Excellent	Excellent	Excellent	N/A
Requires little manager or develooer soohistication	Fair	Fair	Poor	Fair	Fair

Works with poorly understood architecture refers to how well the lifecycU: model works when you're developing in a new application area or when you're developing in a familiar applications area but are developing unfa* miliar capabilities.

Produces highly reliable system, refers to how many defects a system developed with the lifecycle model is likely to have when put into operation.

Produces system ivith large growth envelope refers to how easily you're likely to be able to modify the system in size and diversity over its lifetime. This includes modifying the system in ways that were not anticipated by the original designers.

Manages risks refers to the model's support for identifying and controlling risks to the schedule, risks to the product, and other risks.

Can be constrained to a predefined schedule refers to how well the lifecycle model supports delivery of software by an immovable drop-dead date.

Has low overhead refers to the amount of management and technical overhead required to use the model effectively. Overhead includes planning, status tracking, document production, package acquisition, and other activities that aren't directly involved in producing software itself.

Allows for midcourse corrections refers to the ability to change significant aspects of the product midway through the development schedule. This does not include changing the product's basic mission but does include sijv fi cantly extending it.

Provides customer with progress visibility refers to the extent to which the model automatically generates signs of progress that the customer can use to track the project's status.

Provides management with progress visibility refers to the extent to which the model automatically generates signs of progress that management can use to track the project's status.

Requires little manager or developer sophistication refers to the level of education and training that you need to use the model successfully. That includes the level of sophistication you need to track progress using the model, to avoid risks inherent in the model, to avoid wasting time using the model, and to realize the benefits that led you to use the model in the first place.

Case Study 7-2. Effective Lifecycle Model Selection

Eddie had volunteered to oversee Square-Tech's development of a new product code named "Cube-It," a scientific graphics package. Rex, the CEO, felt that Square-Calc had given them a foot in the door they could use to become a market leader in scientific graphics.

Eddie met with George and Jill, both developers, to plan the project. "This is a new area for us, so I want to minimize the company's risk on this project. Rex told me that he wanted the preliminary product spec implemented within a year. I don't know whether that's possible, so I want you to use a spiral lifecycle model. For the first iteration of the spiral, we need to find out whether this preliminary spec is pure fantasy or whether we can make it a reality."

George and Jill worked for two weeks and then met with Eddie to evaluate the alternatives they had identified. "Here's what we found out. If the objective is to build the market-leading scientific-graphics package, there are two basic alternatives: beat the competition in features or beat them in ease of use. Right now, the easier niche to fill seems to be ease of use.

"We analyzed the risks for each alternative. If we go the full-feature route, we're looking at a minimum of about 200 staff-months to develop a market-leading product. We have the constraints of a maximum of 1 year to ship a product and a maximum team size of 8 people. We can't meet those constraints with the full-featured product. If we go the usability route, we're looking at more like 75 staff-months. That fits with our constraints, and there will be more room in the market for us."

"That's good work," Eddie said. "I think Rex will like that." Eddie met with Rex later that day, and then he got back together with George and Jill the following morning.

"Rex pointed out that we need to develop some in-house usability experts. He thought that developing a product that emphasizes usability was a good strategic move, so he gave us the thumbs-up.

"Now we need to plan the next iteration of the spiral. Our goal for this iteration is to refine the product spec in ways that minimize our development time and maximize usability."

George and Jill spent 4 weeks on the iteration, and then they met with Eddie to review their findings. "We've created a prioritized list of preliminary requirements," George reported. "The list is sorted by usability and then by estimated

(continued)

Case Study 7-2. Effective Lifecycle Model Selection, *continued*

implementation time. We've made both best-case and worst-case effort estimates for each feature. You can see that there's a lot of variation, and a lot of that variation just has to do with how we define the specifics of each feature. In other words, we have a lot of control over how much time this product takes to implement.

"Having maximum usability as our clear, primary objective really makes some decisions easy for us. Some of the most time-consuming features to implement would also be the least usable. I recommend that we just eliminate some of them because it will be a win for both the schedule and the product."

"That's interesting," Eddie responded. "What high-level alternatives have you come up with?"

"We recommend either of two possibilities," Jill said. "We've got the 'safe' version, which puts a strong emphasis on usability but uses proven technology. And we've got the 'risky' version, which pushes the usability state of the art. Either one should be a lot more usable than anything else on the market. The risky version will make it harder for the competition to catch up with us, but it will also nominally take about 60 staff-months compared with the safe version's 40 staff-months. That's not all that much of a difference, but the worst case for the risky version is 120 staff-months compared with the safe version's 55."

"Wow!" Eddie said. "That's good information. Is it possible to implement the safe version but design ahead so that we can push the state of the art in version 2?"

"I'm glad you asked that," Jill said. "We estimated that the safe version with design-ahead for version 2 would nominally take 45 staff-months, with a worst-case of 60."

"That makes it pretty clear, doesn't it?" Eddie said. "We've got 10½ months left, so let's do the safe version with design-ahead for version 2. While you all were focusing on the technical schedule risk, I've been focusing on the personnel schedule risk, and I've got three developers lined up. We'll add them to the team now and start the next iteration.

"George, you mentioned that a lot of the variation in schedule has to do with how each feature is ultimately defined, right? For the next spiral iteration, we need to focus on minimizing our design and implementation risk, and that means defining as many of those features to take as little implementation time as possible while staying consistent with our usability goal. I also want to have the new developers review your estimates to reduce the risk of any estimation error." George and Jill agreed.

The next iteration, which focused on design, took 3 months, bringing the project to the 4½-month mark. Their reviews had convinced them that their

(continued

Case Study 7-2. Effective Lifecycle Model Selection, *continued*

design was solid—including the design-ahead for version 2. The design work had allowed them to refine their estimates, and they now estimated that the remaining implementation would take 30 staff-months, with a worst case of 40. Eddie thought that was exceptional because it meant that the worst case had them delivering the software only 2 weeks late.

At the beginning of the coding iteration, the developers identified low code quality and poor status visibility as their primary risks. To minimize those risks, they established code reviews to detect and correct coding errors, and they used miniature milestones to provide excellent status visibility.

Their estimates hadn't been perfect, and the final iteration took 2 weeks longer than nominal. They delivered the first release candidate to system testing at 11 months instead of at 10½. But the product's quality was excellent, and it took only two release candidates to declare a winner. Cube-It 1.0 was released on time.

Further Reading

DeGrace, Peter, and Leslie Hulet Stahl. *Wicked Problems, Righteous Solutions.* Englewood Cliffs, N.J.: Yourdon Press, 1990. The subtitle of this book is "A Catalog of Modern Software Engineering Paradigms," and it is by far the most complete description of software lifecytie models available. The book was produced through an unusual collaboration in which Peter DeGrace provided the useful technical content and Leslie Hulet Stahl provided the exceptionally readable and entertaining writing style.

Boehm, Barry W., ed. *Software Risk Management.* Washington, DC: IEEE Computer Society Press, 1989. This tutorial is interesting for the introduction to Section 4, "Implementing Risk Management." Boehm describes how to use the spiral model to decide which software lifecycle model to use. The volume also includes Boehm's papers, "A Spiral Model of Software Development and Enhancement" and "Applying Process Programming to the Spiral Model," which introduce the spiral lifecycle model and describe an extension to it (Boehm 1988; Boehm and Belz 1988).

Jones, Capers. *Assessment and Control of Software Risks.* Englewood Cliffs, N.J.: Yourdon Press, 1994. Chapter 57, "Partial Life-Cycle Definitions' describes the hazards of not breaking down your lifecycie description into enough detail. It provides a summary of the 25 activities that Jones says make up most of the work on a successful software project.