

From: Steve McConnell - "Rapid Development"
(ISBN 1-55615-900-5) pp 154-161

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?

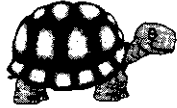
There is no such thing as a "rapid-development lifecycle model" because the most effective model depends on the context in which it's used. (See Figure 7-13.) Certain lifecycle models are sometimes touted as being more rapid than others, but each one will be fastest in some situations, slowest in others. 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 Café de Lifecycle Rapide. Bon Appétit!

Entrees

Spiral

Handmade rotini finished with a risk-reduction sauce.

\$15.95

Evolutionary Delivery

Mouth-watering mélange 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-Tools

Roast canard generously stuffed with julienned 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 lifecycle model. No one lifecycle model is best for all 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 lifecycle 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 Off-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 developer sophistication	Fair	Fair	Poor	Fair	Fair

Works with poorly understood architecture refers to how well the lifecycle model works when you're developing in a new application area or when you're developing in a familiar applications area but are developing unfamiliar 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 with 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 significantly 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.
