20 Evolutionary Delivery

SUMMARY

Evolutionary Delivery is a lifecycle model that strikes a balance between Staged Delivery's control and Evolutionary Prototyping's flexibility. It provides its rapid-development benefit by delivering selected portions of the software earlier than would otherwise be possible, but it does not necessarily deliver the final software product any faster. It provides some ability to change product direction mid-course in response to customer requests. Evolutionary Delivery has been used successfully on in-house business software and shrink-wrap software. Used thoughtfully, it can lead to improved product quality, reduced code size, and more even distribution of development and testing resources. As with other lifecycle models, Evolutionary Delivery is a whole-project practice: if you want to use it, you need to start planning to use it early in the project.

Efficacy

Potential reduction from nominal schedule:	Good
Improvement in progress visibility:	Excellent
Effect on schedule risk:	Decreased Risk
Chance of first-time success:	Very Good
Chance of long-term success:	Excellent

Major Risks

- Feature creep
- Diminished project control
- Unrealistic schedule and budget expectations
- Inefficient use of development time by developers

Major Interactions and Trade-Offs

- Draws from both Staged Delivery and Evolutionary Prototyping
- Success depends on use of designing for change

Some people go to the grocery store carrying a complete list of the groceries they'll need for the week: "2 pounds of bananas, 3 pounds of apples, 1 bunch of carrots," and so on. Other people go to the store with no list at all and buy whatever looks best when they get there: "These melons smell good. I'll get a couple of those. These snow peas look fresh. I'll put them together with some onions and water chestnuts and make a stir-fry. Oh, and these porterhouse steaks look terrific. I haven't had a steak in a long time. I'll get a couple of those for tomorrow." Most people are somewhere in between. They take a list to the store, but they improvise to greater and lesser degrees when they get there.

In the world of software lifecycle models, Staged Delivery is a lot like going to the store with a complete list. Evolutionary Prototyping is like going to the store with no list at all. Evolutionary Delivery is like starting with a list but improvising some as you go along.

The Staged Delivery lifecycle model provides highly visible signs of progress to the customer and a high degree of control to management, but not much flexibility. Evolutionary Prototyping is nearly the opposite: like Staged Delivery, it provides highly visible signs of progress to the customer—but unlike Staged Delivery, it provides a high degree of flexibility in responding to customer feedback and little control to management. Sometimes you want to combine the control of Staged Delivery with the flexibility of Evolutionary Prototyping. Evolutionary Delivery straddles the ground between those two lifecycle models and draws its advantages and disadvantages from whichever it leans toward the most.

CROSS-REFERENCE
For details on these kinds of support for rapid development, see the introductions to Chapter 21, "Evolutionary Prototyping," and Chapter 36, "Staged Delivery."

Evolutionary Delivery supports rapid development in several ways:

- It reduces the risk of delivering a product that the customer doesn't want, avoiding time-consuming rework.
- For custom software, it makes progress visible by delivering software early and often.
- For shrink-wrap software, it supports more frequent product releases.
- It reduces estimation error by allowing for recalibration and reestimation after each evolutionary delivery.
- It reduces the risk of integration problems by integrating early and often—whenever a delivery occurs.
- It improves morale because the project is seen as a success from the first time the product says, "Hello World" until the final version is ultimately delivered.

As with other aspects of Evolutionary Delivery, the extent to which it supports rapid development in each of these ways depends on whether it leans more toward Staged Delivery or Evolutionary Prototyping.

20.1 Using Evolutionary Delivery

To use Evolutionary Delivery, you need to have a fundamental idea of the kind of system you're building at the outset of the project. As Figure 20-1 suggests, in the evolutionary-delivery approach, you start with a preliminary idea of what your customer wants, and you create a system architecture and core based on that. That architecture and core serve as the basis for further development.

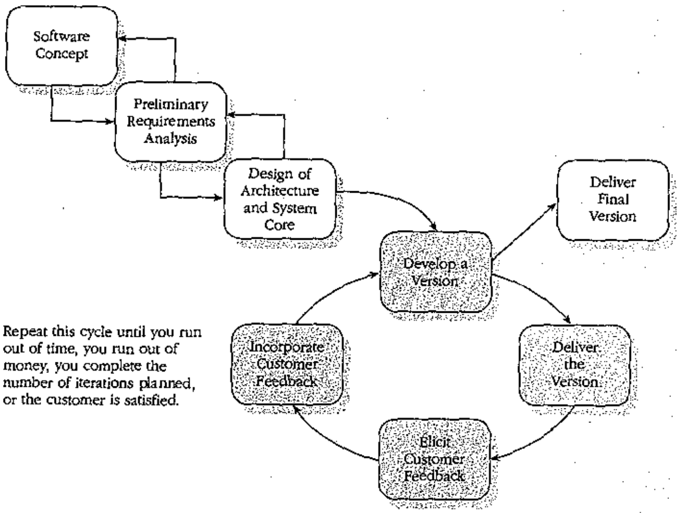
Software Concept
Preliminary Requirements Analysis
Design of Architecture and System Core
Develop a Version
Deliver Final Version
Deliver the Version
Elicit Customer Feedback
Incorporate Customer Feedback
Repeat this cycle until you run out of time, you run out of money, you complete the number of iterations planned, or the customer is satisfied.

Figure 20-1. *The Evolutionary Delivery lifecycle model draws from Staged Delivery's control and Evolutionary Prototyping's flexibility. You can tailor it to provide as much control or flexibility as you need.*

The architecture should anticipate as many of the possible directions the system could go as it can. The core should consist of lower-level system functions that are unlikely to change as a result of customer feedback. It's fine to be uncertain about the details of what you will ultimately build on top of the core, but you should be confident in the core itself.

Properly identifying the system core is one key to success in using Evolutionary Delivery. Aside from that, the most critical choice you make in Evolutionary Delivery is whether to lean more toward Evolutionary Prototyping or Staged Delivery.

In Evolutionary Prototyping, you tend to iterate until you and the customer agree that you have produced an acceptable product. How many iterations will that take? You can't know for sure. Usually you can't afford for a project to be that open-ended.

In Staged Delivery, on the other hand, you plan during architectural design how many stages to have and exactly what you want to build during each stage. What if you change your mind? Well, pure Staged Delivery doesn't allow for that.

With Evolutionary Delivery, you can start from a base of Evolutionary Prototyping and slant the project toward Staged Delivery to provide more control. You can decide at the outset that you will deliver the product in four evolutionary deliveries. You invite the customer to provide feedback at each delivery, which you can then account for in the next delivery. But the process will not continue indefinitely: it will stop after four deliveries. Deciding on the number of iterations in advance and sticking to it is one of the critical factors to success in this kind of rapid development (Burlton 1992).

With Evolutionary Delivery, another option is to start from a base of Staged Delivery and slant the project toward Evolutionary Prototyping to provide more flexibility. You can decide at the outset what you will deliver in stages 1, 2, and 3, but you can be more tentative about stages 4 and 5, thus giving your project a direction but not an exact road map.

Whether you slant more toward Evolutionary Prototyping or Staged Delivery should depend on the extent to which you need to accommodate customer requests. If you need to accommodate most requests, set up Evolutionary Delivery to be more like prototyping. Some experts recommend delivering the software in increments as small as 1 to 5 percent (Gilb 1988). If you need to accommodate few change requests, plan it to be more like Staged Delivery, with just a handful of larger releases.

Release Order

CROSS-REFERENCE
For details on the value of mapping out possible changes to a system at design time, see "Define Families of Programs" in Section 19.1.

You use Evolutionary Delivery when you're not exactly sure what you want to build. But unlike Evolutionary Prototyping, you do have at least some idea, so you should map out a preliminary set of deliveries at the beginning of your project, while you're developing the system architecture and system core.

Your initial delivery schedule will probably contain some functionality that will definitely be in the final system, some that will probably be, and some that might not be. Still other functionality will be identified and added later.

Be sure that you structure your delivery schedule so that you develop the functionality you are most confident in first. You can show the system to your customers, elicit their feedback, and gain confidence that the remaining functionality is really what they want—before you begin working on it.

When to Use Evolutionary Delivery

CROSS-REFERENCE For a related discussion of planning decisions that depend on how well-understood your system is, see Section 13.1, "Team-Structure Considerations."

If you understand your system so well that you don't expect many surprises, you will be better off using Staged Delivery than Evolutionary Delivery. You have more control over the project when you map out exactly what will be delivered at the end of each stage ahead of time. That style of development is most often appropriate for maintenance releases or product upgrades in which you already understand the product area thoroughly and don't expect to discover any revolutionary insights along the development path.

If your system is poorly understood and you expect a lot of surprises, including surprises that are likely to affect the system in fundamental ways, you will be better off using Evolutionary Prototyping than Evolutionary Delivery. Evolutionary Delivery requires that you know enough about the project at the outset to design an architecture and to implement core functionality. With Evolutionary Prototyping, you don't have to do those things at the beginning.

Evolutionary Delivery isn't limited to completely new systems. On an existing system, each new delivery can replace part of the old system and can also provide new capabilities. At some point, you'll have replaced so much of the old system that there's nothing left. The architecture of the new system needs to be designed carefully so that you can transition to it without being infected by the old system's restrictions. That might mean that you can't create certain portions of the new system right away; you might want to wait until the new system can support those portions without kludges.

20.2 Managing the Risks of Evolutionary Delivery

Here are the risks you should plan for if your version of Evolutionary Delivery leans more toward Evolutionary Prototyping:

CROSS-REFERENCE For details on these risks, see Section 21.2, "Managing the Risks of Evolutionary Prototyping."

- Unrealistic schedule and budget expectations arising from rapid early progress
- Diminished project control

e results weren't what you ex-
fy the cost/benefit analysis, or

e your estimation by giving you
se your development speed, but
you promise, which can be just
nment. Tom Gilb calls this the
jects might be useful, but it can't
our present project (Gilb 1988).

nip your product every few weeks
amount of time it takes to learn
ing a product cycle. End-to-end
xperience to have, and when you
get that experience faster.

ry is tilted toward Evolutionary
side effects:

and developers because

for Change (Chapter 19). Evolutionary Delivery invites changes, and your system needs to be set up to accommodate them.

20.5 The Bottom Line on Evolutionary Delivery

CROSS-REFERENCE
For more on the importance of making progress visible, see Section 6.3, "Perception and Reality."

The bottom line on Evolutionary Delivery is that, whether or not it reduces overall development time, it provides tangible signs of progress that can be just as important to rapid development as all-out development speed. It might appear that this incremental approach takes longer than a more traditional approach, but it almost never does because it keeps you and your customer from getting too far out of touch with your real progress.

HARD DATA

Evolutionary Prototyping has been reported to decrease development effort by from 45 to 80 percent (Gordon and Bieman 1995). Staged Delivery, on the other hand, does not generally reduce development time, but it makes progress more visible. The specific reduction in development time that you experience from Evolutionary Delivery will depend on the specific mix of Evolutionary Prototyping and Staged Delivery that you use.

developers

your project leans more toward

nefficient use of development

Delivery

- Early feedback on whether the final system will be acceptable
- Decreased overall code length
- Defect reduction due to better requirements definition
- Smooth effort curves that reduce the deadline effect (which commonly arises when using traditional development approaches)

CROSS-REFERENCE
For details on these side effects, see Section 36.3, "Side Effects of Staged Delivery."

If your version of Evolutionary Delivery is tilted toward Staged Delivery, you might experience these side effects:

- More even distribution of development and testing resources
- Improved code quality
- More likely project completion
- Support for build-to-budget efforts

20.4 Evolutionary Delivery's Interactions with Other Practices

20.6 Keys to Success in Using Evolutionary Delivery

Here are the keys to using Evolutionary Delivery successfully:

- Be sure that the product architecture supports as many of the system's possible directions as you can imagine.
- Define the system's core carefully.
- Decide whether to lean more toward Evolutionary Prototyping or Staged Delivery, based on the extent to which you need to accommodate customer change requests.
- Order the functionality in your initial set of releases from "most certain" to "least certain." Assume that the number of changes will increase in later releases.
- Explicitly manage user expectations having to do with schedule, budget, and performance.
- Consider whether pure Staged Delivery or pure Evolutionary Prototyping might be a better fit for your project.

Further Reading

Gilb, Tom. *Principles of Software Engineering Management.* Wokingham, England: Addison-Wesley, 1988. Chapters 7 and 15 contain thorough discussions of staged delivery and evolutionary delivery.

McConnell, Steve. *Code Complete.* Redmond, Wash.: Microsoft Press, 1993. Chapter 27 describes the evolutionary-delivery practice.

Cusumano, Michael, and Richard Selby. *Microsoft Secrets: How the World's Most Powerful Software Company Creates Technology, Shapes Markets, and Manages People.* New York: Free Press, 1995. Chapter 4 describes Microsoft's milestone process, which could be considered to be a cousin to evolutionary delivery. Cusumano and Selby are professors at MIT and UC Irvine, respectively, and they present the outsider's view of the process.

McCarthy, Jim. *Dynamics of Software Development.* Redmond, Wash.: Microsoft Press, 1995. McCarthy describes the insider's view of Microsoft's milestone process. From an evolutionary-delivery viewpoint, his book is particularly interesting because it focuses on Visual C++, a subscription shrink-wrap product that's shipped every four months.

21

Evolutionary Prototyping

SUMMARY

Evolutionary Prototyping is a lifecycle model in which the system is developed in increments so that it can readily be modified in response to end-user and customer feedback. Most evolutionary-prototyping efforts begin by prototyping the user interface and then evolving the completed system from that, but prototyping can start with any high-risk area. Evolutionary Prototyping is not the same as Throwaway Prototyping, and making the right choice about whether to develop an evolutionary prototype or a throwaway prototype is one key to success. Other keys to success include using experienced developers, managing schedule and budget expectations, and managing the prototyping activity itself.

Efficacy

Potential reduction from nominal schedule:	Excellent
Improvement in progress visibility:	Excellent
Effect on schedule risk:	Increased Risk
Chance of first-time success:	Very Good
Chance of long-term success:	Excellent

Major Risks

- Unrealistic schedule and budget expectations
- Inefficient use of prototyping time
- Unrealistic performance expectations
- Poor design
- Poor maintainability

(continued)

Major Interactions and Trade-Offs

- Trades a reduction in project control for increased end-user and customer feedback and for better progress visibility
- Can be combined with User-Interface Prototyping and Throwaway Prototyping
- Can serve as a basis for Evolutionary Delivery

Evolutionary Prototyping is a development approach in which you develop selected parts of a system first and then evolve the rest of the system from those parts. Unlike other kinds of prototyping, in Evolutionary Prototyping you don't discard the prototyping code; you evolve it into the code that you ultimately deliver. Figure 21-1 shows how this works.

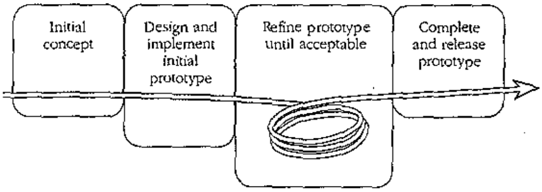
Initial concept
Design and implement initial prototype
Refine prototype until acceptable
Complete and release prototype

Figure 21-1. *Evolutionary Prototyping model. With Evolutionary Prototyping, you start by designing and implementing the most prominent parts of the program in a prototype. You then continue adding to and refining the prototype until you're done. The prototype eventually evolves into the software that you release.*

Evolutionary Prototyping supports rapid development by addressing risks early. You start development with the riskiest areas of your system. If you can overcome the obstacles, you can evolve the rest of the system from the prototype. If you can't, you can cancel the project without having spent any more money than necessary to discover that the obstacles were insurmountable.

Rapid Prototyping

Evolutionary Prototyping is one of a set of practices that are often described as "rapid prototyping." The term "rapid prototyping" is a loose description that generally refers to Evolutionary Prototyping but which often refers to any prototyping practices.

21.1 Using Evolutionary Prototyping

Evolutionary Prototyping is an exploratory activity that you use when you don't know at the outset exactly what you need to build. Because the areas of uncertainty in a development effort vary from one project to another, your first step is to identify the part of the system to use as a starting point. The two main options are to start with the most visible parts or the riskiest parts. The user interface is often both the most visible and the riskiest part, so it is often the obvious place to start.

After you've built the first part of the system, you demonstrate it and then continue to develop the prototype based on the feedback you receive. If your prototyping effort is customer-oriented, you'll continue to solicit feedback from the customer and to refine the user interface until everyone agrees that the prototype is good enough. Then you develop the rest of the system using the prototype's design and code as a foundation.

If your prototyping effort is technically oriented, you might develop some other part of the system first, such as the database. The feedback you'll look for in that case won't necessarily come from the customer. Your mission might be to benchmark database size or performance with a realistic number of end-users. But the general prototyping pattern is the same. You continue to look for feedback and to refine your prototype until everyone agrees that the prototype is good enough, and then you complete the system. The same principle applies to prototyping of user interfaces, complex calculations, interactive performance, real-time timing constraints, data sizing, and proof-of-concept aspects of leading-edge systems.

You can use Evolutionary Prototyping to greater or lesser degrees on most kinds of projects. It is probably best suited to business systems in which developers can have frequent, informal interactions with end-users. But it is also well suited to commercial, shrink-wrap, and systems projects as long as you can get end-users involved. The user interaction for these kinds of projects will generally need to be more structured and formal.

21.2 Managing the Risks of Evolutionary Prototyping

HARD DATA

In spite of a long list of risks to watch for, Evolutionary Prototyping is a relatively low-risk practice. Of published case studies, 33 of the 39 have reported success (Gordon and Bieman 1995). Here are some risks to keep in mind.

CROSS-REFERENCE
For related risks, see Section 38.2, "Managing the Risks of Throwaway Prototyping," and Section 42.2, "Managing the Risks of User-Interface Prototyping."

Unrealistic schedule and budget expectations. Evolutionary Prototyping may create unfulfillable expectations about the overall development schedule. When end-users, managers, or marketers see rapid progress on a prototype, they sometimes make unrealistic assumptions about how quickly you can develop the final product. In some instances, the sales staff passes along unrealistic schedule expectations to customers. Customers later become upset when they hear that the software will take longer than they expected.

It's easy to complete the most visible work quickly, but a great deal of the work in a programming project isn't obvious to the customer or end-user. Low-visibility work includes robust database access, maintenance of database integrity, security, networking, data conversion between successive product versions, design for high-volume usage, multi-user support, and multi-platform support. The better this work is done, the less noticeable it will be.

Another problem with the effort expectations created when prototyping starts with the user interface is that some of the functionality that looks easiest from the end-user's point of view is the most difficult to implement from the developer's point of view. Difficult areas that look easy include cut-and-paste within an application, cut-and-paste from other applications, printing, print preview, and WYSIWYG screen displays.

HARD DATA

Finally, prototypes are generally designed to handle only the nominal cases; they aren't expected to handle the exceptional cases. But as much as 90 percent of a normal program consists of exception handling (Shaw in Bentley 1982), which suggests that even if a prototype were completely functional for all nominal cases, it would still take up to 9 times as much effort to implement the exceptional cases as it took to implement the fully functioning, nominal-case prototype.

If management does not understand the difference between creating a limited prototype and a full-scale product, it will be at risk of seriously underbudgeting prototyping projects. Scott Gordon and James Bieman reported a case in which management bought a 2-year project as a 6-week project. Management believed that the prototyping would produce fully functional results in an amount of time that was out of touch with real-world needs by a factor of almost 20 (Gordon and Bieman 1995).

CROSS-REFERENCE
For more on expectation management, see Section 10.3, "Managing Customer Expectations."

You can manage this risk by explicitly managing the expectations of end-users, managers, developers, and marketers. Be sure they interact with the prototype in a controlled setting in which you can explain the prototype's limitations, and be sure that they understand the difference between creating a prototype and creating completely functional software.

Diminished project control. With Evolutionary Prototyping, it's impossible to know at the outset of the project how long it will take to create an accep able product. You don't know how many iterations it will take or how long each iteration will last.

CROSS-REFERENCE
For details on realizing benefits of both Evolutionary Prototyping and Staged Delivery, see Chapter 20, "Evolutionary Delivery."

This risk is mitigated somewhat by the fact that customers can see steady signs of progress and tend to be less nervous about eventually getting a product than with traditional development approaches. It's also possible to use an evolutionary-delivery approach that's strongly tilted toward Evolutionary Prototyping, which provides some of the control of Staged Delivery and some of the flexibility of Evolutionary Prototyping.

CROSS-REFERENCE
For details on the risk of poor feedback, see "End-User Feedback and Involvement" in Section 42.1.

Poor end-user or customer feedback. Prototyping doesn't guarantee high-quality end-user or customer feedback. End-users and customers don't always know what they're looking at when they see a prototype. One common phenomenon is that they are so overwhelmed by the live software demonstration that they don't look past the glitz to understand what the prototype really represents. If they give you a rubber-stamp approval, you can be virtually certain that they don't fully understand what they're seeing. Be sure that end-users and customers study the prototype carefully enough to provide meaningful feedback.

Poor product performance. Several factors can contribute to poor product performance, including:

- Not considering performance in the product's design
- Keeping poorly structured, inefficient code that was developed quickly for the prototype instead of evolving it to production quality
- Keeping a prototype that was originally intended to be thrown away

You can take three steps to minimize the risk of poor performance:

- Consider performance early. Be sure to benchmark performance early, and be sure that the prototype's design supports an adequate level of performance.
- Don't develop quick-and-dirty code for the prototype. A prototype's code quality needs to be good enough to support extensive modifications, which means that it needs to be at least as good as the code of a traditionally developed system.
- Don't evolve a throwaway prototype into the final product.

Unrealistic performance expectations. The risk of unrealistic performance expectations is related to the risk of poor performance. This risk arises when the prototype has much better performance than the final product. Because a prototype doesn't have to do all of the work that a final product does, it can sometimes be very fast compared to the final product. When customers see the final product, it can appear to be unacceptably slow.

This risk also has an evil twin. If a prototyping language is used that has much worse performance than the target language, customers may equate weaknesses in the prototype's performance with weaknesses in the final product. More than one case study has reported project failure as a result of low expectations created by a prototype (Gordon and Bieman 1995).

You can address this risk by explicitly managing the expectations that customers develop from interacting with the prototype. If the prototype is very fast, add artificial delays to the prototype so that the prototype's performance approximates the performance of the final product. If the prototype is very slow, explain to customers that the performance of the prototype isn't indicative of the performance of the final product.

Poor design. Many prototyping projects have better designs than traditional projects, but several factors contribute to the risk of poor design.

- A prototype's design can deteriorate and its implementation can drift from its original design during successive stages (just as most software products tend to deteriorate during maintenance). The final product can inherit design patches from the prototype system.
- Sometimes customer or end-user feedback steers the product in an unexpected direction. If the design didn't anticipate the direction, it might be twisted to fit without doing a full redesign.
- The design that flows out of a user-interface–focused prototyping activity might focus on the user interface excessively; it might fail to account for other parts of the system that should also have strong influences on the system's design.
- The use of a special-purpose prototyping language can result in a poor system design. Environments that are well suited to making rapid progress during the early stages of development are sometimes poorly suited to maintaining design integrity in the later stages of development.

CROSS-REFERENCE
For more on pushing the envelope with rapid-development languages, see Section 28.3, "Managing Risks of RDLs."

You can mitigate design-quality risks by striking at the root of the problem. Limit the scope of the prototype to specific areas of the product, for example, to the user interface. Develop this limited part of the product using a prototyping language, and evolve it; then develop the rest of the product using a traditional programming language and nonprototyping approaches.

When creating the overall system design, make a conscious effort not to overemphasize the user interface or any other aspect of the system that has been prototyped. Minimize the impact that design in one area can have on the design as a whole.

Include a design phase with each iteration of the prototype to ensure that you evolve the design and not just the code. Use a design checklist at each stage to check the quality of the evolving product.

CLASSIC MISTAKE

Avoid using inexperienced developers on a prototyping project. Evolutionary Prototyping requires that developers make far-reaching design decisions much earlier in the development process than when other development approaches are used. Inexperienced developers are often poorly equipped to make good design decisions under such circumstances. Case studies of prototyping projects have reported projects that have failed because of the involvement of inexperienced developers. They have also reported projects that have succeeded only because of the involvement of experienced developers (Gordon and Bieman 1995).

Poor maintainability. Evolutionary Prototyping sometimes contributes to poor maintainability. The high degree of modularity needed for effective Evolutionary Prototyping is conducive to maintainable code. But when the prototype is developed extremely rapidly, it is also sometimes developed extremely sloppily—Evolutionary Prototyping can be used as a disguise for code-and-fix development. Moreover, if a special-purpose prototyping language is used, the language may be unfamiliar to maintenance programmers, which will make the program difficult to maintain. In the published literature on prototyping, more sources than not have observed worse maintainability with Evolutionary Prototyping than with traditional approaches (Gordon and Bieman 1995).

Minimize the maintainability risk by taking a few simple steps. Be sure that the design is adequate by using the guidelines listed in the discussion of the "Poor design" risk (in the preceding section). Use code-quality checklists at each stage to keep the prototype's code quality from slipping. Follow normal naming conventions for objects, methods, functions, data, database elements, libraries, and any other components that will be maintained as the prototype matures. Follow other coding conventions for good layout and commenting. Remind your team that they will have to use their own code throughout multiple product-release cycles—that will give them a strong incentive to make their code maintainable and to keep it that way.

CROSS-REFERENCE
For details on change management, see Chapter [illegible] "Feature-Set Control."

Feature creep. Customers and end-users typically have direct access to the prototype during an evolutionary-prototyping project, and that can sometimes lead to an increased desire for features. In addition to managing their interactions with the prototype, use normal change-management practices to control feature creep.

CLASSIC MISTAKE

Inefficient use of prototyping time. Prototyping is usually intended to shorten the development schedule. Paradoxically, projects often waste time during prototyping and don't save as much time as they could. Prototyping is an exploratory, iterative process, so it is tempting to treat it as a process that can't be managed carefully. Developers don't always know how far to develop a prototype, so they sometimes waste time fleshing out features that are summarily excluded from the product later. Developers sometimes give the prototype a robustness it doesn't need or waste time working on the prototype without moving it in any clear direction.

To spend prototyping time effectively, make project tracking and control a priority. Carefully schedule prototyping activities. Later in the project you might break the project into chunks of a few days or weeks, but during prototyping break it into hours or days. Be sure that you don't put a lot of work into prototyping and evolving any part of the system until you're certain it's going to remain part of the system.

Developers need to have a well-developed sense of how little they can do to investigate the risky areas a prototype is intended to explore. On one project I worked on, we needed to create a user-interface prototype to demonstrate our user-interface design to our sponsor. One of the other developers thought it would take a team of three developers 6 weeks to build the prototype. I thought I could do all the work myself in about 1 week, which turned out to be accurate. The difference in our estimates didn't arise because I was six times as fast as three developers; they arose because the other developer didn't have a good sense of all the things that could be left out of that kind of a prototype. Ergo, avoid using inexperienced developers on prototyping projects.

21.3 Side Effects of Evolutionary Prototyping

In addition to its rapid-development benefits, Evolutionary Prototyping produces many side effects, most of which are beneficial. Prototyping tends to lead to:

- Improved morale of end-users, customers, and developers because progress is visible
- Early feedback on whether the final system will be acceptable
- Decreased overall code length because of better designs and more reuse
- Lower defect rates because of better requirements definition
- Smoother effort curves, reducing the deadline effect (which is common when using traditional development approaches)

21.4 Evolutionary Prototyping's Interactions with Other Practices

HARD DATA

Evolutionary Prototyping is an effective remedy for feature creep when it is used with other practices. One study found that the combination of prototyping and JAD (Chapter 24) was capable of keeping creeping requirements below 5 percent. Average projects experience levels of about 25 percent (Jones 1994).

Evolutionary Prototyping is also an effective defect-removal practice when combined with other practices. A combination of reviews, inspections, and testing produce a defect-removal strategy with the highest defect-removal efficacy, lowest cost, and shortest schedule. Adding prototyping to the mix produces a defect-removal strategy with the highest cumulative defect-removal efficacy (Pfleeger 1994a).

If Evolutionary Prototyping provides less control than you need or you already know fundamentally what you want the system to do, you can use Evolutionary Delivery (Chapter 20) or Staged Delivery (Chapter 36) instead.

Relationship to Other Kinds of Prototyping

On small systems, Evolutionary Prototyping is preferable to Throwaway Prototyping because the overhead of creating a throwaway prototype makes it economically unfeasible. If you're considering developing a throwaway prototype on a small-to-medium–size project, be sure that you can recoup the overhead over the life of the project.

On large systems (systems with more than 100,000 lines of code), you can use either Throwaway Prototyping or Evolutionary Prototyping. Developers are sometimes leery of using Evolutionary Prototyping on large systems, but the published reports on the use of Evolutionary Prototyping on large systems have all reported success (Gordon and Bieman 1995).

21.5 The Bottom Line on Evolutionary Prototyping

HARD DATA

In case studies, Evolutionary Prototyping has decreased development effort dramatically, by from 45 to 80 percent (Gordon and Bieman 1995). To achieve this kind of reduction, you must manage development risks carefully, particularly the risks of poor design, poor maintenance, and feature creep. If you don't do that, overall effort can actually increase.

Prototyping yields its benefits quickly. Capers Jones estimates that Evolutionary Prototyping will return roughly two dollars for every dollar spent within the first year, and scaling up to roughly ten dollars for every dollar spent within four years (Jones 1994).

Prototyping also produces steady, visible signs of advancement, improving progress visibility and contributing to the impression that you're developing rapidly. That can be especially useful when the demand for development speed is strong.

The up-front cost of initiating an evolutionary-prototyping program is low—only the costs needed for developer training and prototyping tools.

21.6 Keys to Success in Using Evolutionary Prototyping

Here are the keys to successful Evolutionary Prototyping:

- Decide at the beginning of the project whether to evolve the prototype or throw it away. Be sure that managers and developers are both committed to whichever course of action is selected.
- Explicitly manage customer and end-user expectations having to do with schedule, budget, and performance.
- Limit end-user interaction with the prototype to controlled settings.
- Use experienced developers. Avoid using entry-level developers.
- Use design checklists at each stage to ensure the quality of the prototyped system.
- Use code-quality checklists at each stage to ensure the maintainability of the prototyped code.
- Consider performance early.
- Carefully manage the prototyping activity itself.
- Consider whether Evolutionary Prototyping will provide the most benefit or whether Evolutionary Delivery or Staged Delivery would be better.

Further Reading

Gordon, V. Scott, and James M. Bieman. "Rapid Prototyping: Lessons Learned," *IEEE Software*, January 1995, 85–95. This article summarizes the lessons learned from 22 published case studies of prototyping and several additional anonymous case studies.

Connell, John, and Linda Shafer. *Object-Oriented Rapid Prototyping*. Englewood Cliffs, N.J.: Yourdon Press, 1995. This book overviews the reasons that you would prototype and different prototyping styles. It then presents an intelligent, full discussion of evolutionary, object-oriented prototyping including the topics of design guidelines, tools, prototype refinement, and prototyping-compatible lifecycle models.