CHAPTER

4 Recommended Process Model

In Chapters 2 and 3, we provided brief descriptions of several software process models and software engineering frameworks. Every project is different, and every team is different. There is no single software engineering framework that is appropriate for every software product. In this chapter, we'll share our thoughts on using an adaptable process that can be tailored to fit the needs of software developers working on many types of products.

Key Concepts

defining requirements 57
estimating resources 60
evaluating prototype. 64
go, no-go decision. 65
maintain. 69
preliminary architectural design 59
prototype construction 61
prototype evolution. 67
release candidate 68
risk assessment. 66
scope definition. 67
test and evaluate. 63

Quick Look

What is it? Every software product needs a "road map" or "generic software process" of some kind. It doesn't have to be complete before you start, but you need to know where you're headed before you begin. Any road map or generic process should be based on best industry practices.

Who does it? Software engineers and their product stakeholders collaborate to adapt a generic software process model to meet the needs of team and then either follow it directly, or more likely, adapt as needed. Every software team should be disciplined but flexible and self-empowered when needed.

Why is it important? Software development can easily become chaotic without the control and organization offered by a defined process. As we stated in Chapter 3, a modern software engineering approach must be "agile" and embrace changes that are needed to satisfy the stakeholders' requirements. It is important not to be too focused on documents and rituals. The process should only include those activities, controls, and work products that are appropriate for the project team and the product that is to be produced.

What are the steps? Even if a generic process must be adapted to meet the needs of the specific products being built, you need to be sure that all stakeholders have a role to play in defining, building, and testing the evolving software. There is likely to be substantial overlap among the basic framework activities (communication, planning, modeling, construction, and deployment). *Design a little, build a little, test a little, repeat* is a better approach than creating rigid project plans and documents for most software projects.

What is the work product? From the point of view of a software team, the work products are working program increments, useful documents, and data that are produced by the process activities.

How do I ensure that I've done it right? The timeliness, levels of stakeholder satisfaction, overall quality, and long-term viability of the product increments built are the best indicators that your process is working.

A paper by Rajagoplan [Raj14] reviews the general weaknesses of prescriptive software life-cycle approaches (e.g., the waterfall model) and contains several suggestions that should be considered when organizing a modern software development project.

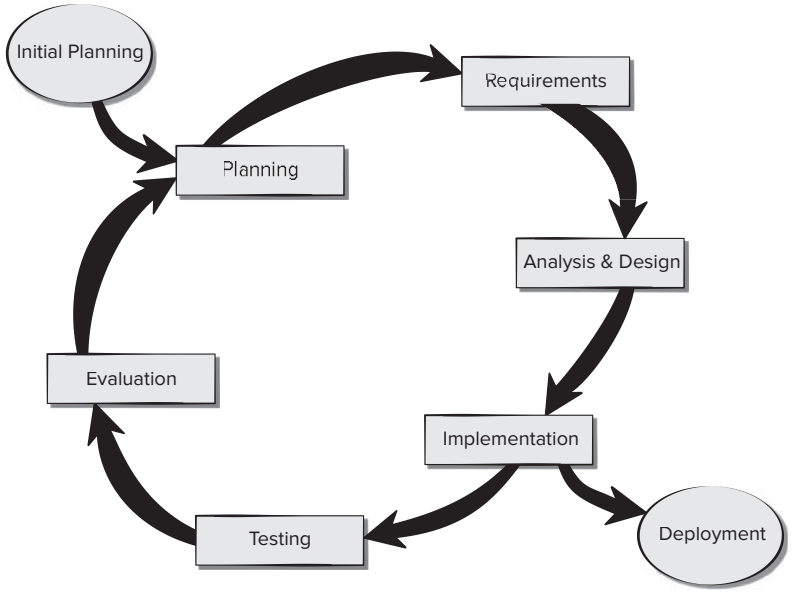
1. It is risky to use a linear process model without ample feedback.
2. It is never possible nor desirable to plan big up-front requirements gathering.
3. Up-front requirements gathering may not reduce costs or prevent time slippage.
4. Appropriate project management is integral to software development.
5. Documents should evolve with the software and should not delay the start of construction.
6. Involve stakeholders early and frequently in the development process.
7. Testers need to become involved in the process prior to software construction.

In Section 2.6, we listed the pros and cons of several prescriptive process models. The waterfall model is not amenable to changes that may need to be introduced once developers start coding. Stakeholder feedback is therefore limited to the beginning and end of the project. Part of the reason for this is the waterfall model suggests that all analysis and design work products be completed before any programming or testing occurs. This makes it hard to adapt to projects with evolving requirements.

One temptation is to switch to an incremental model (Figure 4.1) like the prototyping model or Scrum. Incremental process models involve customers early and often and therefore reduce the risk of creating product that is not accepted by the customers. There is a temptation to encourage lots of changes as stakeholders view each prototype and realize that functions and features they now realize they need are missing. Often, developers do not plan for prototype evolution and create throwaway prototypes.

FIGURE 4.1

Incremental model for prototype development



Recall that the goal of software engineering is to reduce unnecessary effort, so prototypes need to be designed with reuse in mind. Incremental models do provide a better basis for creating an adaptable process if changes can be managed wisely.

In Section 3.5, we discussed the pros and cons of several agile process models other than Scrum. Agile process models are very good at accommodating the uncertain knowledge about the stakeholders' real needs and problems. Key characteristics of agile process models are:

- Prototypes created are designed to be extended in future software increments.
- Stakeholders are involved throughout the development process.
- Documentation requirements are lightweight, and documentation should evolve along with the software.
- Testing is planned and executed early.

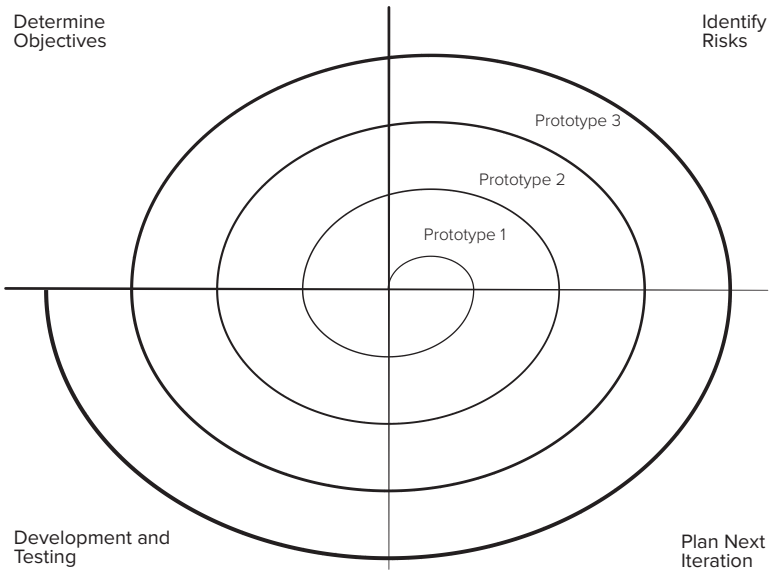
Scrum and Kanban extend these characteristics. Scrum is sometimes criticized for requiring too many meetings. But daily meetings make it hard for developers to stray too far from building products that stakeholders find useful. Kanban (Section 3.5.2) provides a good lightweight tracking system for managing the status and priorities of user stories.

Both Scrum and Kanban allow for controlled introduction of new requirements (user stories). Agile teams are small by design and may not be suitable for projects requiring large numbers of developers, unless the project can be partitioned into small and independently assignable components. Still, agile process models offer many good features that can be incorporated into an adaptable process model.

The spiral model (Figure 4.2) can be thought of as an evolutionary prototyping model with a risk assessment element. The spiral model relies on moderate stakeholder

FIGURE 4.2

Spiral model for prototype development



involvement and was designed for large teams and large projects. Its goal is to create extensible prototypes each time the process is iterated. Early testing is essential. Documentation evolves with the creation of each new prototype. The spiral model is somewhat unique in that formal risk assessment is built in and used as the basis for deciding whether to invest the resources needed to create the next prototype. Some people argue that it may be hard to manage a project using the spiral model, because the project scope may not be known at the start of the project. This is typical of most incremental process models. The spiral is a good basis for building an adaptable process model.

How do agile process models compare to evolutionary models? We've summarized some of the key characteristics in a sidebar.

Characteristics of Agile Models

Agile

1. Not suitable for large high-risk or mission critical projects.
2. Minimal rules and minimal documentation
3. Continuous involvement of testers
4. Easy to accommodate product changes
5. Depends heavily on stakeholder interaction
6. Easy to manage
7. Early delivery of partial solutions
8. Informal risk management
9. Built-in continuous process improvement

Spiral

1. Not suitable for small, low-risk projects
2. Several steps required, along with documentation done up front
3. Early involvement of testers (might be done by outside team)
4. Hard to accommodate product changes until prototype completed
5. Continuous stakeholder involvement in planning and risk assessment
6. Requires formal project management and coordination
7. Project end not always obvious
8. Good risk management
9. Process improvement handled at end of project

Creative, knowledgeable people perform software engineering. They adapt software processes to make them appropriate for the products that they build and to meet the demands of the marketplace. We think that using a spiral-like approach that has agility built into every cycle is a good place to start for many software projects. Developers learn many things as they proceed in the development process. That's why it is important for developers to be able to adapt their process as quickly as practical to accommodate this new knowledge.

4.1 Requirements Definition

Every software project begins with the team trying to understand the problem to be solved and determining what outcomes are important to the stakeholders. This includes understanding the business needs motivating the project and the technical issues which constrain it. This process is called *requirements engineering* and will be discussed in

more detail in Chapter 7. Teams that fail to spend a reasonable amount of time on this task will find that their project contains expensive rework, cost overruns, poor product quality, late delivery times, dissatisfied customers, and poor team morale. Requirements engineering cannot be neglected, nor can it be allowed to iterate endlessly before proceeding to product construction.

It's reasonable to ask what best practices should be followed to achieve thorough and agile requirements engineering. Scott Ambler [Amb12] suggests several best practices for agile requirements definition:

1. Encourage active stakeholder participation by matching their availability and valuing their input.
2. Use simple models (e.g., Post-it notes, fast sketches, user stories) to reduce barriers to participation.
3. Take time to explain your requirement representation techniques before using them.
4. Adopt stakeholder terminology, and avoid technical jargon whenever possible.
5. Use a breadth-first approach to get the big picture of the project done before getting bogged down in details.
6. Allow the development team to refine (with stakeholder input) requirement details "just in time" as user stories are scheduled to be implemented.
7. Treat the list of features to be implemented like a prioritized list, and implement the most important user stories first.
8. Collaborate closely with your stakeholders and only document requirements at a level that is useful to all when creating the next prototype.
9. Question the need to maintain models and documents that will not be referred to in the future.
10. Make sure you have management support to ensure stakeholder and resource availability during requirements definition.

It is important to recognize two realities: (1) it is impossible for stakeholders to describe an entire system before seeing the working software, and (2) it is difficult for stakeholders to describe quality requirements needed for the software before seeing it in action. Developers must recognize that requirements will be added and refined as the software increments are created. Capturing stakeholders' descriptions about what the system needs to do in their own words in a user story is a good place to begin.

If you can get stakeholders to define acceptance criteria for each user story, your team is off to a great start. It is likely that stakeholders will need to see a user story coded and running to know whether it has been implemented correctly or not. Therefore, requirements definition needs to be done iteratively and include the development of prototypes for stakeholder review.

Prototypes are tangible realizations of project plans that can be easily referenced by stakeholders when trying to describe desired changes. Stakeholders are motivated to discuss requirements changes in more concrete terms, which improves communication. It's important to recognize that prototypes allow developers to focus on short-term goals by only focusing on users' visible behaviors. It will be important to review

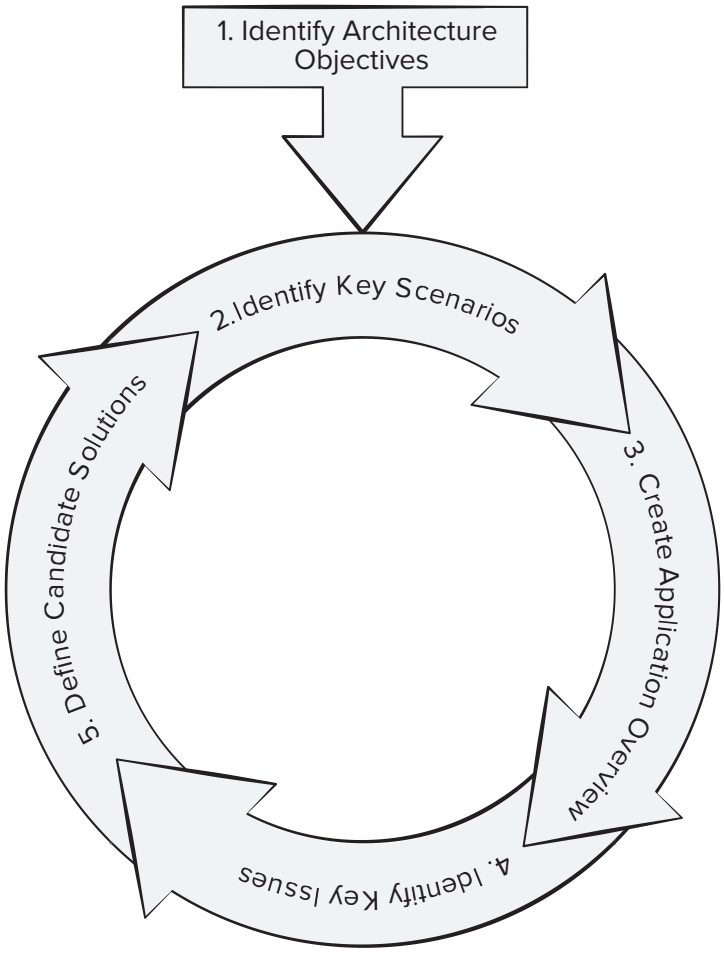
prototypes with an eye to the quality. Developers need to be aware that using prototypes may increase the volatility of the requirements if stakeholders are not focused on getting things right the first time. There is also the risk that creating prototypes before the software architectural requirements are well understood may result in prototypes that must be discarded, wasting time and resources [Kap15].

4.2 PRELIMINARY ARCHITECTURAL DESIGN

The decisions required to develop a solid architectural design are discussed in Chapter 10, but preliminary design decisions must often be made as requirements are defined. As shown in Figure 4.3, at some point in time, architectural decisions will need to be allocated to product increments. According to Bellomo and her colleagues [Bel14], early understanding of requirements and architecture choices is key to managing the development of large or complex software products.

FIGURE 4.3

Architectural design for prototype development



Requirements can be used to inform architecture design. Exploring the architecture as the prototype is developed facilitates the process of detailing the requirements. It is best to conduct these activities concurrently to achieve the right balance. There are four key elements to agile architectural design:

1. Focus on key quality attributes, and incorporate them into prototypes as they are constructed.
2. When planning prototypes, keep in mind that successful software products combine customer-visible features and the infrastructure to enable them.
3. Recognize that an agile architecture enables code maintainability and evolvability if sufficient attention is paid to architectural decisions and related quality issues.
4. Continuously managing and synchronizing dependencies between the functional and architectural requirements is needed to ensure the evolving architectural foundation will be ready just in time for future increments.

Software architecture decision making is critical to the success of a software system. The architecture of a software system determines its qualities and impacts the system throughout its life cycle. Dasanayake et al. [Das15] found that software architects are prone to making errors when their decisions are made under levels of uncertainty. Architects make fewer bad decisions if they can reduce this uncertainty through better architectural knowledge management. Despite the fact that agile approaches discourage heavy documentation, failing to record design decisions and their rationale early in the design process makes it hard to revisit them when creating future prototypes. Documenting the right things can assist with process improvement activities. Documenting your lessons learned is one of the reasons that retrospectives should be conducted after evaluating the delivered prototype and before beginning the next program increment. Reuse of previously successful solutions to architectural problems is also helpful and will be discussed in Chapter 14.

4.3 Resource Estimation

One of the more controversial aspects of using spiral or agile prototyping is estimating the time it will take to complete a project when it cannot be defined completely. It is important to understand before you begin whether you have a reasonable chance of delivering software products on time and with acceptable costs before you agree to take on the project. Early estimates run the risk of being incorrect because the project scope is not well defined and is likely to change once development starts. Estimates made when the project is almost finished do not provide any project management guidance. The trick is to estimate the software development time early based on what is known at the time and revise your estimates on a regular basis as requirements are added or after software increments are delivered. We discuss methods of estimating project scope in Chapter 25.

Let's examine how an experienced software project manager might estimate a project using the agile spiral model we have proposed. The estimates produced by this

method would need to be adjusted for the number of developers and the number of user stories that can be completed simultaneously.

1. Use historic data (Chapter 23), and work as a team to develop an estimate of how many days it will take to complete each of the user stories known at the start of the project.
2. Loosely organize the user stories into the sets that will make up each sprint[1] (Section 3.4) planned to complete a prototype.
3. Sum the number of days to complete each sprint to provide an estimate for the duration of the total project.
4. Revise the estimate as requirements are added to the project or prototypes are delivered and accepted by the stakeholders.

Keep in mind that doubling the number of developers almost never cuts the development time in half.

Rosa and Wallshein [Ros17] found that knowing initial software requirements at the start of a project provides an adequate but not always accurate estimate of project completion times. To get more accurate estimates, it is also important to know the type of project and the experience of the team. We will describe more detailed estimation techniques (e.g., function points or use case points) in Part Four of this book.

4.4 First Prototype Construction

In Section 2.5.2 we described the creation of prototypes as a means of helping the stakeholders move from statements of general objectives and user stories to the level of detail that developers will need to implement this functionality. Developers may use the first prototype to prove that their initial architectural design is a feasible approach to delivering the required functionality while satisfying the customer's performance constraints. To create an operational prototype suggests that requirements engineering, software design, and construction all proceed in parallel. This process is shown in Figure 4.1. This section describes steps that will be used to create the first prototypes. Details of best practices for software design and construction appear later in this book.

Your first task is to identify the features and functions that are most important to the stakeholders. These will help define the objectives for the first prototype. If the stakeholders and developers have created a prioritized list of user stories, it should be easy to confirm which are the most important.

Next, decide how much time will be allowed to create the first prototype. Some teams may choose a fixed time, such as a 4-week sprint, to deliver each prototype. In this case, the developers will look at their time and resource estimate and determine which of the high-priority user stories can be finished in 4 weeks. The team would then confirm with the stakeholders that the selected user stories are the best ones to include in the first prototype. An alternative approach would be to have the stakeholders

1 Sprint was described (Section 3.4) as a time period in which a subset of the system user stories will be delivered to the product owner.

and developers jointly choose a small number of high-priority user stories to include in the first prototype and use their time and resource estimates to develop the schedule to complete the first prototype.

The engineers working at National Instruments published a white paper that outlines their process for creation of a first functional prototype [Nat15]. These steps can be applied to a variety of software projects:

1. Transition from paper prototype to software design
2. Prototype a user interface
3. Create a virtual prototype
4. Add input and output to your prototype
5. Engineer your algorithms
6. Test your prototype
7. Prototype with deployment in mind

Referring to these seven steps, creating a *paper prototype* for a system is very inexpensive and can be done early in the development process. Customers and stakeholders are not usually experienced developers. Nontechnical users can often recognize what they like or do not like about a user interface very quickly once they see it sketched out. Communications between people are often filled with misunderstandings. People forget to tell each other what they really need to know or assume that everyone has the same understanding. Creating a paper prototype and reviewing it with the customer before doing any programming can help avoid wasted time building the wrong prototype. We will talk about several diagrams that can be used to model a system in Chapter 8.

Creating a *prototype user interface* as part of the first functional prototype is a wise idea. Many systems are implemented on the Web or as mobile applications and rely heavily on touch user interfaces. Computer games and virtual reality applications require a great deal of communication with end users to operate correctly. If customers find a software product easy to learn and use, they are more likely to use it.

Many misunderstandings between developers and stakeholders can be alleviated by beginning with a paper prototype of the user interface. Sometimes stakeholders need to see the basics of the user interface in action to be able to explain what they really like and dislike about it. It is less expensive to throw away an early user interface design than to finish the prototype and try to put a new user interface on top of it. Designing user interfaces that provide good user experiences is discussed in Chapter 12.

Adding input and output to your user interface prototype provides an easy way to begin testing the evolving prototype. Testing software component interfaces should be accomplished before testing the code that makes up the component's algorithms. To test the algorithms themselves, developers often use a "test frame" to ensure their implemented algorithms are working as intended. Creating a separate test frame and throwing it away is often a poor use of project resources. If properly designed, the user interface can serve as the test frame for component algorithms, thereby eliminating the effort required to build separate test frames.

Engineering your algorithms refers to the process of transforming your ideas and sketches into programming language code. You need to consider both the functional

requirements stated in the user story and the performance constraints (both explicit and implicit) when designing the necessary algorithms. This is the point where additional support functionality is likely to be identified and added to the project scope, if it does not already exist in a code library.

Testing your prototype demonstrates required functionality and identifies yet undiscovered defects before demonstrating it to the customer. Sometimes it is wise to involve the customer in the testing process before the prototype is finished to avoid developing the wrong functionality. The best time to create test cases is during requirements gathering or when use cases have been selected for implementation. Testing strategies and tactics are discussed in Chapters 19 through 21.

Prototyping with deployment in mind is very important because it helps you to avoid taking shortcuts that lead to creating software that will be hard to maintain in the future. This is not saying that every line of code will make it to the final software product. Like many creative tasks, developing a prototype is iterative. Drafts and revisions are to be expected.

As prototype development occurs, you should carefully consider the software architectural choices you make. It is relatively inexpensive to change a few lines of code if you catch errors before deployment. It is very expensive to change the architecture of a software application once it has been released to end users around the globe.

SafeHome

Room Designer
Considering First Prototype

The scene: Doug Miller's office.

The players: Doug Miller, software engineering manager; Jamie Lazar, software team member; Vinod Raman, software team member.

The conversation: (A knock on the door. Jamie and Vinod enter Doug's office.)

Jamie: Doug, you got a minute?

Doug: Sure, Jamie, what's up?

Jamie: We've been thinking about the scope of this *SafeHome* room design tool.

Doug: And?

Vinod: There's a lot of work that needs to be done on the back end of the project before people can start dropping alarm sensors and trying furniture layouts.

Jamie: We don't want to work on the back end for months and then have the project cancelled when the marketing folks decide they hate the product.

Doug: Have you tried to work out a paper prototype and review it with the marketing group?

Vinod: Well, no. We thought it was important to get a working computer prototype quickly and did not want to take the time to do one.

Doug: My experience is that people need to see something before they know whether they like it or not.

Jamie: Maybe we should step back and create a paper prototype of the user interface and get them to work with it and see if they like the concept at all.

Vinod: I suppose it wouldn't be too hard to program an executable user interface using the game engine we were considering using for the virtual reality version of the app.

Doug: Sounds like a plan. Try that approach, and see if you have the confidence you need to start evolving your prototype.

4.5 PROTOTYPE EVALUATION

Testing is conducted by the developers as the prototype is being built and becomes an important part of prototype evaluation. Testing demonstrates that prototype components are operational, but it's unlikely that test cases will have found all the defects. In the spiral model, the results of evaluation allow the stakeholders and developers to assess whether it is desirable to continue development and create the next prototype. Part of this decision is based on user and stakeholder satisfaction, and part is derived from an assessment of the risks of cost overruns and failure to deliver a working product when the project is finished. Dam and Siang [Dam17] suggest several best-practice tips for gathering feedback on your prototype.

1. Provide scaffolding when asking for prototype feedback.
2. Test your prototype on the right people.
3. Ask the right questions.
4. Be neutral when presenting alternatives to users.
5. Adapt while testing.
6. Allow the user to contribute ideas.

Providing scaffolding is a mechanism for allowing the user to offer feedback that is not confrontational. Users are often reluctant to tell developers that they hate the product they are using. To avoid this, it is often easier to ask the user to provide feedback using a framework such as "I like, I wish, What if" as a means of providing open and honest feedback. *I like* statements encourage users to provide positive feedback on the prototype. *I wish* statements prompt users to share ideas about how the prototype can be improved. These statements can provide negative feedback and constructive criticism. *What if* statements encourage users to suggest ideas for your team to explore when creating prototypes in future iterations.

Getting the right people to evaluate the prototype is essential to reduce the risk of developing the wrong product. Having development team members do all the testing is not wise because they are not likely to be the representative of the intended user population. It's important to have the right mix of users (e.g., novice, typical, and advanced) to give you feedback on the prototype.

Asking the right questions implies that all stakeholders agree on prototype objectives. As a developer, it's important to keep an open mind and do your best to convince users that their feedback is valuable. Feedback drives the prototyping process as you plan for future product development activities. In addition to general feedback, try to ask specific questions about any new features included in the prototype.

Be neutral when presenting alternatives allows the software team to avoid making users feel they are being "sold" on one way to do things. If you want honest feedback, let the users know that you have not already made up your mind that there is only one right way to do things. *Egoless programming* is a development philosophy that focuses on producing the best product the team can create for the intended users. Although it is not desirable to create throwaway prototypes, egoless programming suggests that things that are not working need to be fixed or

discarded. So try not to become too attached to your ideas when creating early prototypes.

Adapt while testing means that you need a flexible mind-set while users are working with the prototype. This might mean altering your test plan or making quick changes to the prototype and then restarting the testing. The goal is to get the best feedback you can from users, including direct observation of them as they interact with the prototype. The important thing is that you get the feedback you need to help decide whether to build the next prototype or not.

Allow users to contribute ideas means what it says. Make sure you have a way of recording their suggestions and questions (electronically or otherwise). We will discuss additional ways of conducting user testing in Chapter 12.

SAFEHOME

Room Designer
Evaluating First Prototype

The scene: Doug Miller's office.

The players: Doug Miller, software engineering manager; Jamie Lazar, software team member; Vinod Raman, software team member.

The conversation: (A knock on the door. Jamie and Vinod enter Doug's office.)

Jamie: Doug, you got a minute?

Doug: Sure, Jamie, what's up?

Jamie: We completed the evaluation of the *SafeHome* room design tool working with our marketing stakeholders.

Doug: How did things go?

Vinod: We mostly focused on the user interface that will allow users to place alarm sensors in the room.

Jamie: I am glad we let them review a paper prototype before we created the PC prototype.

Doug: Why is that?

Vinod: We made some changes, and the marketing people liked the new design better, so that's the design we used when we started programming it.

Doug: Good. What's the next step?

Jamie: We completed the risk analysis and since we did not pick up any new user stories, we think it is reasonable to work on creating the next incremental prototype since we are still on time and within budget.

Vinod: So, if you agree, we'll get the developers and stakeholders together and begin planning the next software increment.

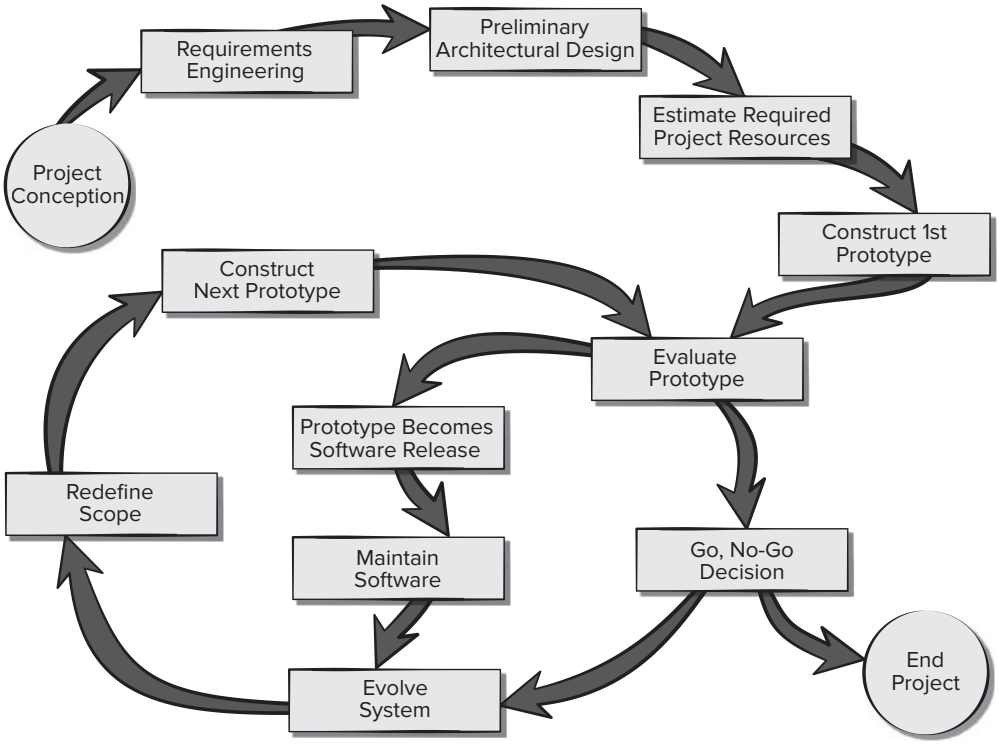
Doug: I agree. Keep me in the loop and try to keep the development time on the next prototype to 6 weeks or less.

4.6 GO, NO-GO DECISION

After the prototype is evaluated, project stakeholders decide whether to continue development of the software product. If you refer to Figure 4.4, a slightly different decision based on the prototype evaluation might be to release the prototype to the

FIGURE 4.4

Recommended software process model



end users and begin the maintenance process. Sections 4.8 and 4.9 discuss this decision in more detail. We discussed the use of risk assessment in the spiral model as part of the prototype assessment process in Section 2.5.3. The first circuit around the spiral might be used to solidify project requirements. But it really should be more. In the method we are proposing here, each trip around the spiral develops a meaningful increment of the final software product. You can work with the project user story or feature backlog to identify an important subset of the final product to include in the first prototype and repeat this for each subsequent prototype.

A pass through the planning region follows the evaluation process. Revised cost estimates and schedule changes are proposed based on what was discovered when evaluating the current software prototype. This may involve adding new user stories or features to the project backlog as the prototype is evaluated. The risk of exceeding the budget and missing the project delivery date is assessed by comparing new cost and time estimates against old ones. The risk of failing to satisfy user expectations is also considered and discussed with the stakeholders and sometimes senior management before deciding to create another prototype.

The goal of the risk assessment process is to get the commitment of all stakeholders and company management to provide the resources needed to create the next prototype. If the commitment is not there because the risk of project failure is too great, then the project can be terminated. In the Scrum framework (Section 3.4), the go, no-go decision might be made during the Scrum retrospective

meeting held between the prototype demonstration and the new sprint planning meeting. In all cases, the development team lays out the case for the product owners and lets them decide whether to continue product development or not. A more detailed discussion of software risk assessment methods is presented in Chapter 26.

4.7 PROTOTYPE EVOLUTION

Once a prototype has been developed and reviewed by the development team and other stakeholders, it's time to consider the development of the next prototype. The first step is to collect all feedback and data from the evaluation of the current prototype. The developers and stakeholders then begin negotiations to plan the creation of another prototype. Once the desired features for the new prototype have been agreed upon, consideration is given to any known time and budget constraints as well as the technical feasibility of implementing the prototype. If the development risks are deemed to be acceptable, the work continues.

The evolutionary prototyping process model is used to accommodate changes that inevitably occur as software is developed. Each prototype should be designed to allow for future changes to avoid throwing it away and creating the next prototype from scratch. This suggests favoring both well-understood and important features when setting the goals for each prototype. As always, the customer's needs should be given great importance in this process.

4.7.1 New Prototype Scope

The process of determining the scope of a new prototype is like the process of determining the scope of the initial prototype. Developers would either: (1) select features to develop within the time allocated to a sprint, or (2) allocate sufficient time to implement the features needed to satisfy the goals set by the developers with stakeholder input. Either approach requires the developers to maintain a prioritized list of features or user stories. The priorities used to order the list should be determined by the goals set for the prototype by the stakeholders and developers.

In XP (Section 3.5.1), the stakeholders and developers work together to group the most important user stories into a prototype that will become the next release of the software and determine its completion date. In Kanban (Section 3.5.2), developers and stakeholders make use of a board that allows them to focus on the completion status of each user story. This is a visual reference that can be used to assist developers using any incremental prototype process model to plan and monitor the progress of the software development. Stakeholders can easily see the feature backlog and help the developers order it to identify the most useful stories needed in the next prototype. It is probably easier to estimate the time required to complete the selected user stories than to find the user stories that need to fit into a fixed time block. But the advice to keep prototype development time to 4 to 6 weeks should be followed to ensure adequate stakeholder involvement and feedback.

4.7.2 Constructing New Prototypes

A user story should contain both a description of how the customer plans to interact with the system to achieve a specific goal and a description of what the customer's definition of acceptance is. The development team's task is to create additional software components to implement the user stories selected for inclusion in the new prototype along with the necessary test cases. Developers need to continue communication with all stakeholders as they create the new prototype.

What makes this new prototype trickier to build is that software components created to implement new features in the evolving prototype need to work with the components used to implement the features included in the previous prototype. It gets even trickier if developers need to remove components or modify those that were included in the previous prototype because requirements have changed. Strategies for managing these types of software changes are discussed in Chapter 22.

It is important for the developers to make design decisions that will make the software prototype more easily extensible in the future. Developers need to document design decisions in a way that will make it easier to understand the software when making the next prototype. The goal is to be agile in both development and documentation. Developers need to resist the temptation to overdesign the software to accommodate features that may or may not be included in the final product. They also should limit their documentation to that which they need to refer to during development or when changes need to be made in the future.

4.7.3 Testing New Prototypes

Testing the new prototype should be relatively straightforward if the development team created test cases as part of the design process before the programming was completed. Each user story should have had acceptance criteria attached to it as it was created. These acceptance statements should guide the creation of the test cases intended to help verify that the prototype meets customer needs. The prototype will need to be tested for defects and performance issues as well.

One additional testing concern for evolutionary prototypes is to ensure that adding new features does not accidentally break features that were working correctly in the previous prototype. *Regression testing* is the process of verifying that software that was previously developed and tested still performs the same way after it has been changed. It is important to use your testing time wisely and make use of the test cases that are designed to detect defects in the components most likely to be affected by the new features. Regression testing is discussed in more detail in Chapter 20.

4.8 Prototype Release

When an evolutionary prototyping process is applied, it can be difficult for developers to know when a product is finished and ready for release to the customers. Software developers do not want to release a buggy software product to the end users and have them decide the software has poor quality. A prototype being considered as a release candidate must be subjected to user acceptance testing in addition to functional and nonfunctional (performance) testing that would have been conducted during prototype construction.

User acceptance tests are based on the agreed-upon acceptance criteria that were recorded as each user story was created and added to the product backlog. This allows

user representatives to verify that the software behaves as expected and collect suggestions for future improvements. David Nielsen [Nie10] has several suggestions for conducting prototype testing in industrial settings.

When testing a release candidate, functional and nonfunctional tests should be repeated using the test cases that were developed during the construction phases of the incremental prototypes. Additional nonfunctional tests should be created to verify that the performance of the prototype is consistent with the agreed-upon benchmarks for the final product. Typical performance benchmarks may deal with system response time, data capacity, or usability. One of the most important nonfunctional requirements to verify is ensuring that the release candidate will run in all planned run-time environments and on all targeted devices. The process should be focused on testing limited to the acceptance criteria established before the prototype was created. Testing cannot prove a software product is bug free, only that the test cases ran correctly.

User feedback during acceptance testing should be organized by user-visible functions as portrayed via the user interface. Developers should examine the device in question and make changes to the user interface screen if implementing these changes will not delay the release of the prototype. If changes are made, they need to be verified in a second round of testing before moving on. You should not plan for more than two iterations of user acceptance testing.

It is important, even for projects using agile process models, to use an issue tracking or bug reporting system (e.g., Bugzilla[2] or Jira[3]) to capture the testing results. This allows developers to record test failures and makes it easier to identify the test cases that will need to be run again to verify that a repair properly corrects the problem that was uncovered. In each case the developers need to assess whether the changes can be made to the software without causing a cost overrun or late product delivery. The implications of not fixing a problem need to be documented and shared with both the customer and senior managers who may decide to cancel the project rather than committing the resources needed to deliver the final project.

The issues and lessons learned from creating the release candidate should be documented and considered by the developers and stakeholders as part of the project postmortem. This information should be considered before deciding to undertake future development of a software product following its release to the user community. The lessons learned from the current product can help developers make better cost and time estimates for similar projects in the future.

Techniques for conducting user acceptance testing are discussed in Chapters 12 and 20. A more detailed discussion on software quality assurance is presented in Chapter 17.

4.9 Maintain Release Software

Maintenance is defined as the activities needed to keep software operational after it has been accepted and delivered (released) in the end-user environment. Maintenance will continue for the life of the software product. Some software engineers believe that the majority of the money spent on a software product will be spent on maintenance activities. *Corrective maintenance* is the reactive modification of software to repair problems

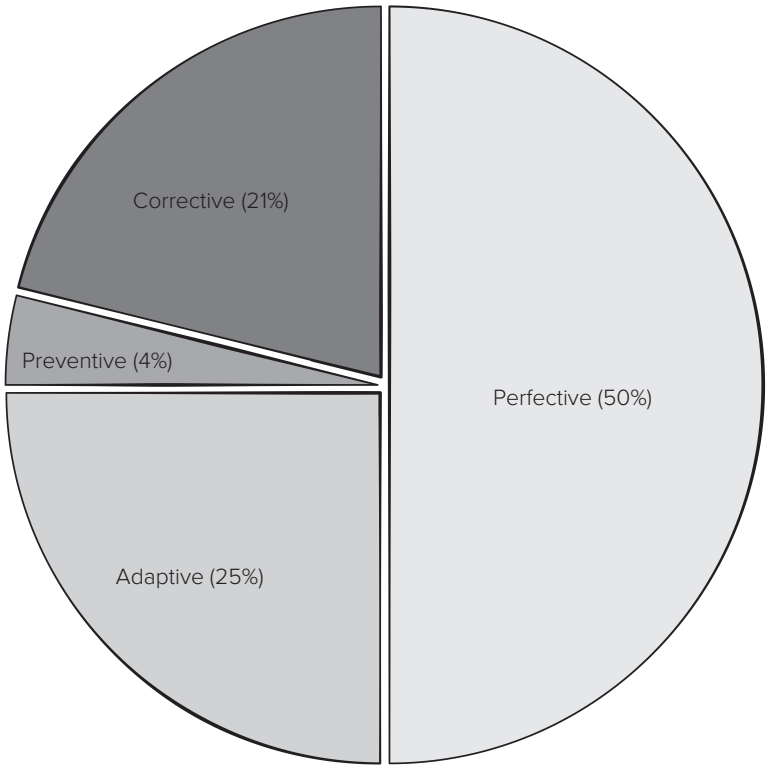
2 https://www.bugzilla.org/.
3 https://www.atlassian.com/software/jira.

discovered after the software has been delivered to a customer's end user. *Adaptive maintenance* is reactive modification of software after delivery to keep the software usable in a changing end-user environment. *Perfective maintenance* is the proactive modification of the software after delivery to provide new user features, better program code structure, or improved documentation. *Preventive maintenance* is the proactive modification of the software after delivery to detect and correct product faults before they are discovered by users in the field [SWEBOK[4]]. Proactive maintenance can be scheduled and planned for. Reactive maintenance is often described as *firefighting* because it cannot be planned for and must be attended immediately for software systems that are critical to the success of the end-users' activities. Figure 4.5 shows that only 21 percent of the developers' time is typically spent on corrective maintenance.

For an agile evolutionary process model like the one described in this chapter, developers release working partial solutions with the creation of each incremental prototype. Much of the engineering work done is preventive or perfective maintenance as new features are added to the evolving software system. It is tempting to think that maintenance is handled simply by planning to make another trip around the spiral. But software problems cannot always be anticipated, so repairs may need to be made quickly and developers may be tempted to cut corners when trying to repair the broken

FIGURE 4.5

Distribution of maintenance effort



4 SWEBOK refers to the Software Engineering Body of Knowledge, which can be accessed using the following link: https://www.computer.org/web/swebok/v3.

software. Developers may not want to spend time on risk assessment or planning. Yet, developers cannot afford to make changes to software without considering the possibility that the changes required to fix one problem will cause new problems to other portions of the program.

It is important to understand the program code before making changes to it. If developers have documented the code, it will be easier to understand if other people need to do the maintenance work. If the software is designed to be extended, maintenance may also be accomplished more easily, other than emergency defect repairs. It is essential to test the modified software carefully to ensure software changes have their intended effect and do not break the software in other places.

The task of creating software products that are easy to support and easy to maintain requires careful and thoughtful engineering. A more detailed discussion on the task of maintaining and supporting software after delivery is presented in Chapter 27.

Recommended Software Process Steps

1. Requirements engineering
   - Gather user stories from all stakeholders.
   - Have stakeholders describe acceptance criteria user stories.
2. Preliminary architectural design
   - Make use of paper prototypes and models.
   - Assess alternatives using nonfunctional requirements.
   - Document architecture design decisions.
3. Estimate required project resources
   - Use historic data to estimate time to complete each user story.
   - Organize the user stories into sprints.
   - Determine the number of sprints needed to complete the product.
   - Revise the time estimates as use stories are added or deleted.
4. Construct first prototype
   - Select subset of user stories most important to stakeholders.
   - Create paper prototype as part of the design process.
   - Design a user interface prototype with inputs and outputs.
   - Engineer the algorithms needed for first prototypes.
   - Prototype with deployment in mind.
5. Evaluate prototype
   - Create test cases while prototype is being designed.
   - Test prototype using appropriate users.
   - Capture stakeholder feedback for use in revision process.
6. Go, no-go decision
   - Determine the quality of the current prototype.
   - Revise time and cost estimates for completing development.
   - Determine the risk of failing to meet stakeholder expectations.
   - Get commitment to continue development.
7. Evolve system
   - Define new prototype scope.
   - Construct new prototype.
   - Evaluate new prototype and include regression testing.
   - Assess risks associated with continuing evolution.
8. Release prototype
   - Perform acceptance testing.
   - Document defects identified.
   - Share quality risks with management.
9. Maintain software
   - Understand code before making changes.
   - Test software after making changes.
   - Document changes.
   - Communicate known defects and risks to all stakeholders.

4.10 Summary

Every project is unique, and every development team is made up of unique individuals. Every software project needs a road map, and the process of developing software requires a predictable set of basic tasks (communication, planning, modeling, construction, and deployment). However, these tasks should not be performed in isolation and may need to be adapted to meet the needs of each new project. In this chapter, we suggested the use of a highly interactive, incremental prototyping process. We think this is better than producing rigid product plans and large documents prior to doing any programming. Requirements change. Stakeholder input and feedback should occur early and often in the development process to ensure the delivery of useful product.

We suggest the use of an evolutionary process model that emphasizes frequent stakeholder involvement in the creation and evaluation of incremental software prototypes. Limiting requirements engineering artifacts to the set of minimal useful documents and models allows the early production of prototypes and test cases. Planning to create evolutionary prototypes reduces the time lost repeating the work needed to create throwaway prototypes. Making use of paper prototypes early in the design process can also help to avoid programming products that do not satisfy customer expectations. Getting the architectural design right before beginning actual development is also important to avoiding schedule slippage and cost overruns.

Planning is important but should be done expeditiously to avoid delaying the start of development. Developers should have a general idea about how long a project will take to complete, but they need to recognize that they are not likely to know all the project requirements until the software products are delivered. Developers would be wise to avoid detailed planning that extends beyond planning the current prototype. The developers and stakeholders should adopt a process for adding features to be implemented in future prototypes and to assess the impact of these changes on the project schedule and budget.

Risk assessment and acceptance testing are an important part of the prototype assessment process. Having an agile philosophy about managing requirements and adding new features to the final product is important as well. The biggest challenges developers have with evolutionary process models is managing scope creep while delivering a product that meets customer expectations and doing all this while delivering the product on time and within budget. That's what makes software engineering so challenging and rewarding.

Problems and Points to Ponder

4.1. How does the Extreme Programming (XP) model differ from the spiral model in its treatment of incremental prototypes?

4.2. Write the acceptance criteria for the user story that describe the use of the "favorite places" or "favorites" feature found on most Web browsers that you wrote for Problem 3.7 in Chapter 3.

4.3. How would you create a preliminary architectural design for the first prototype for a mobile app that lets you create and save a shopping list on your device?

4.4. Where would you get the historic date needed to estimate the development time for the user stories in a prototype before it is written?

4.5. Create a series of sketches representing the key screens for a paper prototype for the shopping list app you created in Problem 4.3.

4.6. How can you test the viability of the paper prototype you created for Problem 4.5?

4.7. What data points are needed to make the go, no-go decision during the assessment of an evolutionary prototype?

4.8. What is the difference between reactive and proactive maintenance?