**Chapter 6. Software Requirements**

**It seems obvious** that we need to know what software is supposed to do before we build it. Nevertheless, many projects are delayed (or fail completely) because development begins before anyone on the project team really understands how the software should behave. The solution to this problem is to take the time to gather and verify the *software requirements—* documentation that completely describes the behavior that is required of the software—before the software is designed, built, and tested.

When starting a new project, programmers are often tempted to just dive in, beginning to build the software as soon as they have the general gist of what it is supposed to do. It's not hard to see why people do this: many programmers develop advanced programming skills without ever having to figure out and write down the requirements for even a simple software project. For example, many good programmers hone their craft by building software that they intend to use themselves. In these projects, the programmer does not need to take time to understand the behavior of the software—she intuitively knows what the software is supposed to do from the beginning of the project.

However, most software is built to meet the needs of someone other than the programmer. If those needs are going to be satisfied, the behavior of the software must be planned before the software is built. *Software requirements engineering* is the art and science of developing an accurate and complete definition of the behavior of software that can serve as the basis for software development. Like project management, programming, and testing, software requirements engineering encompasses a set of skills that require training and practice.

Software requirements engineering tasks are usually performed by skilled requirements analysts (who sometimes have the title "Business Systems Analyst"). If a project manager does not have a requirements analyst on his team, he may be able to rely on existing team members to fill this role. There is a good deal of overlap between the skills required for design, programming, and testing, and those required for software requirements engineering. A team member willing to spend time learning new skills (and a team willing to work with him and help him through this task) will often be able to build software requirements that are sufficient for building and testing. This chapter covers some of the most important practices that a requirements analyst uses.

**Requirements Elicitation**

*Requirements elicitation* is the process through which a requirements analyst gathers, understands, reviews, and articulates the needs of the software project's stakeholders and users. Elicitation involves fact-finding, validating one's understanding of the information gathered, and communicating open issues for resolution. The objective of this activity is to create a complete list of what the users believe are important requirements. Elicitation activities can include:

* Interviews with the users, stakeholders, and anyone else whose perspective needs to be taken into account during the design, development, and testing of the software
* Observation of the users at work
* Distribution of discussion summaries to verify the data gathered in interviews

When a software system is developed, the first step is to determine what the system will do. This sounds trivial to some people—the vision and scope document tells us what features will be developed, so why not start designing the software once everyone agrees that it's complete? This might seem like a reasonable course of action, but there's a big difference between deciding the features that will go into the software and defining the behavior of each of those features. (If a vision and scope document does not yet exist, developing one will do much more good for the project than jumping directly into the requirements. The requirements engineering activities should wait until after the vision and scope document is ready—see [Chapter 2](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch02.html).)

The objective of all elicitation activities is to create a complete list of what the users and stakeholders believe are the important needs that must be filled by the software, the behavior that the software must exhibit, and the constraints to which the software must adhere. A variety of elicitation practices can be used to gain a complete understanding of user needs. Each practice should be considered carefully, to determine which is best suited for a particular project; in addition, several techniques can be used in combination. The goal of using any of the techniques is to gain the commitment of the users so a common view of the system is established.

**Conduct Interviews**

It's been said that users don't have requirements; they have information. The requirements analyst must figure out how to get that information out of each user, stakeholder, expert, and anyone else who has information that may impact the project. The most straightforward and effective practice for doing this is by conducting interviews.

Interviews with users and stakeholders are the requirements analyst's most important elicitation tool. The goal of the interview is to identify specific needs that the person being interviewed has for the software. This generally requires understanding what the interviewee does on a day-to-day basis that will require her to interact with the software.

The first step in conducting interviews is to identify who the interviewees should be. Start with the list of users and stakeholders listed in the vision and scope document. While they are being interviewed, additional people may be mentioned. The interviewer should determine whether any of those people should be interviewed as well. If the software is being built for a specific industry or area of expertise, the requirements analyst may need to seek out subject matter experts in that area in order to ensure that the software meets the needs of a typical professional familiar with the subject.

When the software is intended to be marketed or sold by the organization, many of the market needs will originate in a sales or marketing department. In this case, it is important that sales and marketing personnel are considered subject matter experts and are interviewed for requirements. If the goal of the project is to enhance or maintain software that is already out in the field, then the interviewees should include actual external users.

The most important rule of the interview is to get the interviewee talking. Software generally doesn't get developed for its own good; a project is usually started because somebody inside an organization needs it. If the requirements analyst is talking to that person, he almost certainly will be happy to talk about it. It is likely that he has been complaining about specific problems and issues for a long time.

There are many leading questions that an interviewer can ask to help uncover important information:

* Why is the software being built? What benefits will the interviewee directly see? What benefits will other people in the organization see?
* What problems need to be addressed with new software? Why do those problems exist? How would you solve them?
* Who will use the software once it is built? Why do they need it? How frequently will they be using the software? Who will support the software?
* In what environment will the software be used? Will it be run within the organization or by its customers? Who will control the hardware?
* Are there any known constraints on the performance, design, or quality of the software?
* What inputs will be used by the software? What outputs will it create? (If examples of these exist, they should be saved for later requirements activities.)
* Are your answers "official," or is there someone else who might be able to answer these questions better? Are there "experts" in the organization who may have additional information?
* How do the users currently do their jobs? How do they expect their jobs to change after the software goes into production? Are there typical problems they currently encounter that they would like to fix?
* Are there any "workarounds" that the users perform, in order to make up for the shortcomings of existing software? Can these workarounds be incorporated into planned features or turned into additional ones? (If the workarounds are not part of the original scope of the project, does that scope need to be expanded to include them?)
* "Is there anything that I missed?"

In the early stage of interviews, any kind of open-ended questions are good for getting the interviewee talking. It is important that the requirements analyst does not interrupt the interviewee. If he has something to say, it could be important. One key to interviewing is to make sure that the predispositions of the interviewer do not interfere with the free exchange of information. Questions should center on the interviewee's problems. The requirements analyst should try to gain an understanding of the real problem without introducing bias into the user's information.

When there are multiple people with similar job functions or expertise, it is often helpful to interview them in groups. When there are too many individual interviews, they will often turn into an endless chain of conflicting statements, making it difficult to reach consensus.

After an initial set of interviews, a requirements analyst should follow up with meetings to verify his or her notes and to gather any new information that the interviewees may have come up with. These meetings may consist of additional interviews, ad-hoc meetings with one or more people, brainstorming sessions, or other kinds of interactions.

NOTE

More information on interviews can be found in *Managing Software Requirements: A Use Case Approach* by Dean Leffingwell and Don Widrig (Addison Wesley, 2003).

**Observe Users at Work**

Observing a user's workflow in the context of his or her environment allows a requirements analyst to see problems that the user encounters with the current system and to identify ways to enhance and streamline the behavior of the software. Watching users at work provides a more accurate understanding of their activities than simply asking them to communicate the steps involved in performing their tasks. Often there are many details that someone familiar with a task might not think to mention, but that are very important for the requirements analyst to fully understand the task.

Most software is built to help users with a job that they already do. Many software projects have a goal of automating or augmenting existing manual processes, providing new capabilities to existing people in an organization, or replacing and extending legacy software programs. In all of these cases, there are already people in the organization who are doing work that is relevant to the behavior of the software.

Once the requirements analyst has identified people who will use the software and whose day-to-day work is relevant to the behavior of the software, it is useful to observe those people at their jobs.

There are a few guidelines that may be useful:

* Many software projects are started because people who are doing work face tedious or difficult tasks that could be automated. Those people are often happy to open up and talk about the problems that need to be solved.
* Some people may feel self-conscious being observed. It is important that they know that the goal of the requirements analyst is to understand their needs in order to build software to help them. It is also important that they understand that the software is not being built to replace them, but to help make their jobs easier.
* Many organizations have training programs for new employees. If a training program exists for the job being observed, the requirements analyst should attend it. This will often yield more insight into the work, especially if there are training materials that can be used as part of the requirements gathering process.
* If possible, the requirements analyst should try to participate in the work being observed. This is often an effective way to understand the perspective of the users of the software.

**Use a Discussion Summary**

All of the interaction with the users, stakeholders, and other people who have relevant perspectives will yield a great deal of information. The requirements analyst should capture this information as accurately as possible. Some people find that taking notes on paper (rather than using a computer) allows for better capture of information. However, note-taking software, which captures the interviews in an audio file and links notes to their correct time offset in that file, can also be a useful tool.

Once all of the information has been gathered from the elicitation activities, the requirements analyst should use a *discussion summary* to validate the information. The discussion summary allows all of the notes to be summarized into a single document; that document should be distributed to the main users and stakeholders for a deskcheck (see [Chapter 5](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch05.html)). This will help catch any potential defects before they make it into the requirements documentation. [Table 6-1](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch06.html#appliedprojectmgmt-CHP-6-TABLE-1) contains a template for the discussion summary.

*Table 6-1. Discussion summary template*

|  |
| --- |
| 1. Project background    1. Purpose of project    2. Scope of project    3. Other background information 2. Perspectives    1. Who will use the system?    2. Who can provide input about the system? 3. Project objectives    1. Known business rules    2. System information and/or diagrams    3. Assumptions and dependencies    4. Design and implementation constraints 4. Risks 5. Known future enhancements 6. References 7. Open, unresolved, or TBD issues |

Most, if not all, of the notes gathered by a requirements analyst during the elicitation activities should fit into the discussion summary template. Conversely, the discussion summary template can serve as a guide for a novice requirements analyst in leading interviews and meetings.

The requirements analyst should go through all of her notes when writing the discussion summary. Each note that will be relevant to the behavior of the software should be summarized and added to one of the sections of the discussion summary. The discussion summary will be ready to distribute when it contains all of the information gathered during the elicitation sessions.

This "Project Background" section contains a summary of all the notes that pertain to the background of the project. It contains these subsections:

*Purpose of the project*

Every stakeholder and user has a reason that the software should be developed. This section should contain a summary of each of those reasons. The goal is to give the reader an understanding of why these people need the system to be developed.

*Scope of the project*

The vision and scope document described the scope of the software to be developed, by listing each feature that would be included. This section should go into greater detail, elaborating on each feature by listing specific behaviors and tasks the software will perform.

*Other background information*

This section should contain any additional information that may help a reader understand why the system is needed. (Additionally, most of the notes that don't fit anywhere else in the discussion summary can go into this section.)

The "Perspectives" section is used to identify the people who will help define the behavior of the software. Each person's perspective should be taken into account. Some of these perspectives may conflict with each other; that's okay, as long as they are described accurately. These conflicts will be worked out later, when the behavior is described in the use cases. This section must provide/contain answers to the following key questions:

*Who will use the system?*

The people who will be using the software should be divided into categories. Each category of users should have a unique name ("Salesperson") and description ("A member of the North America sales team who will be selling the software in a specific territory."). The names that are given to the categories of users should make sense to people in the organization—most organizations have their own names for different roles or positions, and the requirements analyst should use that terminology wherever possible. The analyst will have many notes that pertain to each of these categories. They should be divided up by category and summarized in this section.

*Who can provide input about the system?*

The organization contains many people who can provide some input about the system to be developed. This section should list everyone who was consulted about the system behavior, and summarize any notes that describe the needs of each one.

The "Project objectives" section summarizes the information that was gathered in the elicitation phase, such as the functionality that the software must implement, the work currently being done or planned in the organization that will be affected or augmented by the software, and any constraints that must be taken into account. This section contains:

*Known business rules*

This section should contain details of any procedures that are currently being performed or that are needed in the organization and that will affect the software. The section should indicate who is involved or will be affected.

*System information and/or diagrams*

This section should contain a summary of any notes that describe functionality that must be implemented, existing or planned organizational workflow, specific user interactions, information about the environment in which the software will be used, calculations that must be performed, and any other functionality that must be implemented. This will probably be the longest section in the discussion summary.

*Assumptions and dependencies*

During elicitation meetings, many assumptions and dependencies will be brought up. They should be summarized in this section. (See [Chapter 3](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch03.html) for an explanation of how assumptions affect software development.) Many of these assumptions may already be in the vision and scope document, or in the results of a Wideband Delphi estimation session; it is often sufficient to reference that document, rather than reproduce them in the discussion summary.

*Design and implementation constraints*

Many times there are constraints that must be placed on the software: known input or output data formats; tools, code libraries, visual controls, programming languages, or APIs that must be used; visual or GUI design standards that must be adhered to; and performance or quality requirements and other known nonfunctional requirements. These should be listed in this section in detail.

The "Risks" section summarizes any risks identified during the elicitation process that are not already included in the vision and scope document or the project plan.

The "Known future enhancements" section lists expected future enhancements. Often, during elicitation, there are feature requests from users or stakeholders that will not be included in the software. Those requests should be described, in order to make sure everyone knows that they are not going to be implemented.

The "References" section should include any internal or external documents needed to understand the software project—for example, any existing system documentation, screen shots, or original system requirements.

The "Open, unresolved, or TBD [to be determined] issues" section is the last part of the discussion summary. There are usually issues that remain unaddressed at the time the discussion summary is distributed for review. Open issues may be under active discussion with users or stakeholders. Some problems may await resolution. And the requirements analyst may not yet have raised some issues. All of these issues should be summarized here.

Once the discussion summary is complete, it should be distributed for a deskcheck (see [Chapter 5](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch05.html)). Minimally, it should be reviewed by the lead users and stakeholders (who are usually listed in the vision and scope document). But ideally, everyone who contributed to the discussion summary should have a chance to review it and give feedback. The wider the audience, the more likely it is that defects will be caught early. When the reviewers return their comments, the discussion summary does not need to be updated. However, those comments should be archived along with the discussion summary, and should be taken into account in later requirements activities.

**Use Cases**

Once the initial round of requirements elicitation is done and a discussion summary has been distributed and reviewed, the requirements analyst is ready to begin creating use cases . A *use case* is a description of a specific interaction that a user may have with the software. Use cases are deceptively simple tools for describing the behavior of the software.

A use case contains a textual description of all of the ways that the intended users could work with the software through its interface. Use cases do not describe any internal workings of the software, nor do they explain how that software will be implemented. They simply show the steps that the user follows to use the software to do his work. All of the ways that the users interact with the software can be described in this manner.

A typical use case includes these sections, usually laid out in a table. [Table 6-2](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch06s02.html#appliedprojectmgmt-CHP-6-TABLE-2) shows a template for describing a use case.

*Table 6-2. Use case template*

| **Name** | **Use case number and name** |
| --- | --- |
| Summary | Brief description of the use case |
| Rationale | Description of the reason that the use case is needed |
| Users | A list of all of the categories of users that interact with this use case |
| Preconditions | The state of the software before the use case begins |
| Basic Course of Events | A numbered list of interactions between the user and one or more users |
| Alternative Paths | Conditions under which the basic course of events could change |
| Postconditions | The state of the software after the basic course of events is complete |

**Name, Summary, Rationale, and Users**

Each use case must begin with information that allows the reader to uniquely identify it. Every use case has a descriptive name and a unique identifying number. The number is used as a way to refer to a specific use case in the SRS (see below). In addition to identifying information, each use case has a summary, or a brief description of what the use case does.

The "Rationale" section of the use case contains one or more paragraphs that describe why the use case is needed. This serves as an important quality check to ensure the correctness of the use case. While it is important that the team agrees on the behavior of the software, it is equally important that they also agree on why the software is being created.

Each use case represents a series of interactions between the software and one or more users. The users are divided into categories based on the way they interact with the software; the "Users" section lists the kinds of users that interact with this use case. If all categories of users interact with this particular use case, it should list "any user" in this section.

**Preconditions and Postconditions**

Any software that is being executed can be thought of as being in a state of operation. When the software is in a certain state, it means that a specific set of operations are available to the user that are not available when the software is in other states. For example, a word processor could have an existing document loaded, a new document displayed, or it could be displaying no document at all. It could be showing a configuration window or a dialog box. These are all different, distinct states that the word processing software can be in. There are certain actions that are only available to the user when the software is in a particular state. For example, the user can enter text into the document if the word processor has an existing document loaded, but not if it is displaying a dialog box.

The *precondition* is the state of the software at the beginning of the use case. This represents the entry criteria for the use case: if the software is not in that state, the use case does not apply. The *postcondition* is the state that the software is left in after the use case is completed. In the use case, each of these states can be described in words ("the word processor is loaded but no document is being edited"), although it is also possible to create a name for each state and refer to it by name.

**Basic Course of Events**

The *basic course of events* is the core of the use case. [Table 6-3](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch06s02.html#appliedprojectmgmt-CHP-6-TABLE-3) shows a very simple basic course of events for a word processor's search-and-replace feature.

*Table 6-3. Basic course of events for search-and-replace*

|  |  |
| --- | --- |
| Precondition | A document is loaded and being edited. |
| Basic course of events | 1. The user indicates that the software is to perform a search-and-replace in the document. 2. The software responds by requesting the search term and the replacement text. 3. The user inputs the search term and replacement text and indicates that all occurrences are to be replaced. 4. The software replaces all occurrences of the search term with the replacement text. |
| Postcondition | All occurrences of the search term have been replaced with the replacement text. |

The basic course of events consists of a series of steps. The first step will generally be the action that the user takes in order to initiate the use case. The remaining steps are a series of interactions between the user and the software. Each interaction consists of one or more actions that the user takes, followed by a response by the software. In this case, the software is responding to the user entering the search term and replacement text and indicating that the replacement should occur.

This basic course of events does not contain words like "click," "button," "text box or "window." User interface design elements should be left out of use cases to allow the designer as many options as possible. For example, this particular use case could be implemented as a pop-up dialog box that contains text boxes for the search term and replacement text, and a button that says "Replace All." (This is how many word processors, including Microsoft Word, do it). But that's not the only way to satisfy this use case. In the Emacs text editor, for example, the user hits Meta-X and enters "replace-string" on the bottom line of the window, followed by the search term and the replacement term. Either implementation would satisfy this use case.

Sometimes there is functionality that is replicated in many use cases. For example, say the use cases for a program to play music files includes UC-15 (see [Table 6-4](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch06s02.html#appliedprojectmgmt-CHP-6-TABLE-4)), a use case that allows a user to explore and edit information in one of his audio files.

*Table 6-4. Basic course of events for "UC-15: Edit Audio File Information"*

|  |  |
| --- | --- |
| Precondition | An audio file is highlighted. |
| Basic course of events | 1. The user indicates that the software is to display information about the audio file. 2. The software responds by displaying the information fields (track name, artist, length, genre, and year) associated with the audio file. 3. The user indicates that one of the information fields is to be replaced and specifies a replacement value. 4. The software updates the audio file with the new value. |
| Postcondition | Same as precondition state, except the audio file has an updated information field. |

If the requirements analyst intends that the user be given the option to select an audio file in many different use cases and edit that file at any time, each of those use cases may have a use case step or an alternative path that reads: "The user indicates that the file is to be edited. Use case UC-15 is executed." In this way, multiple use cases can be extended to include this functionality. The requirements analyst needs to describe it only once, which makes the use cases clearer. This technique has the benefit of giving the designers and programmers hints about where they can reuse code when the software is being developed.

**Alternative Paths**

Often, a use case has one basic course of events, as well as several alternative courses that are very similar and that share many of the same steps. In these cases, they are documented as different *alternative paths* (rather than in separate use cases), to show that they are closely related. An alternative path provides behavior that is similar to the basic course of events but that differs in one or more key behaviors.

For example, an alternative path for the search-and-replace use case would be to only replace the first occurrence of the search string. [Table 6-5](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch06s02.html#appliedprojectmgmt-CHP-6-TABLE-5) shows the alternative path for this behavior, as well as an alternative path for searching without replacement and another one for aborting the operation:

*Table 6-5. Alternative paths for search-and-replace*

|  |  |
| --- | --- |
| Alternative paths | 1. In Step 3, the user indicates that only the first occurrence is to be replaced. In this case, the software finds the first occurrence of the search term in the document being edited and replaces it with the replacement text. The postcondition state is identical, except only the first occurrence is replaced and the replacement text is highlighted. 2. In Step 3, the user indicates that the software is only to search and not replace, and does not specify replacement text. In this case, the software highlights the first occurrence of the search term and the use case ends. 3. The user may decide to abort the search-and-replace operation at any time during Steps 1, 2, or 3. In this case, the software returns to the precondition state. |

As the requirements analyst defines additional alternative paths, it may become clear that one of them is more likely to be used than the basic course of events. In this case, it may be useful to swap them—make the alternative path into the basic course of events, and add a new alternative path to describe the behavior previously called the basic course of events.

[Table 6-6](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch06s02.html#appliedprojectmgmt-CHP-6-TABLE-6) shows a final use case for a search-and-replace function, which is numbered UC-8 in this example.

*Table 6-6. Use case for a simple search-and-replace function*

| **Name** | **UC-8: Search** |
| --- | --- |
| Summary | All occurrences of a search term are replaced with replacement text. |
| Rationale | While editing a document, many users find that there is text somewhere in the file being edited that needs to be replaced, but searching for it manually by looking through the entire document is time-consuming and ineffective. The search-and-replace function allows the user to find it automatically and replace it with specified text. Sometimes this term is repeated in many places and needs to be replaced. At other times, only the first occurrence should be replaced. The user may also wish to simply find the location of that text without replacing it. |
| Users | All users |
| Preconditions | A document is loaded and being edited. |
| Basic course of events | 1. The user indicates that the software is to perform a search-and-replace in the document. 2. The software responds by requesting the search term and the replacement text. 3. The user inputs the search term and replacement text and indicates that all occurrences are to be replaced. 4. The software replaces all occurrences of the search term with the replacement text. |
| Alternative paths | 1. In Step 3, the user indicates that only the first occurrence is to be replaced. In this case, the software finds the first occurrence of the search term in the document being edited and replaces it with the replacement text. The postcondition state is identical, except only the first occurrence is replaced, and the replacement text is highlighted. 2. In Step 3, the user indicates that the software is only to search and not replace, and does not specify replacement text. In this case, the software highlights the first occurrence of the search term and the use case ends. 3. The user may decide to abort the search-and-replace operation at any time during Steps 1, 2, or 3. In this case, the software returns to the precondition state. |
| Postconditions | All occurrences of the search term have been replaced with the replacement text. |

**Develop Use Cases Iteratively**

As the use cases are developed, additional information about how the software should behave will become clear. Exploring and writing down the behavior of the software will lead a requirements analyst to understand various aspects of the users' needs in a new light, and additional use cases and functional requirements will start to become clear as well. As this happens, they should be written down with a name, number, and summary—once they are in this form, the analyst can apply the four-step process to complete them.

The first step in developing use cases is identifying the basic ones that will be developed. The list of features in the vision and scope document is a good starting point, as there will usually be at least one use case per feature (usually more than one). This will probably not be the final set of use cases—additional ones will probably be discovered during the development of the use cases.

Many requirements analysts have found that a four-step approach is effective in developing use cases. [Table 6-7](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch06s02.html#appliedprojectmgmt-CHP-6-TABLE-7)contains a script that describes this approach.

*Table 6-7. Use case development script*

| **Name** | **Use case development script** |
| --- | --- |
| Purpose | A four-step approach to use case development |
| Summary | This approach to developing use cases allows the information to be gathered and documented naturally, in a way that lends itself to an iterative approach of alternating iteration, documentation, and verification of use cases. |
| Work products | *Output*  Use Cases |
| Entry criteria | A requirements analyst has received feedback from elicitation and is ready to develop use cases. |
| Basic course of events | 1. Identify the basic set of use cases. Assign a name and number to each use case. 2. Add a rationale and summary to each use case. Identify which users will interact with each use case, and add them as well. Create a master list of user categories that identifies all of the information known about each kind of user: titles, roles, physical locations, approximate number of users in the category, organizational policies they must adhere to, and anything else that makes someone part of their category. Where possible, add precondition and postcondition states to the use cases. 3. Define the basic course of events and the alternative paths for each use case. Finish adding the precondition and postcondition states. If additional users and use cases are discovered, add them as well (starting with just a name and number, and then adding the other information as in Step 2). 4. Verify each use case, ensuring that all paths make sense and are correct. Go through each step with user representatives to make sure that they accurately represent the way they expect the users to interact with the software. Look for any steps that are awkward for the user that could be made more efficient. Finish all use cases that were added in Step 3. |
| Exit criteria | The use cases are complete and no additional information has been uncovered, which may lead to additional use cases being developed. If additional use cases have been discovered, return to Step 1 to fill them in. |

A requirements analyst defining a set of use cases for this software would start by creating one use case for each feature. Initially, each of these would have a name and a number. The numbering system does not matter, as long as it is unique. (A number such as "UC-1" is sufficient.) The requirements analyst should create a new use case document with a blank template for each of these use cases, filling in the name and number for each of them and proceeding through each of the four steps to create a complete set of use cases.

**Software Requirements Specification**

A *software requirements specification* (SRS) is a complete description of the behavior of the software to be developed. It includes a set of use cases that describe all of the interactions that the users will have with the software. In addition to use cases, the SRS contains *functional requirements*, which define the internal workings of the software: that is, the calculations, technical details, data manipulation and processing, and other specific functionality that shows how the use cases are to be satisfied. It also contains *nonfunctional requirements*, which impose constraints on the design or implementation (such as performance requirements, quality standards, or design constraints).

The SRS is the most important work product that is produced by the requirements engineering project activities. All later work products—software design and architecture, code, test plans—are based on the SRS.

**SRS Template**

The SRS contains the use cases, functional requirements, and nonfunctional requirements. It should also contain overall information about the project, in order to orient the team. [Table 6-8](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch06s03.html#appliedprojectmgmt-CHP-6-TABLE-8) shows the SRS template .

*Table 6-8. Software requirements specification outline*

|  |
| --- |
| 1. Introduction    1. Purpose    2. Scope    3. System overview    4. References 2. Definitions 3. Use cases 4. Functional requirements 5. Nonfunctional requirements |

The introduction serves to orient the reader. It describes both the system and the SRS itself:

*Purpose*

This section describes the purpose of the document. Typically, this will contain a brief two- or three-sentence description, including the name of the project. For example: "The purpose of this document is to serve as a guide to designers, developers, and testers who are responsible for the engineering of the *(name of project)* project. It should give the engineers all of the information necessary to design, develop, and test the software." This is to ensure that the person reading the document understands what he or she is looking at.

*Scope*

This section contains a brief description of the scope of the document. If the SRS is a complete description of the software, then it will state something similar to: "This document contains a complete description of the functionality of the *(name of project)* project. It consists of use cases, functional requirements , and nonfunctional requirements, which, taken together, form a complete description of the software." For complex software, the requirements for the project might be divided into several SRS documents. In this case, the scope should indicate which portion of the project is covered in this document.

*System overview*

This section contains a description of the system. This is essentially a brief summary of the vision and scope of the project.

*References*

Any references to other documents (including the vision and scope document) should be included here. These may include other documents in the organization, work products, articles, and anything else that is relevant to understanding the SRS. If there is an organizational intranet, this section often includes URLs of referenced documents.

*Definitions*

The "Definitions" section contains any definitions needed to understand the SRS. Often, it will contain a glossary, defining terms that the reader may not be familiar with (or that may have a specific meaning here that differs from everyday use). This section may also contain definitions of any data files that are used as input, a list of any databases that may be needed, and any other organizational or workflow-related information that is needed to understand the SRS.

The remaining sections contain a complete description of the behavior of the software. The "Use Cases*"* section contains each of the use cases. Each use case is represented by a table, which is in the format shown in [Table 6-6](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch06s02.html#appliedprojectmgmt-CHP-6-TABLE-6). The "Functional requirements" section contains the functional requirements, and the "Nonfunctional requirements" section contains the nonfunctional requirements. Each functional and nonfunctional requirement is added to the SRS as a table (using the format shown in [Table 6-9](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch06s03.html#appliedprojectmgmt-CHP-6-TABLE-9)).

The SRS should also contain a complete table of contents that includes the name and number of each use case, functional requirement, and nonfunctional requirement.

**FUNCTIONAL REQUIREMENTS**

Once an initial set of use cases has been created and filled in, the requirements analyst begins documenting the *functional requirements*. [Table 6-9](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch06s03.html#appliedprojectmgmt-CHP-6-TABLE-9) shows the template for a functional requirement.

*Table 6-9. Functional and nonfunctional requirement template*

|  |  |
| --- | --- |
| Name | Name and number of the functional requirement |
| Summary | Brief description of the requirement |
| Rationale | Description of the reason that the requirement is needed |
| Requirements | The behavior that is required of the software |
| References | Use cases and other functional and nonfunctional requirements that are relevant to understanding this one |

The name, summary, and rationale of each functional requirement are used in the same way as those of the use cases. The behavior that is to be implemented should be described in plain English in the "Requirements" section. Most requirements are only relevant to a small number of use cases—these should be listed by name and number in the "References" section. (Some requirements are not associated with use cases.)

The core of the requirement is the description of the required behavior. It is very important to make this clear and readable. This behavior may come from organizational or business rules, or it may be discovered through elicitation sessions with users, stakeholders, and other experts within the organization. Many requirements will be uncovered during the use case development. When this happens, the requirements analyst should create a placeholder requirement with a name and summary, and research the details later, to be filled in when they are better known.

[Table 6-10](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch06s03.html#appliedprojectmgmt-CHP-6-TABLE-10) shows an example of a requirement that might be discovered during the development of the search-and-replace use case in [Table 6-6](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch06s02.html#appliedprojectmgmt-CHP-6-TABLE-6).

*Table 6-10. Functional requirement example*

| **Name** | **FR-4: Case sensitivity in search-and-replace** |
| --- | --- |
| Summary | The search-and-replace feature must have case sensitivity in both the search and the replacement. |
| Rationale | A user will often search for a word that is part of a sentence, title, heading, or other kind of text that is not all lowercase. The search-and-replace function needs to be aware of that, and give the user the option to ignore it. |
| Requirements | When a user invokes the search-and-replace function, the software must give the option to do a case-sensitive search.  By default, the search will match any text that has the same letters as the search term, even if the case is different. If the user indicates that the search is to be done with case-sensitivity turned on, then the software will only match text in the document where the case is identical to that of the search term.  During a search and replace, when the software replaces original text in the document with the replacement text specified by the user, the software retains the case of the original text as follows:   * If the original text was all uppercase, then the replacement text must be inserted in all uppercase. * If the original text was all lowercase, then the replacement text must be inserted in all lowercase. * If the original text had the first character uppercase and the rest of the characters lowercase, then the replacement text must reflect this case as well. * If the original text was sentence case (where the first letter of each word is uppercase), then the replacement text must be inserted in sentence case. * In all other cases, the replacement text should be inserted using the case specified by the user. |
| References | UC-8: Search |

The behavior in the requirement can contain lists, bullets, equations, pictures, references to external documents, and any other material that will help the reader understand what needs to be implemented. The goal of the requirement is to communicate this behavior in as clear and unambiguous a manner as possible.

One pitfall is to get hung up on ambiguity. For example, the example above refers to "the same letters as the search term." This is potentially ambiguous—an overly pedantic reading could interpret it to mean "the same letters as the search term but in any order," and that would technically be correct because the requirement does not specify that the letters should not be out of order. However, it is unlikely that the text will be read that way. In this case, all of the developers will almost certainly understand what the requirements analyst meant when the text was written, and writing the extra explanation into the requirement could make it more confusing.

The requirements analyst has a great deal of discretion about how to break requirements up. For example, the example above could potentially be broken into two separate requirements, with one requirement allowing for a case-sensitive search and another specifying that the replacement text needs to retain the case of the original text when the search is not case-sensitive. A good general rule is that requirements should only be split up when that split offers more clarity. In this case, a split probably would not make the requirement any clearer, so keeping these together makes sense.

Many requirements define behavior that the user can turn on or off (like the user's ability to specify a case-sensitive search in the example). In this case, the requirement should always specify the default behavior (in the example case, the default search is case-insensitive). Requirements often specify behavior that only occurs under certain conditions (like the conditions in which the case is retained). It is also important to specify a fall-through case to show what happens if none of the specified conditions are met ("In all other cases...").

As with the use case, the requirements should not contain any design elements. The requirements analyst should avoid words like window, button, click, checkbox, radio button, form, etc. For example, many search-and-replace features use a checkbox to implement context-sensitive searching. But there is no need to require that the designer use a checkbox—it's possible that an innovative designer might come up with a better way to let the user turn on case-sensitive searching (like with two separate buttons, a radio button, a separate tab, or some other way that isn't immediately obvious). Leaving the design unconstrained allows the designers and programmers more freedom to implement the requirements creatively. The more that the design is constrained in the SRS, the harder it makes their jobs.

**NONFUNCTIONAL REQUIREMENTS**

Users have implicit expectations about how well the software will work. These characteristics include how easy the software is to use, how quickly it executes, how reliable it is, and how well it behaves when unexpected conditions arise. The *nonfunctional requirements* define these aspects about the system. (The nonfunctional requirements are sometimes referred to as "nonbehavioral requirements" or "software quality attributes.")

The nonfunctional requirements should be defined as precisely as possible. Often, this is done by quantifying them. Where possible, the nonfunctional requirements should provide specific measurements that the software must meet. The maximum number of seconds it must take to perform a task, the maximum size of a database on disk, the number of hours per day a system must be available, and the number of concurrent users supported are examples of requirements that the software must implement but do not change its behavior.

There are many kinds of nonfunctional requirements, including:

*Availability*

A system's availability, or "uptime," is the amount of time that it is operational and available for use. This is specified because some systems are designed with expected downtime for activities like database upgrades and backups.

*Efficiency*

Specifies how well the software utilizes scarce resources: CPU cycles, disk space, memory, bandwidth, etc.

*Flexibility*

If the organization intends to increase or extend the functionality of the software after it is deployed, that should be planned from the beginning; it influences choices made during the design, development, testing, and deployment of the system.

*Portability*

Portability specifies the ease with which the software can be installed on all necessary platforms, and the platforms on which it is expected to run.

*Integrity*

Integrity requirements define the security attributes of the system, restricting access to features or data to certain users and protecting the privacy of data entered into the software.

*Performance*

The performance constraints specify the timing characteristics of the software. Certain tasks or features are more time-sensitive than others; the nonfunctional requirements should identify those software functions that have constraints on their performance.

*Reliability*

Reliability specifies the capability of the software to maintain its performance over time. Unreliable software fails frequently, and certain tasks are more sensitive to failure (for example, because they cannot be restarted, or because they must be run at a certain time).

*Reusability*

Many systems are developed with the ability to leverage common components across multiple products. Reusability indicates the extent to which software components should be designed in such a way that they can be used in applications other than the ones for which they were initially developed.

*Robustness*

A robust system is able to handle error conditions gracefully, without failure. This includes a tolerance of invalid data, software defects, and unexpected operating conditions.

*Scalability*

Software that is scalable has the ability to handle a wide variety of system configuration sizes. The nonfunctional requirements should specify the ways in which the system may be expected to scale up (by increasing hardware capacity, adding machines, etc.).

*Usability*

Ease-of-use requirements address the factors that constitute the capacity of the software to be understood, learned, and used by its intended users.

The nonfunctional requirements can use the same template as the functional requirements. [Table 6-11](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch06s03.html#appliedprojectmgmt-CHP-6-TABLE-11) shows an example of a nonfunctional requirement:

*Table 6-11. Nonfunctional requirement example*

| **Name** | **NF-7: Performance constraints for search-and-replace** |
| --- | --- |
| Summary | The search-and-replace feature must perform a search quickly. |
| Rationale | If a search is not fast enough, users will avoid using the software. |
| Requirements | A case-insensitive search-and-replace performed on a 3MB document with twenty 30-character search terms to be replaced with a different 30-character search term must take under 500ms on a 700MHz Pentium III running Microsoft Windows 2000 at 50% CPU load. |
| References | UC-8: Search. |

**Develop the SRS Iteratively**

Like use cases, the SRS should be developed in a highly iterative manner. [Table 6-12](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch06s03.html#appliedprojectmgmt-CHP-6-TABLE-12) shows the SRS development script, an iterative process that guides a requirements analyst through the development of a software requirements specification.

*Table 6-12. SRS development script*

| **Name** | **Software requirements specification development script** |
| --- | --- |
| Purpose | To elicit software requirements, document them in a software requirements specification, and verify that it is correct. |
| Summary | The development of the software requirements specification should be the most iterative part of the entire project. This is the point where the behavior of the software to be developed is at its most malleable—it has only been described in words, and has not yet been realized in design, architecture, code, test plans, or any other work product. The goal of this script is to ensure that as many defects are found as possible, because each defect missed at this stage will be much more costly to detect and fix later on in the project. |
| Work products | *Output*  Software Requirements Specification (SRS) |
| Entry criteria | A requirements analyst has a vision and scope document for a project and has identified a set of users, stakeholders, and other people who will participate in the elicitation process. |
| Basic course of events | 1. *Elicitation:* The script begins with the elicitation process. The requirements analyst works with users, stakeholders, and other people to elicit their needs and any known requirements. If there are outstanding issues or SRS defects, the analyst resolves them during the elicitation activities. 2. *Documentation:* The requirements analyst creates or updates the draft of the SRS to reflect the data elicited in Step 1. 3. *Verification:* A team of reviewers performs a review of the SRS draft. In the first iteration, a small number of reviewers perform a deskcheck of the draft. Later reviews will include more people, and may be inspections instead of deskchecks. Walkthroughs should be conducted for non-technical people who still need to understand the contents of the SRS. The last iteration must be an inspection. |
| Exit criteria | The script ends after the draft was inspected in Step 3 and no defects were found. If defects were found or there was only a deskcheck performed in Step 3, then the script returns to step 1 for the next iteration. |

The goal of the SRS development script is to remove as many defects as possible from the SRS. Many people have trouble figuring out what constitutes a defect. In this case, a defect is any planned software behavior a project team member, user, stakeholder, or decision-maker does not agree with. This means that defects could be caused by any number of problems:

* Somebody does not believe that the planned behavior will satisfy the users' or stakeholders' needs.
* Somebody believes that those needs may be better satisfied with different behavior.
* An inspector does not understand what's written or feels that it is ambiguous or confusing.
* A project team member does not believe that the behavior can be implemented as written.
* Two or more requirements contradict each other—an implementation that satisfies one cannot satisfy the others.

If there is a requirement in the SRS that has one of these problems, it must be identified and fixed so that everyone agrees on everything in the document. There will almost certainly be defects that slip through—no inspection team is perfect—and each of these will cost much more time to fix after it has been designed, coded, and tested than it would have if it had been caught using the SRS development script. The goal is to find as many defects as possible, in order to reduce the amount of time that the team must spend later on in the project undoing the few that slipped past.

One of the most common mistakes in software engineering is to shortchange the SRS. A good project manager knows that the fastest way to finish a project is to take the time up front to get the requirements right. For many managers, however, cutting the requirements short sometimes seems tempting. They are impatient to have the programmers begin working on something concrete that they can use, and do not necessarily see the value of planning the behavior of the software before it is built. But skipping the software requirements activities will always come back to damage the project later. Any defects in the SRS will get magnified in later work products. If a defect is left in the SRS and not caught until testing, that defect will be designed into the software. If the defect is still not caught in the design phase, then code will be built to implement that design, and a test plan will be created and executed that verifies that behavior is accurately reflected in the final build. When that defect is eventually discovered by a user, fixing it will require an enormous amount of effort from the entire project team. This is why it is worth spending time doing extra iterations of reviews at the outset of the project—it's much more efficient to build the software right the first time than it is to go back and fix it later.

The best way to prevent defects is through iteration. This is why the SRS development script calls for repeated reviews of the SRS. It may seem to the project team that they are sitting through redundant deskchecks, walkthroughs, and inspections of a single SRS. It is sometimes tempting to cut this process short and declare that the SRS is "good enough." The project manager needs to resist this urge. When the SRS goes through what some people might perceive as "yet another iteration," it's because somebody found a defect in it that needs to be fixed. If this defect is not fixed now, the effort required to fix it later will dwarf the effort required to do another review of the SRS.

There is a risk that when nontechnical people read the SRS draft, they will fail to understand what is written in it and simply agree to whatever is written down. This often happens in an organization in which software requirements have never been used before. To address this problem, the project manager should work to help everyone understand why it is so important that they take the time to read and understand each SRS draft. This often requires sitting down with each person to make sure he really read and understood the document. Even if that seems time-consuming and patronizing, it's better than losing an enormous portion of the project's effort because the team is busy fixing defects that could have been caught in the first place.

If users and stakeholders are used to playing with intermediate builds and vetoing software features that they don't like, they may object to sitting through a walkthrough meeting rather than waiting until the programmers produce something. However, that's a very costly way to build software. Words on a page are much easier to change than functions and objects in source code. A defect that only takes a few minutes to fix in a draft of an SRS can require days, weeks, or months of the most irritating sort of programming to repair after it's been coded. This is why it's in the project team's best interest to catch the defects as early as possible.

Another problem is that some people may not read repeated drafts; after seeing the first few drafts, they may start to ignore later revisions. One way to reduce the risk of this happening is to keep the initial iterations' review teams very small, and only expand them when the draft is close to completion. People who are not used to reading technical documentation are much more likely to read a single draft than they are to read a half-dozen of them. But in the last iterations, it still may be difficult to get nontechnical people to work their way through a technical document. This is where a walkthrough can be a useful tool to ensure that everyone understands the document. (See [Chapter 5](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch05.html) for more information on walkthroughs.)

**Requirements Differ from Design**

Many people have trouble understanding the difference between scope (which demonstrates the needs of the organization), the requirements (which describe the behavior of the software to be implemented), and design (which shows how the behavior will be implemented). It is hard for them to describe the behavior of a piece of software without talking about windows, forms, buttons, checkboxes, clicking, dragging, etc. The difference between scope, requirements, and design is a very important distinction that the entire project team should be familiar with.

[Table 6-13](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch06s03.html#appliedprojectmgmt-CHP-6-TABLE-13) shows several examples that illustrate the difference between the *user needs* that might appear in a vision and scope document, the *behavior* that might appear in an SRS, and the *design* that might appear in a design specification or code comments. As a general rule, unless the requirements analyst specifically intends to constrain the designers and programmers, design elements should be left out of the SRS.

*Table 6-13. Examples to illustrate the difference between needs, behavior, and design*

| **Needs** | **Behavior** | **Design** |
| --- | --- | --- |
| The users need to be able to easily search and replace text. | A user indicates that search-and-replace is needed. The software responds by prompting for a search term and replacement text. The user enters the text and indicates that the software is to do a case-sensitive search to replace all occurrences. | A user selects "Search" from the "Edit" menu. The software pops up a window with a text box for a search term, a text box for the replacement text, a checkbox for case-sensitivity (defaulted to unchecked), and a button labeled "Replace All." The user enters the search term and replacement text into the appropriate text boxes, checks the checkbox, and clicks the button to initiate the search. |
| A user composing email needs to be able to look up contacts in an address book. | A user entering the header information in an email being composed indicates that the current header field should be populated using the address book. The software responds by displaying a list of contacts, including full name and email address. The user specifies one. The software responds by returning to the email and adding the specified name and address to the header field. | The user has the "compose email" window active, selects one of the text boxes that contains the To: or From: header, and clicks the address book button next to the text box. The software responds by popping up a modal dialog that contains the address book entries in a listbox. The user scrolls down to the email address to be added to the field, clicks on the name, and clicks the "Add" button. The software closes the address book window and adds the name to the field that was being edited. |
| A user needs to search the Internet for web pages that contain certain words. | A user indicates a search term to the search engine. The software responds by displaying a list of pages that match the criteria. The list is limited to 100 results. The user may indicate that the next 100 results are to be displayed, in which case the system displays those results instead. | A user navigates to the home page of the search engine. The software responds by displaying a text box and a button labeled "Search." The user enters the search term into the text box and clicks the button. The software responds by displaying a page with 100 results, with 1 link per result. At the bottom of the page is a link labeled "Next 100 hits." If the user clicks on that link, the search engine displays the next 100 hits, with a link labeled "Next 100 hits" and another labeled "Previous 100 hits." |

**SRS Inspection Checklist**

The SRS development script specifies that its last iteration must include an inspection of the SRS. The following checklist can serve as a guide to SRS inspectors. It is divided into sections that provide criteria for evaluating the SRS document in general, the use cases, and the requirements.

The following checklist items apply to the entire SRS.

*Document completeness*

Does the document meet all established templates and standards?

Is the document complete?

Is there any information that should be included, but is not?

Is there any information that should be removed?

Are all of the references valid?

*Document feasibility*

Can the project as specified be accomplished within known cost and schedule constraints?

*Document modifiability*

Is the document structured so that any changes can be made easily, completely, and consistently?

*Document feasibility*

Can every element of the SRS be implemented with the available resources and tools?

The following checklist items apply to the use cases.

*Use case clarity*

Does each use case have a clear flow of events?

Is every action that the system takes performed in response to an action by the user or to a specific event?

Has unnecessary duplication been removed using generalization and/or references?

Is each use case uniquely identified with a name and a number?

*Use case completeness*

Are all of the steps in each use case necessary?

Are there any steps that are missing?

Are all alternative paths and exceptions accounted for?

*Use case level of detail*

Does any use case contain details (such as specific calculations, constraints, or other internals that would not directly be observed by the user) that should really be part of the functional requirements instead?

Does any use case unintentionally constrain the design?

*Use case testability*

Is each use case testable?

The following checklist items apply to the functional and nonfunctional requirements.

*Requirement clarity*

Is each requirement clear, unambiguous, and readable?

Is each requirement uniquely identified with a name and a number?

*Requirement completeness*

Is each requirement complete?

Are there requirements that are missing?

*Requirement level of detail*

Do any requirements unintentionally constrain the design?

*Requirement consistency*

Are the requirements consistent?

Does any requirement contradict another requirement?

Are all data structures, calculations, and functions named and used consistently?

*Requirement functionality*

Is every requirement correct?

Are all inputs and outputs clearly specified?

*Requirement performance*

Are all nonfunctional requirements that constrain performance (speed, resource utilization, etc.) clearly and quantitatively defined?

*Requirement testability*

Is each requirement testable?

**Change Control**

Throughout the course of most projects, many of the people involved come up with changes to the planned software that could be implemented. Many poorly managed software projects have been driven to failure because the designers, developers, and testers had to repeatedly switch directions because of uncontrolled changes. Changes originate from all over the project: a stakeholder may discover a new need that should be addressed, a senior manager could change his mind about a feature, a programmer could figure out a way to combine behaviors to make the software more efficient, a tester could discover conflicting requirements, etc. Some of these changes will be worth doing, while others should probably be scrapped. But every change will come with a sense of urgency, and the project manager needs a way to sort through them to make sure that only the right changes are made.

*Change control* is a method for implementing only those changes that are worth pursuing, and for preventing unnecessary or overly costly changes from derailing the project. Change control is essentially an agreement between the project team and the managers that are responsible for decision-making on the project to evaluate the impact of a change before implementing it. Many changes that initially sound like good ideas will get thrown out once the true cost of the change is known. The potential benefit of the change is written down, and the project manager works with the team to estimate the potential impact that the change will have on the project. This gives the organization all of the information necessary to do a real cost-benefit analysis. If the benefit of the change is worth the cost, the project manager updates the plan to reflect the new estimates. Otherwise, the change is thrown out and the team continues with the original plan.

Changes that seem "small" because they are easy to describe in words can have an unexpectedly large impact on the project. Even the most carefully planned and tracked project can be thrown off course by unexpected changes. A project manager can use change control to keep this from happening.

**Establish a Change Control Board**

The most important part of change control is a *change control board* (CCB ). There are certain people in the organization who have the power to change the scope of the project. Usually there is a senior manager or decision-maker who has the authority to make sweeping changes at will; sometimes there are several people in this position. For change control to be effective, these people must be part of the CCB.

In addition, the CCB should contain people from the project team:

* The project manager
* An important stakeholder or user (or someone who understands and advocates the team's perspective)
* Someone who understands the effort involved in making the change (usually, this is a representative from the programming team)
* Someone who understands the engineering decisions that the team makes over the course of the project (a design team member, requirements analyst, or, if neither is available, a programmer who participated in the design of the software)
* Someone who is familiar with the expected functionality of the software and with the behavior being discussed for each individual change (typically a tester)

This last person fulfills a very important role in the change control process. Typically, she is involved in the tracking of changes and defects in the product. When a bug is reported, part of her job is to figure out whether it is a defect (meaning that the software does not behave the way its specification requires it to behave) or a change (meaning that the software behaves as designed, but that this behavior is not what the users or stakeholders need).

Changes will generally be reported through the defect tracking system. (If there is currently no defect tracking system in place, establishing one will have a far greater impact on controlling changes than a change control process will—see [Chapter 8](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch08.html)). The job of the tester on the CCB is to understand the change: the behavior specified in the requirements, why this behavior is incorrect, and how and why the software should be changed.

Before the project begins, the list of CCB members should be written down and agreed upon, and each CCB member should understand why the change control process is needed and what her role will be in it. The project manager must ensure that everyone buys into the idea of change control and agrees that it is their job to evaluate each change before the software project plan can be altered. The project manager must also reassure them that the programmers and other team members cannot deviate from the plan. This agreement between CCB members is the most important part of the change control process.

**Change Control Process**

[Table 6-14](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch06s04.html#appliedprojectmgmt-CHP-6-TABLE-14) shows the script for a change control process.

*Table 6-14. Change control process script*

| **Name** | **Change control process script** |
| --- | --- |
| Purpose | To control changes by evaluating their impact before agreeing to implement them |
| Summary | The change control process ensures that the impact of each change is evaluated before the decision is made to implement that change. A change is proposed by anyone evaluating the software. A *change control board* (CCB), made up of the decision-makers, project manager, stakeholder or user representatives, and selected team members, evaluates the change. The CCB either accepts or rejects the change. |
| Work products | *Input*  Issue report in the defect tracking system that describes the change  *Output*  Modified issue report that reflects the impact of the change and the decision on whether or not to move forward with it |
| Entry criteria | A change has been discovered, and an issue report that describes it has been entered into the defect tracking system. |
| Basic course of events | 1. A CCB member (typically a tester) who is familiar with the expected functionality of the software reads and understands the issue report, which describes the requested change. 2. The CCB member familiar with the change meets with the project manager to explain its scope and significance. Together, they identify all project team members who will be impacted by the change, and work with them to evaluate its impact. The project manager updates the issue report to reflect the result of that evaluation. 3. At the next CCB meeting , the project manager presents the scope and significance of the change, along with its expected impact. The CCB discusses the change, and performs a cost-benefit analysis to determine if its benefits are worth the cost. The CCB approves the change. 4. The project manager updates the issue report to indicate that the change has been approved, and then updates the project plan to reflect the change. The team members begin implementing the change. |
| Alternative paths | 1. In Step 1, if the CCB member does not understand the change, it can be returned to the submitter for further explanation. The submitter may choose to either update the issue report to clarify the change (in which case, the script returns to step 1) or drop it entirely (in which case, the change control process ends). 2. In Step 3, if the CCB determines that the benefits of the change are not worth the cost, it can reject it. The change control process ends, and no changes are made to the project. The project manager updates the issue report to reflect the fact that it was rejected. |
| Exit criteria | The project plan has been updated to reflect the impact of the change, and work to implement the change has begun. |

**Evaluate Changes in the CCB Meeting**

There are points in the course of the project when the CCB may need to meet regularly:

* During the requirements phase, the CCB will need to discuss the scope of the project if it turns out that there are major areas that the vision and scope failed to cover.
* During the design and programming of the software, the team may discover that the requirements need to be changed. For example, programmers may discover requirements in the SRS that seemed reasonable at the time but that turn out to be contradictory or convoluted, which could be changed to simplify the implementation.
* During the testing phase, the testers may discover omissions in the SRS or design that cause defects. For example, an enhancement to a software project may require that a new record type is to be added to one feature, but the requirements and design fail to specify how that record type is handled in another related feature. The programmers never implemented any handling for this new record type in the second feature because the design failed to indicate how that was to be handled.
* The users or stakeholders may discover during a design walkthrough, demo, user acceptance testing, or beta testing that the software does not fulfill their needs.

During the CCB meeting, the project manager explains the change to the rest of the CCB. Once the CCB is brought up to speed, it must determine what project work products will have to be changed. Each change will affect at least one work product that has already been approved—if this were not the case, the CCB would not have to meet because the change could be handled as part of the regular review process.

The CCB must evaluate every change that is requested. This ensures that nothing slips through the cracks, and that a real decision is made for each request. However, this could mean that at some very busy points in the project, the CCB must meet periodically—sometimes weekly or even daily. At each meeting, it may discuss many changes that have been submitted since the previous meeting.

It is important that when the CCB is being organized, the project manager makes sure that each member understands that this sort of time commitment may be necessary. These meetings are an important way for all of the CCB members to stay on top of the issues that come up during development. This knowledge is very valuable later on in the project when it is time to determine whether the software is ready for release (see [Chapter 8](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch08.html)).

**Analyze the Impact of Each Change**

It is vital that the project manager understand and manage all of the information produced in the change control process. While all of the project team members' opinions are necessary in evaluating the change, it is the project manager who owns the change, and who makes sure that it is properly understood and evaluated. This is generally a lot of work: the project manager needs to take the time to understand why each change is needed, what needs to be changed, and how much work it will be to make the change. His understanding must be complete enough that he can present this information at the CCB meeting.

The project manager who is responsible for guiding each change through the change control process. This is why it is important that the project manager be the one who updates the issue report to reflect both the effort that was estimated in the evaluation and the final decision of the CCB. And it is the project manager who is responsible for making sure that each CCB member is given enough information to understand the change.

During the change control process, the project manager works with the team to evaluate the impact. One effective way to do this is to use the Wideband Delphi estimation process (see [Chapter 3](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch03.html)). If this is done, the project manager can append the results of the Delphi meeting to the issue report and present them to the CCB for cost-benefit analysis. But while a full estimation may be necessary for large changes, there are many small changes that do not require the project manager to gather an estimation team and hold a meeting. For example, a change that amounts to a small bug fix may simply require that the project manager talk to the programmer responsible for that code and ask how long it will take to fix it.

There is some overlap here between the responsibility of the project manager and that of the QA team. Some QA engineers may question the need for the project manager to be so involved in updating specific issue reports in the defect tracking system—this is generally a task that is exclusively controlled by QA. This is true both when the issue is a defect and when it is a change; in both cases, the impact must be evaluated, to determine whether the change should be implemented or whether the defect should be fixed.

The reason the project manager is responsible for this is because he serves as a conduit for all of the information relevant to the change. By updating the issue report, he ensures that he has gathered all the information relevant to the change, and that all of the information is in one place. The issue report is still owned by QA, but it makes sense to have the project manager directly update it rather than having to sit down with the QA team each time a change is evaluated. As long as a member of the QA team on the CCB initiates each change, they will still be kept in the loop.