Is Your Design Clear, Concise, and Complete?

23

The objective of this chapter is to discuss quality assurance (QA) methods that should be used to ensure your design is clear, concise, and complete.

23.1 Preconditions

The following should be true prior to starting this chapter.

- You understand five program design criteria: separation of concerns, design for reuse, design only what is needed, performance, and security.
- You understand the six characteristics of a good software design: simplicity, coupling, cohesion, information hiding, performance, and security.
- You understand the notion of abstraction as a design process. Abstraction is the ability to generalize a concept by removing details that are not needed to properly convey the design perspective being emphasized.
- You have created and/or modified models that described an object-oriented or structured software design.
- You understand how to apply the Model–View–Controller architectural pattern to modify an existing design into these three components.
- You understand how to apply the Model–View–Controller architectural pattern to create a high-level design that accurately reflects the domain requirements while satisfying the design constraints inherent in MVC.

D. P. Voorhees, *Guide to Efficient Software Design*, Texts in Computer Science,
https://doi.org/10.1007/978-3-030-28501-2_23

23.2 Concepts and Context

The terms *quality* and *assurance* often get misused or are used interchangeably to represent the same concept. One way to remember the distinction between these two terms is to understand the history of the terms. The historical perspectives described below are based on the way in which these two terms are being used in this chapter. Quality has its roots in manufacturing, where the goal is to eliminate defects from being present in the final product. Assurance has its roots in security, where the goal is to produce a product that gives the user a degree of confidence that it is safe to use. These terms are now described from a software perspective.

Software Quality: Software is free of bugs/defects and meets user needs.
Software Assurance: Software is free of vulnerabilities and functions as intended.
Software Quality Assurance: Software is free of bugs/defects, free of vulnerabilities, meets user needs, and functions as intended.

While it may be impossible to prove that a piece of *meaningful* software contains no defects or no vulnerabilities, it is still appropriate to have as a goal the complete elimination of defects and vulnerabilities. The notion of proving the absence of defects (or vulnerabilities) is the holy grail of computer science. If some future includes the ability to mathematically prove that a piece of software, no matter how large or complex, contains no defects (or vulnerabilities), than the process of creating software is likely quite different from how we do it today.

While *meets user needs* and *functions as intended* sound like similar goals, these are actually quite different. Meeting user needs has to do with understanding the needs of the users, documenting these needs, and then translating these needs into a design and implementation. Throughout a software development process, the two significant challenges are (1) misinterpreting someone's needs, whether expressed verbally or in writing, and (2) understanding and dealing with the fact that user needs will change over time. In contrast, functions, as intended, is more of an engineering goal. If the requirements say the software must do X, Y, and Z, can we demonstrate that the software indeed does X, Y and Z? The challenge with this goal is to ensure that the software meets the requirements without also having unintended consequences. For example, perhaps the X, Y, and Z features are indeed in the software, but using these features in a certain way also allows the user to do Q, which is not a requirement. If feature Q happens to be a vulnerability then our unintended consequence has resulted in a less safe piece of software.

To do the best we can in ensuring our software quality assurance goals have been met, we need to think about the following four questions. The sections that follow will describe quality assurance methods which may be used to address one or more of these questions.

- What can we do to find as many bugs/defects as possible?
- What can we do to find as many vulnerabilities as possible?
- What can we do to determine whether user needs are being met?
- What can we do to determine whether the software functions as intended?

The thinking regarding the first two questions is perhaps obvious, but very important—the more defects and vulnerabilities we find and correct during software development, the fewer of these will exist once the software is deployed.

23.2.1 Software Quality Assurance

Software quality assurance is also described as a collection of verification and validation methods. These two terms are described using the following two questions [1].

1. Are we building the product right?
   This is called *software verification*. We are building the product right if the software meets the requirements and design specifications. We can (and should) apply verification methods throughout the entire software development process.
2. Are we building the right product?
   This is called *software validation*. We are building the right product if the software is meeting user needs and the requirements and design specifications are correct in the first place. We can (and should) apply validation methods throughout the entire software development process.

The remaining subsections introduce some of the common verification and validation methods.

23.2.2 Formal Review

A formal review may verify and/or validate a software artifact.[1] The formal review process described below is modeled after the Fagan inspection process [2].

A formal review is a process that involves a moderator, one or more inspectors, and an author. The moderator assigns a specific task to each inspector and gives each inspector the maximum amount of time to spend on the task. Each inspector looks at the artifact(s) under review and logs any defects that are found. An inspection meeting is then held which is facilitated by the moderator and attended by all inspectors and the author. During this meeting each inspector reads their list of defects out loud. There is no debate about a defect—if an inspector identifies something as a defect, then it's a defect. However, the author may ask questions to clarify their understanding of a defect. After the inspection meeting the author is tasked with addressing each defect. By *addressing each defect*, the author may correct the defect or provide an explanation of why it is not a defect. The moderator and author would then review the defect logs and the way in which the author addressed each defect.

[1] A software artifact is a document produced during a software development process; it contains descriptions of the software. An artifact may describe project plans, user needs, design decisions, code, tests, or some combination of these.

A few key points regarding this process are as follows:

- The moderator should have experience as an inspector and (ideally) in managing the inspection process.
- The entire inspection process is carefully managed from a time/resource perspective. The adage *time is money* is certainly applicable in this case. Software development projects usually have a budget. This budget would (hopefully) include costs (i.e., time and resources) to perform various quality assurance methods—including formal reviews. Giving each inspector the maximum amount of time to spend on the task they've been assigned allows the moderator to manage the costs of each formal review.
- The lack of any debate regarding the merits of a specific defect may seem odd. This will be addressed below as the other QA methods are being introduced.

The following sections provide a detailed description of the formal review process.

23.2.2.1 Formal Review: History

The process described below is based on Fagan's inspection process. Michael Fagan, while working for IBM in the 1970's, developed an inspection process that was used during the development of the S/360 operating system [2].

23.2.2.2 Formal Review: Roles and Process Steps

There are three types of participants in a formal review. Each participant adheres to one of the following roles.

Moderator: The leader of the inspection, manages the inspection process.
Tester: An inspector; someone that is assigned a specific role as a reviewer of artifacts.
Author: The primary author of the artifact(s) being reviewed or a representative from a group of authors.

The steps to be followed when doing a formal review are as follows:

1. Plan Inspection
2. Initial Meeting
3. Preparation
4. Inspection Meeting
5. Rework
6. Follow-up

23.2.2.3 Formal Review: Process Details

A detailed description of each step is provided below.

1. Plan Inspection
   The moderator performs this step, which involves

   a. Collecting the artifacts (i.e., documents) to be inspected.
   b. Collecting other artifacts that were used to develop the document(s) under inspection.
   c. Identifying and obtaining commitment from testers.
   d. Scheduling two meetings: the initial meeting and the inspection meeting.
   e. Assigning each tester a specific responsibility (see Sect. 23.2.2.4 for details).
   f. Indicating how much time each tester should take to inspect the artifact(s).

2. Initial Meeting
   The initial meeting is attended by the moderator, all testers, and the author. The moderator facilitates this meeting, which involves

   - Describing to each tester their responsibility and the time they have to complete their inspection.
   - Making available the artifact(s) to be inspected and any other related documents.

3. Preparation
   Each tester performs the following to prepare for the inspection meeting.

   a. Inspect artifact based on their responsibility (see Sect. 23.2.2.4 for details).
   b. Record each defect found in the artifact(s) being inspected on a defect log.
   c. Record the time taken to do the inspection.
   d. Submit their defect log to the moderator prior to the inspection meeting.

   The moderator will collect the defect log from each tester and make a copy of each defect log.
4. Inspection Meeting
   The inspection meeting is attended by the moderator, all testers, and the author. This meeting involves the following.

   - The moderator controls the meeting. There is *no arguing* during the meeting about whether something is a defect. If a tester says it's a defect, then it's a defect!
   - The moderator hands the original copy of each defect log to the appropriate tester.
   - The moderator hands a second copy of each defect log to the author.
   - One at a time, each tester reads aloud their defects, using their defect log. The author may ask questions to clarify their understanding of a defect. The author may update their defect log with additional notes to further document a defect.

5. Rework
   The author (or group of authors) will then *resolve* each defect. *Resolving* a defect may result in either

   - The artifact(s) being updated to correct the defect.
   - Or an explanation as to why this is not a defect.

   In either case, the defect resolution is recorded on the defect log.
6. Follow-up
   The moderator reviews the defect log updated by the author(s). This review is done to ensure that each defect has been *resolved*. The moderator may decide to schedule a second inspection if the number of defects found or the severity of the defects warrants a second review.

23.2.2.4 Formal Review: Tester Responsibilities

A tester has the following responsibilities.

- Find as many defects as possible!
- Focus on the particular type of inspection they've been assigned by the moderator. The types of inspections include the following.
  - Clear and Concise
    - Is the artifact easy to understand?
    - Are the diagrams and text written in a clear understandable manner?
    - Is the artifact verbose (overly wordy)?
  - Complete
    - Does the artifact include everything it needs to include?
    - Does the artifact include extraneous information?
  - Consistent
    - Is the artifact consistent with itself?
    - Is the artifact consistent with other artifacts in the project?
- For complete and consistent defect types
  - The tester may be given other documents that were used to develop the artifact under inspection.
  - The tester uses these other documents to check for completeness or consistency.

Table 23.1 Defect log: header

Project:
Artifact:
Author:
Date:

Table 23.2 Defect log: defect recording section

Number	Type	Inject phase	Removal phase	Fix time	Fix reference
Description:					

23.2.2.5 Formal Review: Example

Let's assume that a design document is to be inspected. The requirements document would have been used (as an input document) to develop the design. In this scenario, the moderator could assign tester responsibilities as follows:

- Tester 1 is assigned clear and concise. An example defect would be *the design artifact has a sentence (page 1, paragraph 3) that contains 60 words*.
- Tester 2 is assigned complete, and is given both design and requirements artifacts. An example defect would be *the design artifact is missing a design element for requirement 5.4*.
- Tester 3 is assigned consistent. An example defect would be *the design artifact uses terms* database *and* file *to mean the same thing. The design should consistently use one of these terms*.
- Tester 4 is assigned consistent, and is given both design and requirements artifacts. An example defect would be *the design artifact uses the term loan while the requirements artifact calls it a liability*.

23.2.2.6 Formal Review: Defect Log

A typical defect log has a header that identifies project, artifact, author, and date of the inspection. This is shown in Table 23.1.

Each defect is recorded in a section of the log that repeats itself to fill-out a page. Table 23.2 shows a typical defect recording section.

The information specified for each defect is now described. The tester would fill-out the Number, Type, Inject Phase, and Description fields during step 3 Preparation. The author would fill-out the Removal Phase, Fix Time, and Fix Reference during step 5 Rework.

Number: Enter the defect number. For each artifact, use a sequential number starting with one.

Type: Enter the defect type number from the list of defect types described in Table 23.3. Use your best judgment in selecting which type applies.

Inject Phase: Enter the SDP phase during which the defect was injected.

Removal Phase: Enter the SDP phase during which the defect was fixed (corrected).

Fix Time: Enter the time, in hours, that you took to find and fix the defect. This time is often an approximate value.

Fix Reference: If you or someone else injected this defect while fixing another defect, record the number of the improperly fixed defect. Leave blank when the fix is not associated with a prior defect incorrectly fixed.

Description: Write a succinct description of the defect that is clear enough to later remind someone of the error and how or why it was made.

Table 23.3 Formal review defect types

Type	Name	Description
10	Documentation	Design: does not conform to guidelines Code: comment not appropriate/correct, comment can be misunderstood
20	Syntax	Design: spelling or grammar mistake Code: spelling mistake, format of code inconsistent or hard to read
30	Build, Package	Code: change management not documented, library use not appropriate/incorrect, version control not documented
40	Assignment	Code: variable name not descriptive, using duplicate variable name leads to misunderstanding, variable scope not appropriate
50	Interface	Design: missing an interface design, logical error in interface design, technical error in interface design, HCI design is confusing or inefficient Code: function/method calls are too complex, I/O is too complex
60	Checking	Design: validation is missing, has logical errors, or has technical errors Code: error messages are vague/incorrect, missing validation checks
70	Data	Design: missing a logical or physical data design, error in logical data model, error in physical data model Code: data structure use not appropriate
80	Function	Design: missing a functional requirement, logical error found, technical error found Code: logic error found, use of pointers/references not correct, selection, iteration, or recursion errors
90	System	Design: architecture incorrect or missing a component Code: does not match design
100	Environment	Code: design, compile, test, or other support system problems

23.2.2.7 Formal Review: List of Defect Types

Table 23.3 identifies different types of defects that may be found in an artifact. This list is not intended to be complete but rather serve as a basis for developing a more complete list of defect types. An organization should develop their own defect types and use these consistently in their defect logs. This allows an organization to measure quality improvements for each defect type, and develop training based on defect types that consistently appear in their artifacts.

23.2.3 Informal Review

An informal review may verify and/or validate a software artifact. There are a few variations in how an informal review is conducted. The description below is intended to provide a sense for how an informal review is performed and begins to explain the differences between a formal and informal review.

In an informal review, each reviewer looks at the artifact(s) and identifies the defects they find. A review session is then held which is attended by all reviewers and the author. The reviewers and the author discuss each defect found by a reviewer. Consensus should be obtained about the list of defects that need to be corrected. After the review session, the author is tasked with fixing each identified defect.

One key point regarding an informal review

- I have participated in these types of informal reviews where there was no facilitator (aka moderator) for the discussion. Instead, the artifact author or one of the reviewers *took charge* of the meeting as an unofficial moderator. This can be an effective approach assuming that the individuals involved consider these discussions as constructive criticism of the author's work. However, I've seen situations where discussions become emotional as egos clash between the artifact author and one or more reviewers. In these situations, the effectiveness of the informal review is greatly diminished, and participants walk out of the session thinking it was a waste of their time.

The lesson I've learned from these experiences is to have an individual facilitate the informal review session without also being an author or reviewer. This allows the facilitator to (hopefully) remain objective during discussions.

23.2.4 Design/Code Walkthrough

A design or code walkthrough may verify and/or validate a software artifact. There are a few variations to how a walkthrough is conducted. The description below is intended to provide a sense for how a walkthrough is performed and begins to explain the differences between a formal review, an informal review, and a walkthrough.

In a design or code walkthrough, a meeting is held that is attended by all reviewers and the author. At this meeting, the author describes each artifact in a fair amount of

detail. As the author is walking through an artifact, the reviewers may comment on a particular aspect of the artifact that they feel could be improved. Consensus should be obtained about the parts of each artifact that should be modified/improved. After the meeting, the author is tasked with making each modification/improvement. The only substantive difference between a design walkthrough and a code walkthrough is the type of artifact being reviewed (i.e., review a structure chart or class diagram versus review source code).

Similar to the discussion regarding informal reviews, having the right person facilitate discussions during a walkthrough is an important consideration.

23.2.5 Customer Survey

A customer survey is typically done to validate a software artifact (i.e., does the software meet your needs?). However, a survey instrument could be created that verifies a software artifact (i.e., does the software satisfy the requirements?). A customer survey is generally completed anonymously and has more significance the larger the number of survey responses obtained.

Using a customer survey as a validation method tends to focus on one or more of the following.

- Assess the level of satisfaction from the user community.
- Elicit feedback on parts of system that are used and/or not used.
- Identify new or changing needs that the system must meet.

23.2.6 Software Testing

Software testing may verify and/or validate a software artifact and is typically applied only to source code artifacts (or only to an executable image produced from source code artifacts). While there are a handful of different types of software testing that may be performed, at the core of software testing is the notion of a test case. In essence, doing software testing involves developing test cases and then performing/executing each test case.

In its simplest form, a test case identifies the input data to be injected into the software and the expected results. When the documented expected results do not match the actual results produced by the software, a defect exists. This defect is either located in the software (i.e., we've found a bug) or in the test case (i.e., perhaps we did not accurately describe the expected results). Test code is often written as a way to implement test cases. In this case, the test code may automatically verify the results or may leave it up to visual inspection to confirm the results match what is expected.

Generally, there are two types of test cases. Note that a test case may satisfy both types described below.

White box: A single case that tests a specific control flow through a function/method, component, or program. Each test case is developed by knowing what the code does and how it does it.

Black box: A single case that tests one or more requirements or use cases. Each test case is developed by knowing the requirements (i.e., what the software is supposed to do).

Four different types of software testing are now described.

23.2.6.1 Unit Testing

Unit testing is generally done as the code is being developed. It's purpose is to verify that each function/method performs as it was designed. Thus, unit testing is performed on the smallest units of code—statements, functions, and methods. It is a verification method since it is looking at whether the design specifications were correctly implemented. It is *not* a validation method since unit testing does not focus on user needs.

When creating unit tests, each test case is written to test an individual function or method. That is, each function/method is tested independent of all other functions/methods. Each unit test case is a white box test case since it is written with knowledge of what the code is doing and how it is doing it. At a minimum, the number of unit test cases for a function/method should equal the number of distinct control flow paths through the function/method.[2]

23.2.6.2 Integration Testing

Integration testing is generally done as the code is being developed. It's purpose is to verify the design of each component/module that makes up the software application or system. Thus, integration testing is about testing the interactions between functions/methods that are combined to form a higher level design element (e.g., a component or module). For example, a Model–View–Controller design would have a set of integration test cases for the model component, another set of test cases for the view component, and a third set of test cases for the controller component. It is a verification method since it is looking at whether the design specifications were correctly implemented. It may also be a validation method if some of the cases are testing user needs. (Here, the user need may be expressed as a fundamental use case or requirement. By fundamental, I mean that the user need is so basic to the software that it is often implied by users to exist. For example, a user wanting to have a software program playing a card game would likely assume that there is a deck

[2]While the topic of creating white box test cases is beyond the scope of this book, an excellent resource to learn more about developing test cases is *The Art of Software Testing* by Glenford J. Myers. This book describes other types of software testing not included in this book and also discusses debugging, extreme testing, and testing Internet applications. The second edition of this book was published in 2004.

containing 52 cards, with 13 distinct card values in each suit and 4 distinct suits. The need for a deck having 52 cards may go unsaid by the user.)

When creating integration tests, each test case is written to test an interaction between two or more functions or methods. Most integration test cases are white box since they are written with knowledge of what the code is doing and how it is doing it. Some of the unit test cases may be used for integration testing, but it is expected that new cases are developed to specifically test the integration of functions/methods within a component or module. Black box test cases may also be developed during integration testing. These cases would address user's needs and represent the validation portion of the integration test.

23.2.6.3 System Testing

System testing is generally done at certain project milestones and after the code has been integrated. It's purpose is to verify the software meets requirements, as understood by developers. Thus, system testing is about testing all of the components/modules that make up the application or system. It is a verification method since it is looking at whether the design specifications were correctly implemented. It is also a validation method since it would now include the testing of use cases or requirements, both documented and implied.

When creating system tests, each test case is written to test a specific scenario or processing flow through the entire system. Some of the unit and integration test cases may be used for system testing, but it is expected that new cases are developed to test the entire system. New white box test cases would focus on the interaction of the components that make up the system while new black box test cases would focus on user needs.

23.2.6.4 Acceptance Testing

Acceptance testing is generally done by end-users. It's purpose is to validate the software meets their needs. Thus, acceptance testing is about testing those features the user community feels are most important or relevant.

Acceptance testing would involve the creation of all new test cases since this type of testing is most often performed by users. All of these test cases would be black box and are designed to validate a user's need. Thus, acceptance testing is a validation method but *not* a verification method.

23.2.7 Summary of QA Methods

Table 23.4 shows a summary of the non-testing QA methods, but does not include the customer survey method. This table shows the differences between the three types of *review* methods as described in this chapter. Please note, the description of these QA methods may be inconsistent with other sources, either due to terminology used or the culture within the information technology organization.

Table 23.4 Summary of non-testing methods

	Formal review	Informal review	Design/Code walkthrough
Well-defined process?	Yes	No	Maybe
Participants come prepared?[a]	Yes	Yes	No
Author describes artifact?[b]	No	No	Yes
Discussion/consensus on defects?	No	Yes	Yes
Example use as verification	Check design against requirements		
Example use as validation[c]	Check requirements against use cases		Check design against use cases

[a] When a participant comes prepared, they've identified and listed defects they found
[b] Author steps through artifact and explains its content
[c] Assumes use cases were developed by users

Table 23.5 Summary of testing methods

	Unit	Integration	System	Acceptance
Uses white box test cases?	Yes	Yes	Yes	No
Uses black box test cases?	No	Yes	Yes	Yes
Does verification?	Yes	Yes	Yes	No
Does validation?	No	Maybe	Likely	Yes
Done by programmer?	Yes	Likely	Maybe	No

Table 23.5 shows a summary of the testing QA methods. This table shows the differences between the four types of *testing* methods. These descriptions may be inconsistent with other sources, either due to terminology used or the culture within the information technology organization.

To conclude the goal of verification and validation methods are to

- Find as many defects as possible
- Find as many vulnerabilities as possible
- Ensure software meets user needs
- Ensure software functions as intended

23.3 Software QA in Software Design

Learning to design software is challenging for many reasons, one of which is the fact that we cannot test a design like we can test code. How do we know whether our design is any good if we cannot test it? The answer is hopefully obvious at this

point—the use of formal review, informal review, and design walkthrough allows us to assess the quality of our design based on the results obtained from using one of these QA methods.

23.3.1 Formal Review of ABA Software Design

The author strongly recommends the use of the formal review method when assessing the quality of a software design. The formality of this review method is a benefit, but just as important is the fact that this method avoids debate about what is good and not so good about a design. While most professional environments would result in a healthy discussion during an informal review or design walkthrough, the author has seen instances where these discussions get too personal. In these situations, an author's ego might prevent them from acknowledging the defects or arguing in defense of their design. Or, a reviewer might express their opinion too strongly resulting in a degradation in the professionalism of the review meeting. In contrast, an experienced moderator will facilitate the inspection meeting (which is part of the formal review process) to avoid a confrontational environment.

23.3.1.1 Formal Review of object-oriented design (OOD)

For the case study, we have a list of requirements and five design artifacts. For the OOD approach described in Chap. 15, the ABA Version B solution has a package diagram, a class diagram, a statechart, and two communication diagrams. A moderator may look at the scope of the requirements and the number of design artifacts and conclude that two inspectors are all that is needed.

1. One inspector would spend 30 min looking at the requirements and the design artifacts to ensure all of the requirements have been addressed in the design.
   For example, the two validation requirements should result in a design that continues to obtain user input until a valid value has been entered. The statechart should show the validation via states and transitions. Similarly, the OOD communication diagram should list a sequence of messages that shows validation being performed. When the inspector cannot confirm these requirements are in the design, a defect would be identified and documented on the defect log.
2. The second inspector would spend 30 min looking only at the design artifacts to ensure they are clear, concise, and consistent with each other.
   For the OOD approach, any method specified in the communication diagram should be listed in the correct class within the class diagram. Any deviation between these two diagrams results in a defect being identified and documented on the defect log. A defect resulting in a simple correction is a misspelling of a method name in one of the diagrams. A more complex defect might be when there are many methods specified in the communication diagram that does not appear anywhere in the class diagram.

23.3.1.2 Formal Review of structured design (SD)

For the case study, we have a list of requirements and nine design artifacts. For the SD approach described in Chap. 16, the ABA Version B solution has three structure diagrams, five structure charts, and a statechart. A moderator may look at the scope of the requirements and the number of design artifacts and conclude that two inspectors are all that is needed.

1. One inspector would spend 30 min looking at the requirements and the design artifacts to ensure all of the requirements have been addressed in the design.
   For example, the two validation requirements should result in a design that continues to obtain user input until a valid value has been entered. The statechart should show the validation via states and transitions. Similarly, the SD structure diagrams and structure charts should identify validation functions being called. When the inspector cannot confirm these requirements are in the design, a defect would be identified and documented on the defect log.
2. The second inspector would spend 30 min looking only at the design artifacts to ensure they are clear, concise, and consistent with each other.
   For the SD approach, a function specified in a structure diagram should also appear in a structure chart, as long as the two diagrams represent the same component or module within the design. Any deviation between these two types of diagrams would result in a defect being identified and documented on the defect log. A defect resulting in a simple correction is a misspelling of a function name in one of the diagrams. A more complex defect might be when there are many functions specified in the structure diagram that do not appear anywhere in the structure chart.

23.3.2 Design Walkthrough of ABA Software Design

A design walkthrough is an effective QA method when there are some experienced reviewers participating in the walkthrough. As indicated earlier, having a facilitator of the walkthrough session who is respected by all participants can help avoid disagreements from spilling over into unprofessional behavior.

For the OOD case study, we have a list of requirements and five object-oriented design artifacts. For the OOD approach described in Chap. 15, the ABA Version B solution has a package diagram, a class diagram, a statechart, and two communication diagrams. A moderator may look at the scope of the requirements and the number of design artifacts and conclude that two reviewers are all that is needed.

For the SD case study, we have a list of requirements and nine design artifacts. For the SD approach described in Chap. 16, the ABA Version B solution has three structure diagrams, five structure charts, and a statechart. A moderator may look at the scope of the requirements and the number of design artifacts and conclude that two reviewers are all that is needed.

Given the small scope of the case study, having two reviewers and the author participate in a walkthrough would be appropriate. One of the two reviewers should

have significant experience. When the reviewers are not familiar with the project, the author would likely begin by describing the requirements within scope of this design. As the author explains each design artifact, the reviewers would ask questions to clarify their understanding of the design. Any possible flaws discovered by a reviewer would be discussed and a determination would be made as to whether a defect has been found. When consensus is achieved, the author would record the defect so it can be corrected at a later point.

A challenge in doing a design walkthrough is the temptation to also discuss ways to correct a defect that has been found. This should be avoided during the design walkthrough session. Why? First, any discussion about correcting a defect may take a significant amount of time. This time is better spent focusing on identifying as many defects as possible. Second, discussing ways to correct a defect will distract the reviewers from their primary responsibility—to identify defects. When a reviewer has ideas about how to correct a design defect, this should be noted by the author. The author can then follow-up with the reviewer at a later time.

23.4 Software QA in Software Development

Learning to develop software is challenging for many reasons, one of which is the many different skills needed to effectively develop quality software. These skills include planning the work to be completed, analyzing the needs of the users, developing design alternatives and selecting a design to implement, implementing the selected design, testing your implementation, distributing the resulting software to the user community, and then supporting the software once it starts to be used.

The quality assurance methods described in this chapter may be used at many points during a software development project. Some examples of when to use these methods follows.

- Have experienced project managers do an informal review on the project plan. These reviewers may identify important milestones or tasks that need to be included in the plan.
- Have selected users do a formal review on the requirements document. The requirements may be assessed based on a high-level scope document previously produced or on a collection of use cases developed by the user community.
- Have other designers do a formal review on the design. These inspectors could assess the design based on the requirements document, based on the technical merits of the design, and based on whether the design is clear, concise, and complete.
- Have experienced designers do a design walkthrough. These reviewers may identify important technical shortcomings of the design or may think the design can be simplified.
- Have experienced testers do an informal review on the test plan/strategy and test cases. These reviewers may identify additional test cases designed to detect certain vulnerabilities in the software.

23.5 Post-conditions

The following should have been learned when completing this chapter.

- You understand the purpose of the following quality assurance methods—formal review, informal review, design/code walkthrough, unit testing, integration testing, system testing, and acceptance testing.
- You've assessed the quality of design artifacts using a formal review, informal review, or design walkthrough.

Exercises

Discussion Questions

1. Software quality assurance includes activities like formal reviews and design walkthroughs. How might these types of activities be used to improve the security of a design?
2. What are some reasons why a formal review is better than an informal review in discovering defects or vulnerabilities?
3. Is there a situation where an informal review may be better to use than a formal review?

Hands-on Exercises

1. Use an existing design solution you've developed and do a formal review to identify as many defects and vulnerabilities as possible. Improve your design based on the list of defects in the defect log.

References

1. Boehm B (1981) Software engineering economics. Prentice Hall, Upper Saddle River
2. Fagan M (1986) Advances in software inspections. IEEE Trans Softw Eng 12(7)