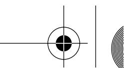
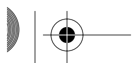
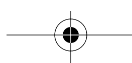
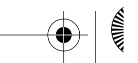


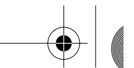
Testing Analysis and Design Models

- **Want to learn how to inspect the semantics of UML models? See The Basics of Guided Inspection on page 116**
- **Need to set up an inspection session? See Organization of the Guided Inspection Activity on page 120**
- **Need a technique for testing a model for extensibility? See Testing Models for Additional Qualities on page 151**

Chapter 4

Developers model the software they are constructing because it assists in understanding the problem they are solving and because it helps manage the complexity of the system being developed. The models of analysis and design information will eventually be used to guide the implementation activities of the project. If the models are of high quality, they make a valuable contribution to the project, but if they contain faults, they are equally detrimental. In this chapter we present **guided inspection,** an enhanced inspection technique for verifying the models as they are created and for validating the completed models against project requirements. The principal shortcoming of standard review techniques is that they focuse primarily on what *is* there (in the model) rather than what *should be* there. Reviews do not provide a means for systematically



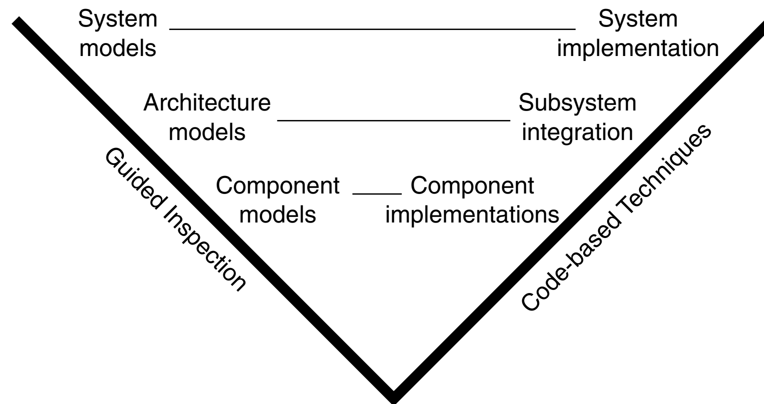


Figure 4.1 The new V model

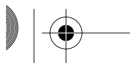
searching for omissions from products. Even Fagan inspections [Faga86], which use checklists to make the process more detailed, do not provide a means for determining what is missing from a model.

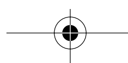
Guided inspection applies the testing perspective very early in the development process. Traditionally, testing has begun at the unit implementation level and has continued as code segments are integrated into larger pieces until the entire system is available to be tested. In this chapter, we will begin "system testing" when the "system" is still represented only as analysis or design information. A new version of the traditional "V" testing model, shown in Figure 4.1, relates the repeated applications of guided inspections to the various levels of testing.

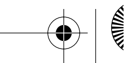
Guided inspection requires valuable resources, and the time and attention of project personnel. So is it worthwhile? Studies have reported widely varying savings ratios for finding and repairing faults early in the development process as opposed to during the compilation or system test phases. For example, repairing a fault found at system test time may cost as much as one hundred times the cost of repairing the same fault during analysis. So even a moderate effort at testing the models can result in big savings.

An Overview

Let's look at a quick example of using the guided inspection technique. To set the stage, we are in the initial stages of developing *Brickles*. The team has produced the design-level class diagram shown in Figure 2.18 and other diagrams such as the state diagram shown in Figure 2.19 and the sequence diagram shown in Figure 2.20. We are about to begin coding but want to validate the design model before spending extensive time coding the wrong definitions.







JMCH04.fm Page 111 Wednesday, December 27, 2000 10:17 AM

We begin by assigning the inspection team. The team includes the two of us who developed the model, a system tester from our company and our company's process person who will be the moderator. The tester will develop a set of test cases from the use-case diagram. We developers will show how the classes in the design model handle each test case. The moderator will define the inspection boundaries, schedule the guided inspection session, distribute materials, keep the session moving forward, and then complete the final report.

In preparation for the session, the moderator defines the boundaries of the inspection by identifying the scope and depth of the information to be inspected. The scope is defined by a set of use cases. In our case, the scope covers all the use cases and thus the complete application. The depth is defined by identifying levels of containment in the composition hierarchies. In the case of *Brickles*, we will not inspect the objects that are aggregated within the BricklesView object. We will focus instead on those that represent the state of the match at any given time in a BricklesDoc object.

The tester writes test cases using the use cases found in Figure 2.11. We will focus on one test case, shown in Figure 4.2. Before the meeting, the developers complete the Design Model checklist shown in Figure 4.3. This exercise is completed individually by each developer. It requires that the developer compare the class diagram from the analysis model, shown in Figure 2.13, with the class diagram in the design model. Finally, the moderator sends out notice of the meeting along with either paper copies of the model or a URL to the web version.

The test report from the guided inspection section notes the problems found during the symbolic execution of the test cases. With regards to the test case being considered here, the design is considered to have failed the test. The test report would reflect that it was not possible to determine how to complete the symbolic execution at this point in the algorithm. We do not want to confuse testing and debugging, but since we know exactly where execution terminated, it should be reported. The test report also includes the sequence diagram used to record the test execution, as shown in Figure 4.4.

The use case: A player stops a *Brickles* match by selecting Quit from the File menu.

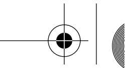
Preconditions: The player has started the *Brickles* game, hasmoved the paddle, and has broken some bricks.

Test input: The player selects Quit.

Expected output: All game action freezes and the game window disappears.

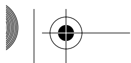
Figure 4.2 Test case #1

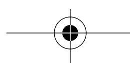
JMCH04.fm Page 112 Wednesday, December 27, 2000 10:17 AM

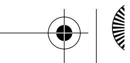


UML Detailed Design CheckList Questionnaire	***Yes***	***No***
Analysis-to-design model transformation issues. Are all classes in the analysis model that are not in the design model outside the scope of the application?	✔	
Are all the states in the analysis-model statecharts also states in the statechart diagrams in the design model?		✔
Are the sequences of messages in all design-level sequence diagrams the same, even though additional messages may have been inserted between the analysis-level messages?		✔
Internal design model issues. Are all associations shown with no navigation information that is truly bidirectional.		✔
Are all composition relationships shown as unidirectional?		
Is every sequence diagram a subset of some activity diagram?		
Does every message sent in an interaction diagram appear as a method in the public interface of the class of the receiving object?		
Does every message sent in an interaction diagram go to the logically appropriate object?		
Are the transitions out of a state diagram mutually exclusive?		
Do all state machines, except for perpetual objects, contain initial and final states?		
Are all public modifier methods represented as transitions on each state even if they only result in staying in the same state?		
Is there a sequence diagram for each postcondition clause of each method that corresponds to use cases that meet the frequency/criticality threshold?		
Are all messages shown correctly as synchronous or asynchronous?		
Do the number of forks and joins balance in every activity diagram?		

Figure 4.3 Example design phase checklist







A Portion of a Session Transcript

MELISSA (moderator): OK, let's get started. Everyone has had a chance to look at the model and Dave and John have completed the checklists. Let me remind everyone that we are focusing on the locally designed classes and will ignore the standard user-interface classes such as menus. So let's begin with the first test case, Jason.

JASON (tester): Here is the first case [he hands out the first test case]. With John's help, I've laid out the beginnings of a sequence diagram for this test case based on the test-case preconditions [see Figure 4.4]. So, the player selects Quit from the menu and...

JOHN (developer): My aBricklesView would receive the Quit message. And my object would send the Quit message to Dave's aBricklesDoc. [He draws these onto the sequence diagram.]

DAVE (developer): When aBricklesDoc gets the Quit message it will send the Stop message to aTimer. [He begins to draw this on the sequence diagram.]

JASON: Wait a minute! As I read the class diagram there is no association between those two classes.

DAVE: John, where is that Stop message suppose to go? I thought you said you were going to implement that method.

JOHN: [Busy shuffling between the sequence diagram and the class diagram with a confused look on his face.] Defect!

MELISSA: Sounds like we are ready for the next test case.

JMCH04.fm Page 114 Wednesday, December 27, 2000 10:17 AM

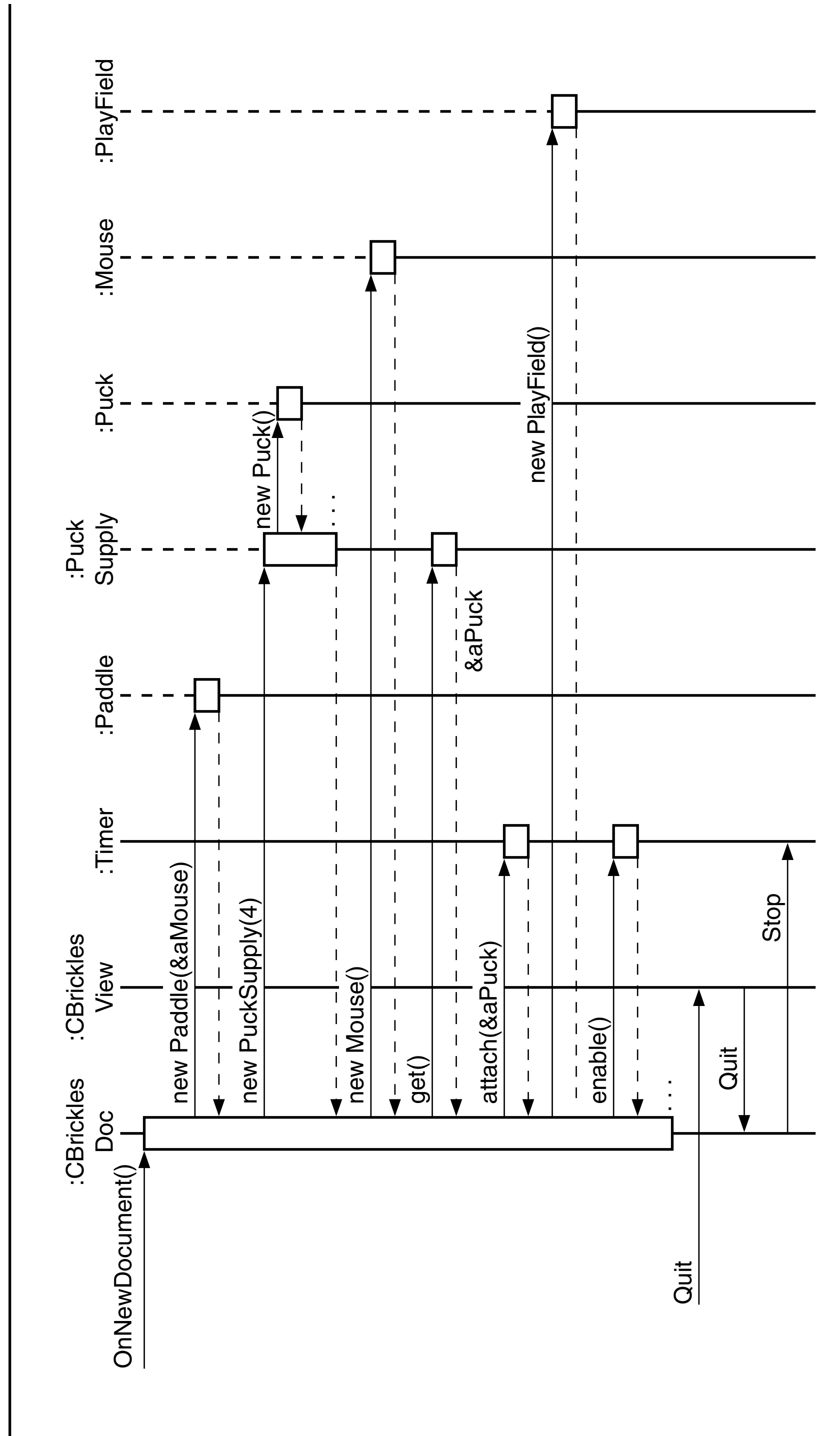
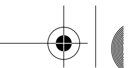


Figure 4.4 Partial sequence diagram for test case 1

JMCH04.fm Page 115 Wednesday, December 27, 2000 10:17 AM

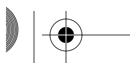
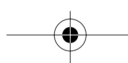
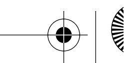


Place in the Development Process

The last activity in each phase in the development process should be a verification that the work produced in the phase possesses the qualities we desire. That work is in the form of either a UML model or code in a programming language. (See A Development Process Overview on page 66) The process is structured so that each development phase moves the product a step toward the final system, which results in a sequence of models in which the model produced in one development phase is more specific and more complete than the model from the previous phase. For example, during the application-analysis phase, a model is created by filtering the information in the domain-analysis model and the requirements model to eliminate information that is not specifically relevant to the application under development. Two of the differences between the succeeding models are the scope of the content and the level of abstraction. The requirements model filters the domain model so that any information not required for the immediate application is not included in the application-analysis model. As the information in each succeeding model level becomes more specific, the inspection of each model can also become more specific and narrowly focused. This sequence of models described in Figure 4.5—actually it is just one model that is being transformed incrementally—provides an opportunity to establish a "chain of quality" in which each model is verified before moving to the next phase.

Phase/Model	Content	Transformed from...
Domain Analysis	Domain concepts, standard algorithms	The minds of the domain experts
Application Analysis	Concepts needed to explain the specific problem; standard algorithms	The domain-analysis model and the requirements model
Architectural Design	Basic structure of interfaces and their interactions	Standard architectural patterns and creativity of designers
Detailed Design	Each interface in the architecture is implemented by one or more components	Architectural-design model and standard design patterns and algorithms

Figure 4.5 Models and phases

JMCH04.fm Page 116 Wednesday, December 27, 2000 10:17 AM

The Basics of Guided Inspection

The **guided inspection** technique provides a means of objectively and systematically searching a work product for faults by using explicit test cases. This testing perspective means that reviews are treated as a test session. The basic testing steps are as follows:

1. Define the test space.
2. Select values from the test space using a specific strategy.
3. Apply the test values to the product being tested.
4. Evaluate the results and the percentage of the model covered by the tests (based on some criteria).

These steps are specialized to the following steps (we will elaborate on each of these in this chapter):

1. Specify the scope and depth of the inspection. The scope will be defined by describing a body of material or a specific set of use cases. For small projects, the scope may always be the entire model. The depth will be defined by describing the level of detail to be covered. It may also be defined by specifying the levels in aggregation hierarchies on certain UML diagrams in the model under test (MUT).
2. Identify the basis from which the MUT was created. The basis for all but the initial model is the set of models from the previous development phase. For example, the application-analysis model is based on the domain-analysis model and the use-case model. Initial models are based on the knowledge in the heads of select groups of people.
3. Develop test cases for each of the evaluation criteria to be applied using the contents of the basis model as input. (see Selecting Test Cases for the Inspection on page 123.) The scenarios from the use-case model are a good starting point for test cases for many models.
4. Establish criteria for measuring "test coverage." For example, a class diagram might be well covered if every class is touched by some test case.
5. Perform the static analysis using the appropriate checklist. The MUT is compared to the basis model to determine consistency between the two diagrams.
6. "Execute" the test cases. We will describe the actual test session in detail later in this chapter.
7. Evaluate the effectiveness of the tests using the coverage measurement. Calculate the coverage percentage. If 12 of the classes from a class diagram containing 18 classes have been "touched" by the test cases, the test coverage is 75%. The testing of analysis or design models is so high-level that 100% coverage is necessary to achieve good results.

JMCH04.fm Page 117 Wednesday, December 27, 2000 10:17 AM

8. If the coverage is insufficient, expand the test suite and apply the additional tests, otherwise terminate the testing. Usually the additional test cases cannot be written during the inspection session. The testers identify where the coverage is lacking and work with a developer to identify potential test cases that would touch the uncovered model elements. The tester then creates the full test cases and another inspection session is held.

Coverage in Models

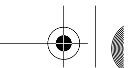
In the UML models we use, the model elements are the usual object-oriented concepts: classes, relationships, objects, and messages. A test case "covers" one of these elements if it uses that element as part of a test case. Of course, a single test case using a particular element probably does not exhaust all possible values of the attributes of that element. For example, using an object from a class to receive a single message does not test the other methods in the same class.

As we move deeper into the development life cycle, the detail of the model increases and the detail at which coverage matters increases as well. For a domain-analysis model, simply creating a single object from a class will be sufficient to consider that we have covered the class. Coverage for this level of model can be stated as a percentage of classes and relationships covered. At the design level, we would typically like to use every method in an interface before saying that a class is covered. Coverage for this level is more likely to be stated by counting all of the methods in the model rather than all of the classes.

The more abstract the classes, the higher the level of coverage that should be required. To omit a single abstract class from the coverage in testing overlooks the defects that could potentially be found in all of the concrete classes that eventually are derived from the abstract class. When testing at a concrete-class level, omitting a class during testing only overlooks the defects in that one class.

The higher the level of abstraction of the model, the higher the level of coverage that is required.

Reviews usually involve a discussion of the role of each piece of a model from a high level. The relationships between pieces are also explained in terms of the specified interfaces at the formal parameter level. The test cases created using this technique allow these same pieces and relationships to be examined at a much more concrete level that assigns specific values to the attributes of the objects. The test cases should be written at a level that is sufficiently specific to support tracing exact paths of execution through the logic of the algorithms, but not so specific that the code must be written first.



Should test cases be available to developers prior to the inspection session?

There has to be a balance between allowing developers to program to the tests and having the developers duplicate the effort of the testers by coming up with their own use scenarios. If the testers were going to develop all possible scenarios then giving those to the developers and sampling from them for model testing would be acceptable. Since the testers usually only create a small percentage of the possible scenarios, it is doubtful that they are duplicating the work of the developers who independently will (we hope) identify other scenarios. So, our general approach is to not let the developers have the scenarios prior to the inspection session.

Many object-oriented software development methods discuss using one or more diagrams within a model to evaluate the other diagrams. For example, a sequence diagram traces a path through the class diagram in which the messaging arrows in the sequence diagram are supposed to correspond to associations found in the class diagram. However, these development methods do not ensure a systematic coverage of the model. One step in guided inspection checks the internal consistency and completeness of the diagrams using the diagrams created during test execution.

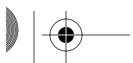
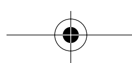
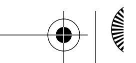
Should testers only use test cases for the current increment in an inspection session?

No. Running a test scenario from a previous increment as a regression check on the model is a useful idea. The regression scenarios should be chosen to include those that failed in the previous increment and those that cover areas most likely to have been changed to incorporate the functionality of the current increment.

Evaluation Criteria

We are essentially trying to answer three questions as we inspect the MUT:

- Is the model correct?
- Is the model a complete representation of the information?
- Is the model internally consistent and consistent with its basis model?

Correctness is a measure of the accuracy of the model. At the analysis level, it is the accuracy of the problem description. At the design level, it is how accurately the model represents the solution to the problem. At both levels, the model must also accurately use the notation. The degree of accuracy is judged with respect to a standard that is assumed to be infallible (referred to as "the oracle"), although it seldom is. The oracle often is a human expert whose personal knowledge is considered sufficient to be used as a standard. The human expert determines the expected results for each test case.

Testing determines that a model is correct with respect to a test case if the result of the execution is the result that was expected. (It is very important that each test case have an expected result explicitly stated before the test-case writer becomes biased by the results of the inspection.) The model is correct with respect to a set of test cases if every test case produces the expected result.

In the real world, we must assume that the oracle can be incorrect on occasion. We often separate the domain experts on a project into two teams who represent different perspectives or approaches within the company. One team constructs the model and at the same time, the second team develops the test cases. This check and balance doesn't guarantee correct evaluations, but it does raise the probability. The same is true for every test case. Any of them could specify an incorrect, expected result. The testers and developers must work together to determine when this is the case.

Completeness is a measure of the inclusiveness of the model. Are any necessary, or at least useful, elements missing from the model? Testing determines whether there are test cases that pose scenarios that the elements in the model can not represent. In an iterative incremental process, completeness is considered relative to how mature the current increment is expected to be. This criteria becomes more rigorous as the increment matures over successive iterations.

One factor directly affecting the effectiveness of the completeness criterion is the quality of the test coverage. The model is judged complete if the results of executing the test cases can be adequately represented using only the contents of the model. For example, a sequence diagram might be constructed to represent a scenario. All of the objects needed for the sequence diagram must come from classes in the class diagram or it will be judged incomplete. However, if only a few test cases are run, the fact that some classes are missing may escape detection. For the early models, this inspection is sufficiently high level that a coverage of 100% of all use cases is necessary.

Consistency is a measure of whether there are contradictions within the model or between the current model and the model upon which it is based. Testing identifies inconsistencies by finding different representations within the model for similar test cases. Inconsistencies may also be identified during the execution of a test case when the current MUT is compared to its basis model or when two diagrams in the same model are compared. In an incremental approach, consistency is judged locally until the current increment is inte-

grated with the larger system. The integration process must ensure that the new piece does not introduce inconsistencies into the integrated model.

Consistency checking can determine whether there are any contradictions or conflicts present either internal to a single diagram or between two diagrams. For example, a sequence diagram might require a relationship between two classes while the class diagram shows none. Inconsistencies will often initially appear as incorrect results in the context of one of the two diagrams and correct results in the other. Inconsistencies are identified by careful examination of the diagrams in a model during the simulated execution.

Additional criteria—defines a number of system attributes that the development team might wish to verify. For example, architectural models usually have performance goals to meet. The guided inspection test cases can be used as the scenarios for testing performance. Structural models used to compute performance can be applied to these scenarios, which are selected based on the use profile to estimate total performance and to identify potential bottlenecks.

If the architecture has an objective of facilitating change, test cases based on the change cases should be used to evaluate the degree of success in achieving this objective (see Testing Models for Additional Qualities on page 151).

Organization of the Guided Inspection Activity

Basic Roles

In the guided inspection activity there are three key roles that must be assumed by the available personnel.

1. **Domain expert**—They are the source of truth (or at least expected results). They define the expected system response for a specific input scenario. In many domains, the experienced developers are experts in their domain. They can provide a first line of validation. However, an additional, outside source of expertise is usually essential to the inspection process.
2. **Tester**—The people in this role conduct the analysis necessary to select effective test cases. Testers are often the creators of the basis model. When the scope of the inspection is at a system-wide level, the test case writers are often the system test team. They construct the input scenario specialized from the preconditions of a use case, the test actions as taken from the scenario, alternate paths, or exceptions sections of the use case, and the expected result as defined by the domain expert.
3. **Developer** —The creators of the MUT perform the role of "developer." They provide information that is not captured in the model. Except for those projects that generate code directly from a model, most developers leave many details out of the models, thus the necessary information

JMCH04.fm Page 121 Wednesday, December 27, 2000 10:17 AM

is only available from the developers. The development staff walks the inspectors through the model, tracing actions on diagrams, showing the relationships between diagrams, and providing the actual system response at a level appropriate to the current maturity of the development.

Individual Inspection

Guided inspection begins with a **desk check** like traditional inspection techniques. Each tester completes a checklist specific to the type of model being inspected. Certain incompleteness and inconsistency faults can easily be found during this task. This also turns out to be the easiest task to automate. A number of tools offer some limited amount of static checks, which are basically syntactic. We have had success in expanding that capability with the scripting languages in some of the design environments. We won't name names since the landscape changes almost daily, but check out this feature as part of your next tool purchase evaluation.

Preparing for the Inspection

Specifying the Inspection

When a guided inspection is planned, the scope and depth of the material to be inspected should be specified. The earliest models, such as requirements and domain models, may be inspected in their entirety at a single session. Later models will usually be too large to allow this. In Realistic Models on page 121, we talk about ways of creating modular diagrams that can be grouped into different-sized pieces. Having modular models facilitates limiting an inspection to the work of a single group or even to a specific class hierarchy.

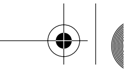
The **scope** of an inspection is defined by specifying a set of use cases, a set of packages, or abstract classes/interfaces. The scope determines starting points for scenarios, but other classes are pulled into scope as they are needed to support the scenarios.

The **depth** of the inspection is defined by specifying layers in aggregation hierarchies under which messages are not sent. The bottom layer classes simply return values with no indication of how the value was computed.

Realistic Models

It is usually not possible, or desirable to capture all of the details of an industrial-strength program in a few comprehensive diagrams in a single model. There will need to be multiple class diagrams, state diagrams, and of course, multitudes of sequence diagrams. In preparation for the guided inspection, the

JMCH04.fm Page 122 Wednesday, December 27, 2000 10:17 AM



developers should organize the model to facilitate the review by creating additional diagrams that link existing ones or by revising diagrams to conform to the scope of the inspection.

One basic technique that makes the model more understandable is to layer the diagrams. This results in more individual diagrams, but each diagram is sufficiently modular to fit within the scope of a specific inspection. The diagrams are easier to create because they follow a pattern.

Figure 4.6 illustrates one type of layering for class diagrams in which classes are grouped into packages and those packages may be enclosed in another package. Additionally, we often show all of the specializations from an abstract class as one diagram (see Figure 4.6) and all of the aggregation relationships for a class in another diagram.

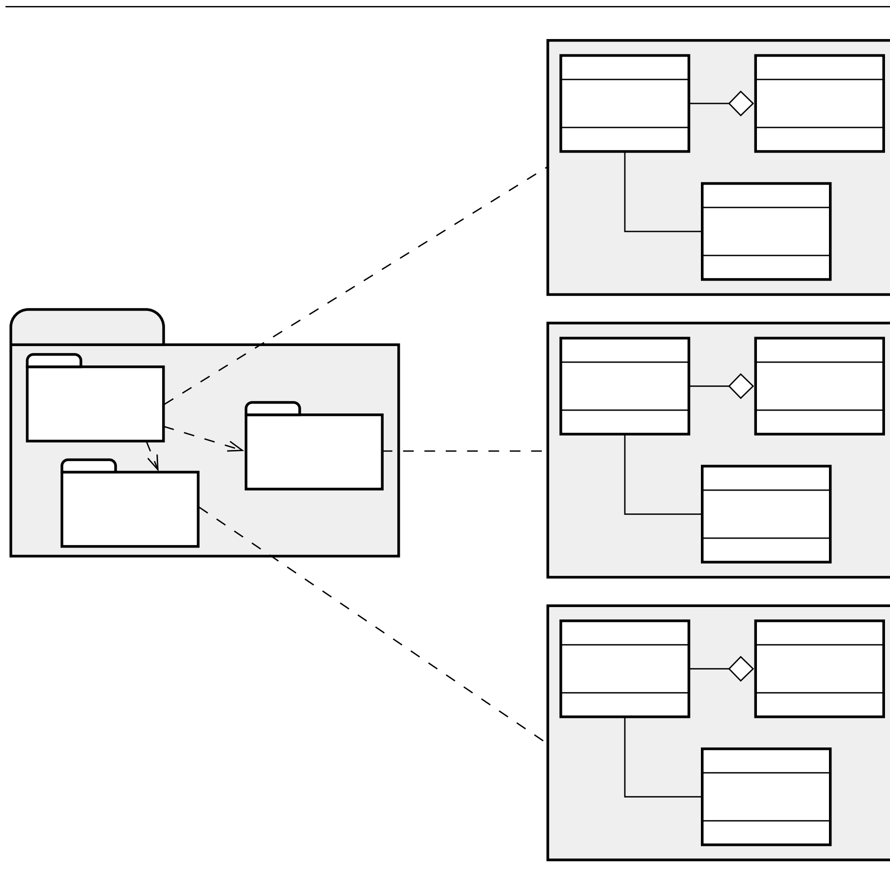
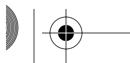
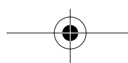
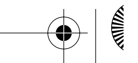


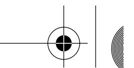
Figure 4.6 Class diagram layered into packages







JMCH04.fm Page 123 Wednesday, December 27, 2000 10:17 AM



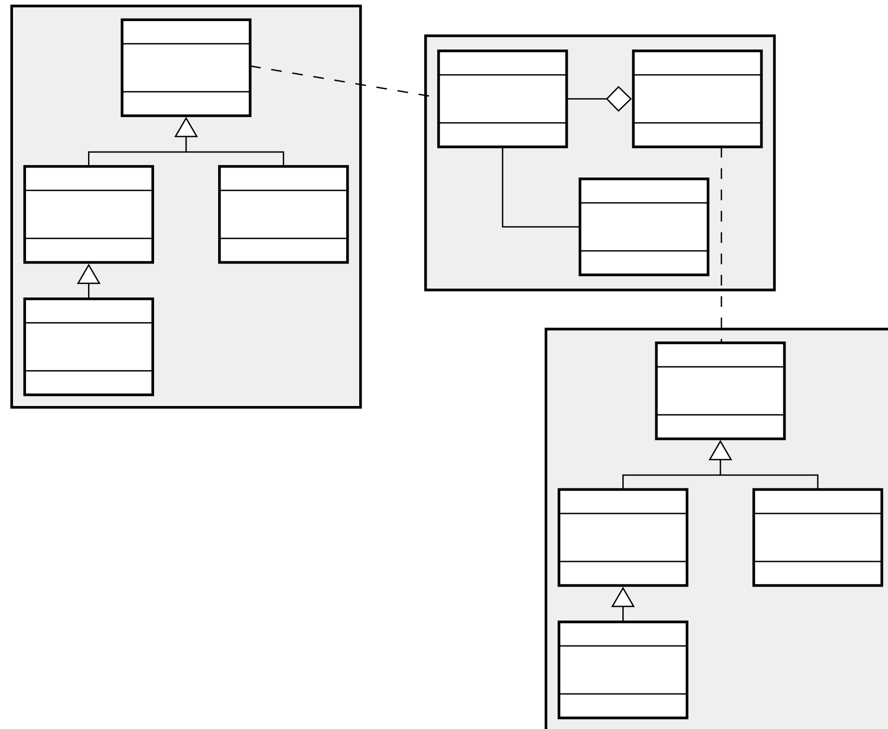


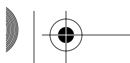
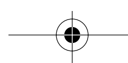
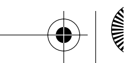
Figure 4.7 Separating relationships

Figure 4.9 illustrates a layering for sequence diagrams. At one level, the diagram terminates at an interface or abstract class. A sequence diagram is then constructed for each class that implements the interface or specializes the abstract class.

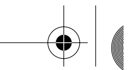
Figure 4.8 shows a technique for linking class diagrams. The work of one team uses the work of other teams. This can be shown by placing a class box from the other team on the edge of the team's diagram and showing the relationships between the classes. An inspection would be limited to the classes in the team's diagram. Messages to objects from the "boundary classes" would be not be traced further. The return value, if any, would simply be noted.

Selecting Test Cases for the Inspection

There are usually many possible test cases that can be developed from any specific use case. Traditional testing techniques use techniques such as equivalence classes and logical paths through the program as ways to select effective test cases. Test cases can be selected to ensure that specific types of coverage

JMCH04.fm Page 124 Wednesday, December 27, 2000 10:17 AM



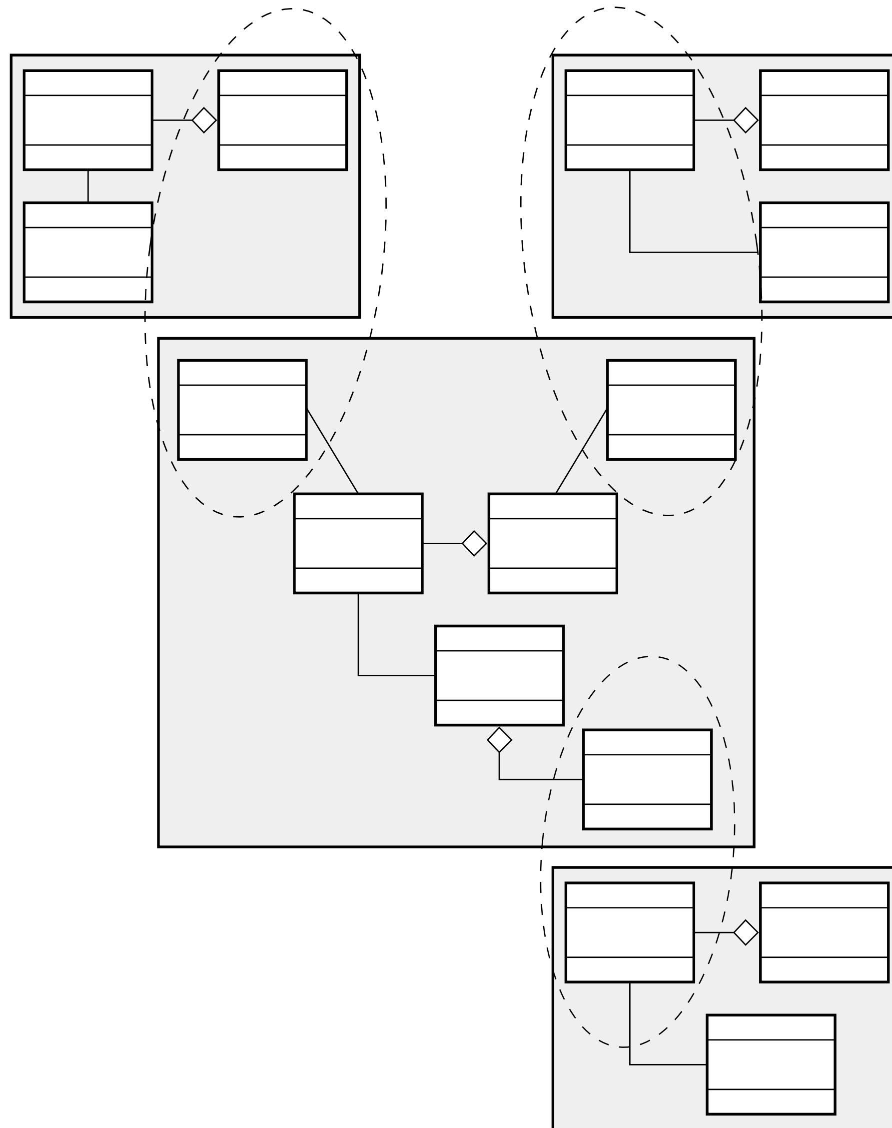
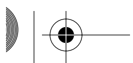
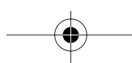
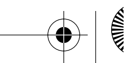
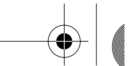


Figure 4.8 Links between class diagrams

are achieved or to find specific types of defects. We use *Orthogonal Defect Classification* to help select test cases that are most likely to identify defects by covering the different categories of system actions that trigger defects. We use a **use profile** to select test cases that give confidence in the reliability of the product by identifying which parts of the program are used the most.



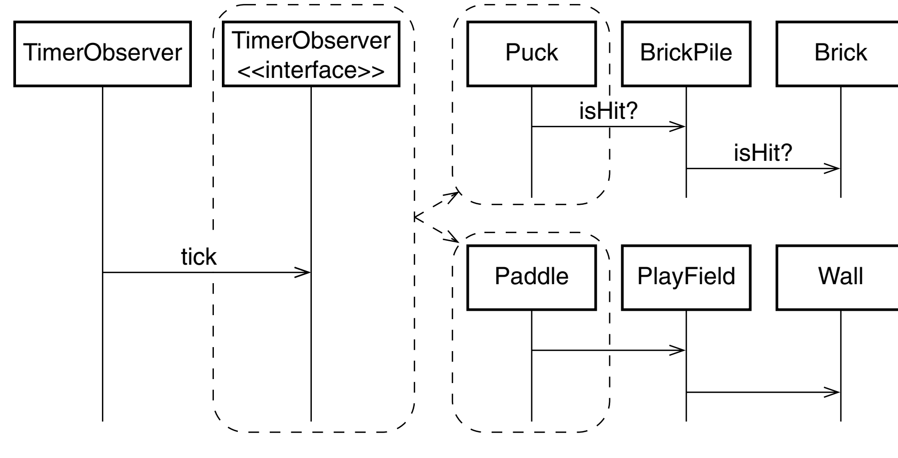


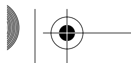
Figure 4.9 Sequence diagram per interface implementation

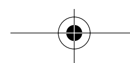
Orthogonal Defect Classification as a Test-Case Selector

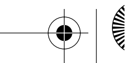
Orthogonal Defect Classification (ODC) [Chill92] is a scheme developed at IBM based on an analysis of a large amount of data. The activities that caused a defect to be detected are classified as "triggers." These are divided into groups based on when the triggers occurred, such as during reviews and inspections. Figure 4.10 is a list of attributes that trigger defects during reviews and inspections. The guided inspection technique uses several of these triggers as a guide to selecting test cases. We will delineate several of these triggers as we proceed, but we will address a few of these now.

1. Design conformance is addressed by comparing the basis model to the MUT as well as comparing the MUT to the requirements. This comparison is a direct result of the test-case execution.
2. Concurrency is a trigger that will be visible in the design model and scenarios can be generated that explicitly explore thread interactions. The UML activity diagram will be the primary source for symbolic execution.
3. Lateral compatibility is activated by the trace of scenarios between objects on sequence diagrams.

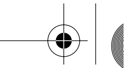
By structuring the guided inspection process so that as many of these triggers as possible are encountered, you ensure that the tests that guide the inspection are more likely to "trigger" as many failures as possible.







JMCH04.fm Page 126 Wednesday, December 27, 2000 10:17 AM



Design conformance—comparison of basis and current model.

Operational semantics—tracing the logic.

Concurrency—examining the synchronization between threads/processes.

Backward compatibility—comparison to previous products.

Lateral compatibility—comparison with interfaces that use this one.

Rare situation—examining unspecified system behavior.

Side effects—examining behavior outside the scope of the current product.

Document consistency/completeness—examine for consistency/completeness.

Language dependencies—examining for language-specific details.

Figure 4.10 ODC review and inspection triggers

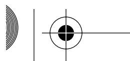
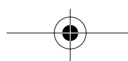
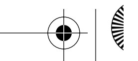
Use Profiles as a Test-Case Selector

A *use profile* [see See Use Profiles on page 130] for a system is an ordering of the individual use cases based on a combination of the frequency and criticality values for the individual use cases. The traditional operational profile used for procedural systems is based strictly on frequency-of-use information. Combining the frequency and criticality ratings to order the use cases provides a more meaningful criteria for ensuring quality. For example, we might paint a logo in the lower right-hand corner of each window. This would be a relatively frequent event, but should it fail, the system will still be able to provide important functionality to the user. Likewise, attaching to the local database server would happen very seldom but the success of that operation is critical to the success of numerous other functions. The number of test cases per use case is adjusted based on the position of the use case in the ranking.

Risk as a Test-Case Selector

Some testing methods use risk as the basis for determining how much to test. This is useful during development when we are actively searching for defects. It is not appropriate after development when we are trying to achieve some measure of reliability. At that time, the use-profile technique supports testing the application in the way that it will be used.

Our use-case template captures the information needed for each of the techniques so that they can be used throughout the complete life cycle. We use the frequency/criticality information instead of the risk information for guided inspection because we are trying to capture the same perspective as the testing of the system after development. For situations in which the inspection is only covering a portion of the design, using the risk information may be equally relevant.

JMCH04.fm Page 127 Wednesday, December 27, 2000 10:17 AM

Technique Summary—Creating Test Cases from Use Cases

A test case consists of a set of preconditions, a stimulus (inputs), and the expected response. A use case contains a series of scenarios: the normal case, extensions, and exceptional cases. Each scenario includes the action taken by an actor and the required response from the system that corresponds to the basic parts of a test case. To construct a test case from the scenario, each part of the scenario is made more specific by giving exact values to all attributes and objects. This requires coordination between the use-case diagram and the other diagrams. The "things" mentioned in the scenario should translate into some object or objects from the class diagram. Each of these objects should be in specific states defined in the state diagrams for those classes. The actions in the use case will correspond to messages to the objects.

Each scenario can result in multiple test cases by selecting different values (that is, states) for the objects used in the use case. The expected result part of the test case is derived from the scenario portion of the use case and the specific values provided in the input scenario. The following is a use-case scenario and the corresponding test case.

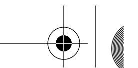
- Subsystem use case: A `movablePiece` receives a tick message. It must then check to determine whether it collided with a `stationaryPiece`.
- Test precondition: The puck is located within less than a tick of a brick and is headed for that brick.
- Test case: Input—The puck receives a tick message.
- Expected result: The puck has changed direction and the brick has changed its state from active to kaput.

Creating Test Cases

Test cases for a guided inspection are scenarios that should be represented in the MUT. Before the requirements model is verified, the scenarios come from a team of domain experts who are not producing the requirements. Later, we will see how this is done. For now we will focus on test cases that are based on the system requirements.

The use-case template that we use (see an abbreviated version in Figure 4.11) has three sources of scenarios. The *Use Scenario* is the "sunny-day" scenario that is most often the path taken. The *Alternative Paths* section may list several scenarios that differ from the use scenario in a variety of ways, but still represent valid executions. The *Exceptional Paths* section provides scenarios that result in error conditions.

JMCH04.fm Page 128 Wednesday, December 27, 2000 10:17 AM



Use Case # 1

Actor: Player

Use Scenario: The user selects the Play option from the menu. The system responds by starting a game.

Alternative Paths: If a game is already in progress, the selection is ignored.

Exceptional Cases: If the game cannot open the display, an error message is displayed and the program aborts.

Frequency: Low

Criticality: High

Risk: Medium

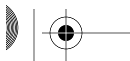
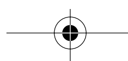
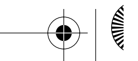
Figure 4.11 An example of a use case

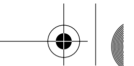
Completing Checklists

Prior to the interactive inspection session, the inspectors examine the models for certain syntactic information that can be evaluated just from the information contained in the model. This portion of the technique is not concerned with the content but only the form of the model. Figure 4.12 shows the checklist used during the design phase. The checklist is divided into two parts. One part addresses comparisons between the analysis model and the MUT. For example, the checklist reminds the inspector to check whether classes that have been deleted should have been deleted because of the differences between analysis and design information. The second part covers issues within the MUT. The checklist guides the inspector to consider whether the use of syntax correctly captures the information. For example, it guides the inspector to consider the navigability of the associations and whether they are correctly represented.

The Interactive Inspection Session

The testing portion of the guided inspection session is organized in one of two ways depending upon whether the model has been automated or not. If a prototype or other working model has been created, the session does not vary much from a typical code-testing session. The test cases provided by the testers are implemented, usually in some scripting language, and executed using the simulation facilities of the prototype of the model. These test cases must be more rigorously specified than the test cases that will be used in an interactive session with symbolic execution. The results of the execution are evaluated and the team determines whether the model passed the test or not.

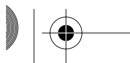
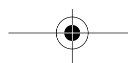
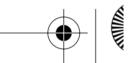






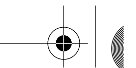
UML Detailed Design CheckList Question	*Yes*	*No*
Analysis-to-design model transformation issues. Are all classes in the analysis model that are not in the design model outside the scope of the application?		
Are all the states in the analysis-model statecharts also states in the statechart diagrams in the design model?		
Are the sequences of messages in all design-level sequence diagrams the same, even though additional messages may have been inserted between the analysis-level messages?		
Internal design-model issues. Are all associations shown with no navigation information truly bidirectional?		
Are all composition relationships shown as unidirectional?		
Is every sequence diagram a subset of some activity diagram?		
Does every message sent in an interaction diagram appear as a method in the public interface of the class of the receiving object?		
Does every message sent in an interaction diagram go to the logically appropriate object?		
Are the transitions out of a state diagram mutually exclusive?		
Do all state machines, except for perpetual objects, contain initial and final states?		
Are all public modifier methods represented as transitions on each state even if they only result in staying in the same state?		
Is there a sequence diagram for each postcondition clause of each method that corresponds to use cases that meet the frequency/criticality threshold?		
Are all messages shown correctly as synchronous or asynchronous?		
Do the number of forks and joins balance in every activity diagram?		

Figure 4.12 Design phase checklist.

If the model has not been prototyped, the testing session is an interactive session involving testers and developers. The developers cooperate to perform a symbolic execution that simulates the processing that will occur when actual code is available. That is, they walk the testers through the scenarios provided by the test cases.

JMCH04.fm Page 130 Wednesday, December 27, 2000 10:17 AM



Use Profiles

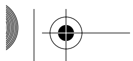
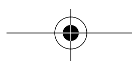
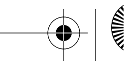
One technique for allocating testing resources determines which parts of the application will be utilized the most and then tests those parts the most. The principle here is "test the most used, most critical parts of the program over a wider range of inputs than the lesser used, least critical portions to ensure the greatest user satisfaction." A use profile is a ranking of the use cases based on the combined frequency/criticality values. This can be viewed as a double sort of the use cases based on the number of times that an end-user function (interface method) is used, or is anticipated to be used, in the actual operation of the program and the criticality of each of these uses. The criticality is a value assigned by the domain experts and recorded in each use case. The frequency information can be obtained in a couple of ways.

First, data can be collected from actual use perhaps during usability testing or during the actual operation if we will be testing a future version of the product. This results in a raw count profile. The count for each behavior is divided by the total number of invocations to produce a percentage. A second approach is to reason about the meanings and responsibilities of the system interface and then estimate the relative number of times each method will be used. The result is an ordering of the end-user methods rather than a precise frequency count. The estimated number of invocations for each behavior is divided by the total number of invocations to provide a percentage. The percentage computed for each use determines the percentage of the test suite that should be devoted to that use.

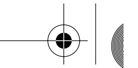
As an example, the `Exit` function for *Brickles* will be successfully completed exactly once per invocation of the program but the `NewGame` method may be used numerous times. It is conceivable that the `Help` function might not be used any at all during a use of the system. This results in a profile that indicates an ordering of `NewGame`, `Exit`, and `Help`. We can assign weights that reflect the relative frequency that we expect. If on average we would estimate that a player would play 10 games prior to `EXIT`ing the system, the weights would be 10, 1, 1. The `NewGame` function should be exercised in 82.5% (10 out of 12) of the test cases while the `Help` function and `Exit` should each constitute 8.5%.

The following additional roles are assigned to individuals in an interactive testing session. A person may take on the following roles simultaneously.

- *Moderator*—The Moderator controls the session and advances the execution through the scenario. The session is not intended to debug, which the developers will want to do, nor to expand the requirements, which the domain experts will want to do. The moderator keeps the session moving over the intended material.

JMCH04.fm Page 131 Wednesday, December 27, 2000 10:17 AM



- *Recorder*—This person, usually a tester, makes annotations on the reference models as the team agrees that a fault has been found. The recorder makes certain that these faults are taken into consideration in the latter parts of the scenario so that time is not wasted on redundant identification of the same fault. The recorder also maintains a list of issues that are not resolved during the testing session. These may not be faults. Information may need to come from a team in another part of the project or a team member who is absent during the inspection.
- *Drawer*—This person constructs a sequence diagram as a scenario is executed. A drawer concentrates on capturing all of the appropriate details such as returns from messages and state changes. They may also annotate the sequence diagram with information between the message arrow and the return arrow.

The guided inspection session can easily slip into an interactive design session. The participants, particularly the developers, will typically want to change the model during the testing session as problems are encountered. Resist this urge. This is the classic confusion between testing and debugging and diverts attention from other defects that are found. The recorder captures the faults found by the inspection so that they can be addressed later. This keeps attention focused on the search for faults and prevents a "rush to judgement" about the precise cause of the defect. If a significant number of problems are found, end the session and let the developers work on the model.

Testing Specific Types of Models

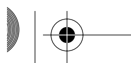
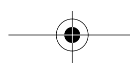
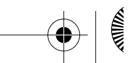
The basic guided inspection technique does not change from one development phase to another, but some characteristics of the model content and some aspects of the team do change.

- The level of detail in the model becomes greater as development proceeds.
- The amount of information also increases as development proceeds.
- The exact interpretation of the evaluation criteria can be made more specific for a specific model.
- The membership of the inspection team changes for different models.

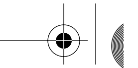
We will now discuss models at several points in the life cycle.

Requirements Model

The requirements for an application are summarized by creating a model of the *uses* of the system. The UML construct used for this model is the use case developed by Jacobson [JCJO92], which is discussed in Chapter 2. Figure 4.11 is an abbreviated version of the text format used for a use case. Figure 4.14 shows

JMCH04.fm Page 132 Wednesday, December 27, 2000 10:17 AM



Use Case # 1

Actor: Player

Use Scenario: The user selects the Play option from the menu. The system responds by starting a game.

Alternative Paths: If a game is already in progress, the selection is ignored.

Exceptional Cases: If the game cannot open the display, an error message is displayed and the program aborts.

Frequency: Low

Criticality: High

Risk: Medium

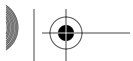
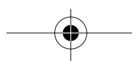
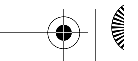
Figure 4.13 An example of a use case

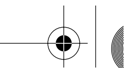
the UML use-case diagram for the *Brickles* example, and Figure 4.16 through Figure 4.21 show the use-case text descriptions. The use case diagram captures relationships between the use cases. Individual use-cases are broken into "sub-use cases" using the **uses** and **extends** relationships. Later, in Chapter 9, we will use these relationships to structure the system test cases. The text descriptions capture the majority of the information for each use case. While the relationships are used to structure tests, the text descriptions are used to provide most of the information for a test case.

Acceptance testing often finds faults that result from problems with the requirements. The typical problems include missing requirements (an incomplete requirements model), requirements that contradict each other (an inconsistent model), and scenarios in which the system does not behave as the client intended (an incorrect model). Many of these problems can be identified much earlier than the acceptance test phase using guided inspection.

The criteria for evaluating the models is interpreted specifically for the Requirements Model in Figure 4.15. Completeness is a typical requirements problem for which the iterative, incremental process model is a partial solution. Guided inspection can offer further help by requiring a detailed examination by an independent group of domain experts and product definition people. This examination will identify many missing requirements much earlier than it used to.

The detailed examination will also search for correctness faults. The act of writing the test cases for the guided inspection will identify many requirements that are not sufficiently precise to allow a test case to be written. Running the test cases will provide an opportunity for the independent group to identify discrepancies between the expected results in the test cases and the actual content of the requirements model.



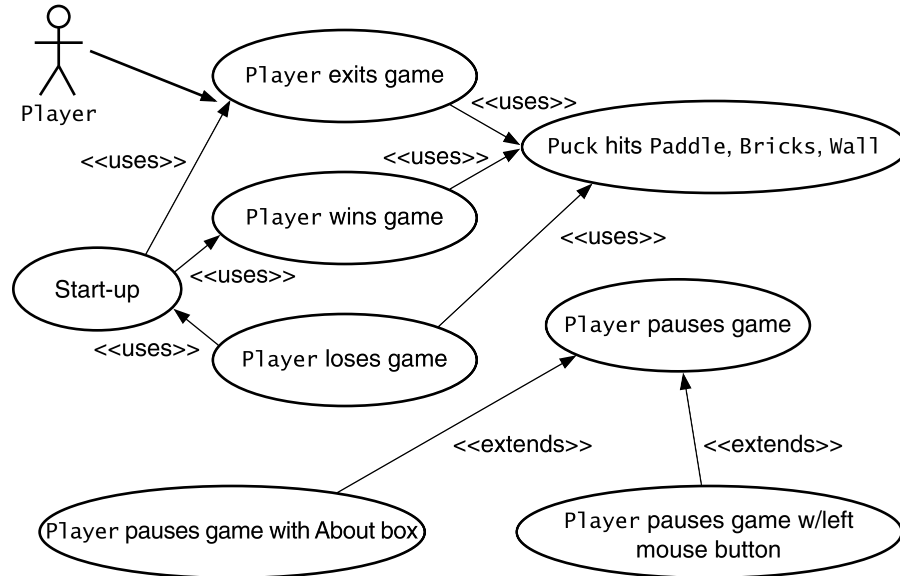
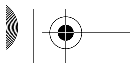
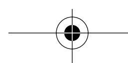
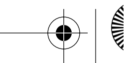


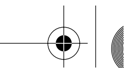
Figure 4.14 *Brickles* use-case model.

Criteria	Interpretation for Requirements
Completeness	The use cases represent all of the functionality needed for a satisfactory product. No use case is included that is not required functionality.
Correctness	Each use case accurately represents a requirement.
Consistency	Any system functionality is specified in the same manner everywhere it is described.

Figure 4.15 Criteria for requirements inspection

The larger the system, the more problem there is with the consistency of the requirements. In addition to contradictions, there is often the need to identify places where one use case supersedes another. For example, one use case calls for an action to happen at least within ten seconds while another expects the same action to occur within seven seconds. The use of the end-to-end scenarios that trace a complete action will help locate these inconsistencies.



Use Case #1

Actor: Player

Use Scenario: The player starts the Brickles executable, plays the game and then selects the Exit option on the File menu.

Alternative Paths:

Exceptional Cases:

Frequency: Low

Criticality: High

Risk: Low

Figure 4.16 An example of use case #1

Use Case #2

Actor: Player

Use Scenario: The player starts the Brickles executable, plays the game, breaks all of the bricks, and wins.

Alternative Paths:

Exceptional Cases:

Frequency: Medium

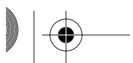
Criticality: High

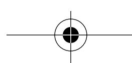
Risk: Medium

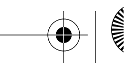
Figure 4.17 An example of use case #2

One feature of the requirements model that affects how the inspection is organized is that there is no UML model on which the requirements are based. So comparisons to the basis model refer to documents produced by marketing, system engineering, or client organizations. Since this is a notorious source of defects, we will expend extra effort in verifying the requirements model.

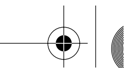
The roles for this inspection are assigned as shown in Figure 4.22. You will want to adapt these to your situation. The domain-expert role provides the "correct" answers for test cases. In this case that means agreeing or disagreeing that a use case adequately represents the required functionality. Using the system testers in the tester role provides the system testers with an early look at the source of information for the system test cases and an opportunity to have







JMCH04.fm Page 135 Wednesday, December 27, 2000 10:17 AM



Use Case #3

Actor: Player

Use Scenario: The player starts the Brickles executable, plays the game, loses all of the pucks, and loses the game.

Alternative Paths:

Exceptional Cases:

Frequency: Medium

Criticality: High

Risk: Medium

Figure 4.18 An example of use case #3

Use Case #4

Actor: Player

Use Scenario: The puck bounces against the paddle, breaks some bricks, hits a wall.

Alternative Paths:

Exceptional Cases:

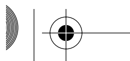
Frequency: High

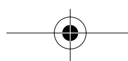
Criticality: High

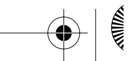
Risk: Medium

Figure 4.19 An example of use case #4

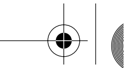
input into improving the use cases. We also use a second group of domain experts and product definition people to work with the system testers. This provides a source of scenarios that is independent of the people who wrote the requirements. Some organizations will have the use cases written by developers rather than a separate organization of system engineers, and these developers will be the ones to execute test cases.







JMCH04.fm Page 136 Wednesday, December 27, 2000 10:17 AM



Use Case #5

Actor: Player

Use Scenario: The player pauses the game by holding down the left mouse button.

Alternative Paths:

Exceptional Cases:

Frequency: Low

Criticality: Low

Risk: Medium

Figure 4.20 A example of use case #5

Use Case #6

Actor: Player

Use Scenario: The player pauses the game by selecting the About menu entry.

Alternative Paths:

Exceptional Cases:

Frequency: Low

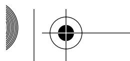
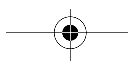
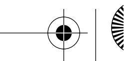
Criticality: Low

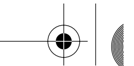
Risk: Medium

Figure 4.21 An example of use case #6

Domain Expert	Domain expert, marketing representative, client, product-definition personnel
Tester	System tester, domain expert
Developer	System engineer

Figure 4.22 Roles in requirements inspection



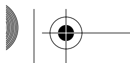
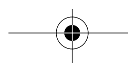
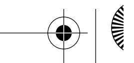
Tip When dividing the domain experts into two groups, don't divide based on ideology. That just precipitates theoretical debates. Divide the experts so that each team has representation from as many "camps" as possible.

The basic outline of "testing" the requirements model is given in the following list along with an example using *Brickles*. The example is given in italics.

1. Develop the ranking of use cases by computing the combined frequency and criticality information for the use cases. *Figure 4.23 gives the ranking for* Brickles*.*
2. Determine the total number of test cases that can be constructed given the amount of resources available. It should be possible to estimate this number from historical data. *We will assume we have time for 15 test cases.*
3. Ration the tests based on the ranking. *Figure 4.23 gives the numbers for* Brickles*. Note that only 14 of the 15 are assigned since it is impossible to evenly split the number of tests. The 15 test would be allocated to the category showing the most failures in the initial round of testing.*
4. Write scenarios based only on the knowledge of those in the domain expert's role. The number of scenarios is determined by the values computed in Step 3. *The player starts the game, moves the paddle, and has broken several bricks by the time he loses the puck. The system responds by providing a new puck.*

Use	Frequency	Criticality	Combined Value	Rank	Number of Test Cases
Start *Brickles*	Medium	High	High	1	3
Pause *Brickles*	Low	Low	Low	3	1
Stop *Brickles*	Medium	Low	Medium	2	2
Break brick	High	High	High	1	3
Wins	Medium	High	High	1	3
Loses	Medium	Low	Medium	2	2

Figure 4.23 *Brickles* use cases

JMCH04.fm Page 138 Wednesday, December 27, 2000 10:17 AM

5. In a meeting of the producers of the requirements and the test-scenario writers, the writer presents each scenario and the requirements modelers identify the use case that contains the test scenario as either a main scenario, extension, exception, or alternative path that represents the scenario. If no match is found, it is listed as an incompleteness defect. If the scenario could be represented by two or more use cases (on the same level of abstraction), an inconsistency defect has occurred. In both of these cases, the first question asked is whether there is an incorrectness defect in the statement of a use case that, if corrected, would handle the scenario accurately. *In the scenario provided in Step 4 there is no mention of the limited number of pucks. The system may not be able to provide a puck if the supply is exhausted. The requirement should be explicit about a fixed number of pucks.*

Much of this effort will be reused in the testing of other models. Both the ranking of use cases and construction of test cases will produce reusable assets. The requirements model will serve as the basis for testing several other models, and therefore, these test cases can be reused.

Analysis Models

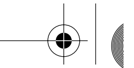
We will be concerned with two types of analysis models: domain- and application-analysis models. The following two types are analysis models that model existing knowledge. One models the knowledge in a domain while the other models knowledge about the product.

Domain-Analysis Model

The domain-analysis model represents information about a domain of knowledge that pertains to the application about to be constructed. As such, it is derived from the literature and knowledge about the domain as opposed to another UML model. Although many projects are satisfied with creating a domain model that is only a simple class diagram, most domains encompass standard algorithms and many refer to states that are characteristic of the concepts being represented. Figure 4.24 shows the interpretation of the evaluation criteria for a domain model.

The domain model is a representation of the knowledge in a domain as seen through the eyes of a set of domain experts. As is to be expected, there can be differences of opinion between experts. For this reason, we have found it useful to divide the available set of experts into two groups. One group, the larger, creates the domain model while the second group serves as the testers of that model. In Figure 4.25, group one is referred to as the developers and group two is referred to as both testers and domain experts. This check and balance between the groups provides a thorough examination of the model.

JMCH04.fm Page 139 Wednesday, December 27, 2000 10:17 AM



Criteria	Interpretation for Domain Modeling
Completeness	The concepts are sufficient to cover the scope of the content specified. Sufficient detail is given to describe concepts to the required depth.
Correctness	The descriptions of domain concepts are accurate; the algorithms will produce the expected results.
Consistency	Model elements should be consistent with the company's definitions and meanings.

Figure 4.24 Criteria for domain model inspection

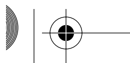
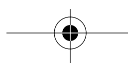
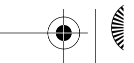
Role in Model Inspection	Role in Project
Domain Expert	Domain expert
Tester	System tester, domain expert
Developer	Domain expert

Figure 4.25 Roles in domain model inspection

Figure 4.26 relates portions of the class diagram from the domain models for *Brickles* to its application-analysis model. Note that there are two domains represented, *Interactive Graphics* and *Games*. The test cases for this model will come from the second group of domain experts. They consider how these concepts are used in the typical applications in which they have had experience. The test cases will be written by a team composed of a system tester who knows how to write test cases, and the second group of domain experts. A test case only states details down to the level of the domain concepts. Any actions are domain algorithms.

A test case for the *Interactive Graphics* domain model would look like the following:

> *Assume that a canvas has been created and asked to display a shape. How will the canvas know where to locate the shape? It is expected that a* `mouseEvent` *would provide the coordinates to which a system user points.*

JMCH04.fm Page 140 Wednesday, December 27, 2000 10:17 AM

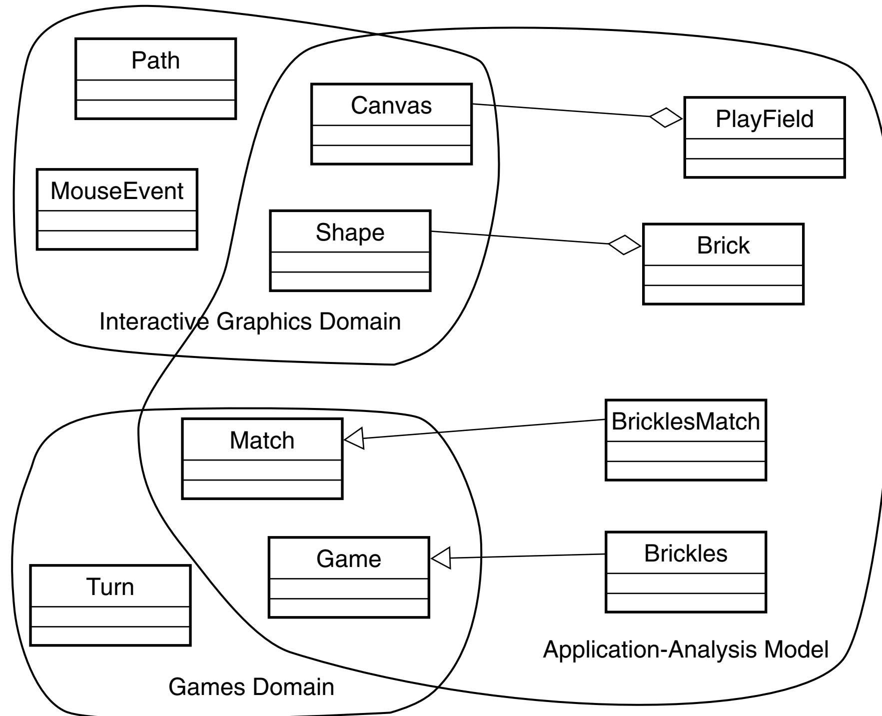


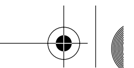
Figure 4.26 Mapping domain models onto application-analysis models

Application-Analysis Model

There will usually be multiple domain models for a large project. All of these contribute to the single application-analysis model. Many parts of each domain model will be thrown away because they are outside the scope of this particular project. Some pieces of domain models will be merged to provide a single element in the application model. This makes judging completeness during the inspection more difficult since there is not a direct mapping from one model to another.

An analysis model can be *too* complete. That is, it can contain design information that the project team has erroneously made part of the requirements. This leads to an overly constrained design that may not be as flexible as possible. As the inspection team measures the test coverage of the model, they examine pieces that are not covered to determine whether they should be removed from the model.

Figure 2.13 shows the class diagram for the application-analysis model for *Brickles*.



Criteria	Interpretation for the Application-Analysis Model
Completeness	The ideas expressed in each use case can be represented by the concepts and algorithms in the model. No design information is included in the model.
Correctness	Experts agree with the attributes and behaviors assigned to each concept; on the steps in each algorithm; major states for each conceptual entity.
Consistency	Where there are multiple ways to represent a concept of action, those ways are equivalent.

Figure 4.27 Criteria for application-analysis model inspection

Role in Inspection	Role in Project
Domain Expert	Domain expert; system engineer
Tester	System tester
Developer	Application developer

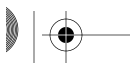
Figure 4.28 Roles in application-analysis model inspection

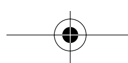
A test case for the application-analysis model would look like the following:

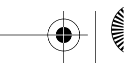
> *Assume that a match has been started and the playfield has been constructed. How will a paddle prevent a puck from striking the floor boundary? It is expected that the paddle will move into the trajectory of the puck and collide with it. The collision will cause the puck to change direction by reflecting off the paddle at a 90-degree angle from the point of impact.*

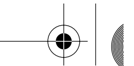
Design Models

There are three levels of design in an object-oriented project: architectural, mechanistic, and detailed. We will focus on two basic design models that encompass those three levels: the architectural design model and the detailed class design model. The architectural model provides the basic structure of the application by defining how a set of interfaces are related. It also specifies the exact content of each interface. The detailed class model provides the precise semantics of each class and identifies the architectural interface to which the class corresponds.









Architectural Model

The architecture model is the skeleton of the entire application. It is arguably the most important model for the application so we will go into a fair amount of detail in this section. This is the model in which the non-functional requirements are blended with the functional requirements. This provides the opportunity to use the scenarios as a basis for modeling performance and other important architectural constraints.

Criteria	Interpretation for the Architectural Design Model
Completeness	A sufficient set of interfaces are defined to provide all of the services needed for the application's functionality. The relationship between the interfaces allows for the flow of control and data necessary to realize all of the uses described in the use-case diagram.
Correctness	The architecture satisfies its constraints; uses the appropriate architectural patterns; represents the interactions between the interfaces.
Consistency	Each use of the system can be handled only in one set of interfaces.

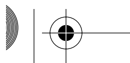
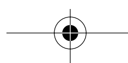
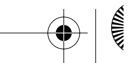
Figure 4.29 Criteria for the architectural design model inspection

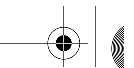
Role in Inspection	Role in Project
Domain Expert	Domain expert; system engineer
Tester	System tester
Developer	Architect

Figure 4.30 Roles in the architectural design model inspection

An architectural design test case would look like the following:

> *Assume that the* `BricklesDoc` *and* `BricklesView` *objects have been constructed. A tick message is sent to every* `MovablePiece`*. How does the* `BricklesView` *receive the information necessary to update the bitmaps on the screen? It is expected that the* `BricklesDoc` *object will calculate the new position of each bitmap before it notifies the* `BricklesView` *that a change has occurred. The* `BricklesView` *object will call methods on the* `BricklesDoc` *object to obtain all of the information that it needs to update the display.*



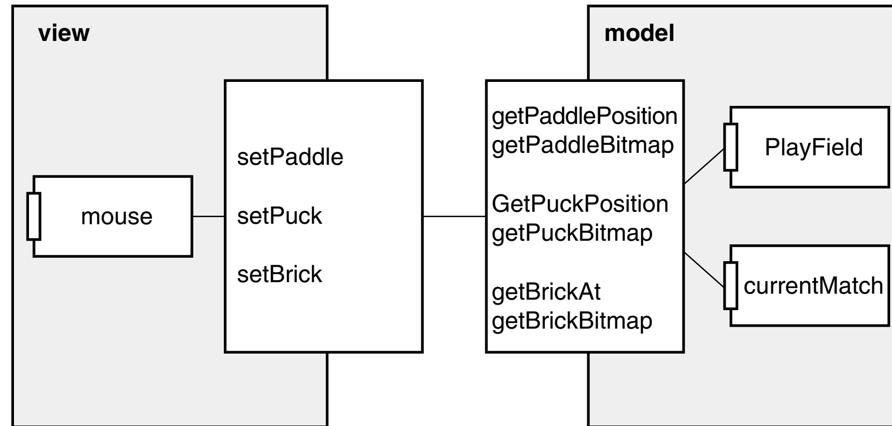


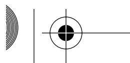
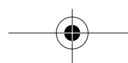
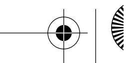
Figure 4.31 An architectural model for *Brickles*

We will use the architecture of our game framework to illustrate the variety of techniques considered in this section. We first implemented the framework in C++ using the Microsoft Foundation Classes (MFC), which imposes an architecture known as Document/View and is a variant of the canonical Model/View/Controller (MVC) architecture [Gold89]. The framework was then implemented in Java using the java.awt package, which supports a slightly different form of MVC. In each of these efforts the user-interface classes present the state of the game to the user. To achieve this, the user interface has to maintain some state itself. A typical fault for these systems would be for the state in the user interface to be different from the state maintained in the classes implementing the model.

A software architecture is the basic structure that defines the system in terms of computational components and interactions among those components [ShGa96]. We will use the terms **component** and **connector** to describe the pieces of an architecture. In the UML notation, components of the architecture are represented as classes with interfaces. If the connectors between components do not have any explicit behavior, they can be represented by simple relationships between classes. If the connectors do have state and/or meaningful behavior then they are represented by objects.

Representations for Architectures

There are three types of information that are widely used to represent an architecture: relationships, states, and algorithms. The basic UML modeling language has the advantage that it can be used for all three design models as well as the

JMCH04.fm Page 144 Wednesday, December 27, 2000 10:17 AM

analysis models. Using the same notation for all three levels of models eliminates the need to learn multiple notations. Notations such as the UML are sufficiently simple in that no special tools are required, although for large models, tool support quickly becomes a necessity. Tools such as Rational's Rose perform a variety of consistency checks on the static-relationship model. With this type of representation, the test cases are manually executed using the technique discussed previously. However, UML does not have specific syntax for describing architectures so the concept/token mapping between the architecture and UML symbols is ad hoc.

Tools such as ObjectTime [ObjectTime94] and BetterState [BetterState00] provide facilities for "animating" design diagrams and provide automatic checking of some aspects of the model. In particular, they support a simulation mechanism that can be used to execute scenarios. The diagrams are annotated with detailed scenario information as well as special simulation information. The developer can "play" scenarios and watch for a variety of faults to be revealed. This approach makes the creation of new scenarios (and test cases) easier by providing a generalized template. One advantage of this approach is the combination of easy model creation with powerful simulation facilities. Usually however, these tools have a limited set of diagram types. BetterState for example focuses on building a state model as the specification for the system. This leaves incomplete those static portions of the system that do not affect the state. The obvious benefit is that the scenarios are executed automatically. This makes it easier to run a wide range of scenarios at the price of more time needed to create the model initially. This approach is best suited to small, reactive systems or those systems whose requirements change very little during development.

Often these tools will assist in finding some types of faults as the model is entered into the tool. Consistency checks will prevent certain types of connections from being established. Scenarios are represented in some appropriate format such as a sequential file of input values that are read at appropriate times. The actions of the simulation are often represented by events. Event handlers can be used to "catch" and "generate" events at a high level without the need to write detailed algorithms. This level of execution is sufficient for verifying that required interfaces are provided. It obviously is not sufficient for determining whether the functionality is correctly implemented.

Finally, architectural-description languages provide the capability to represent a system at a high level of abstraction. Languages such as Rapide [Luckham95], which has developed at Stanford University, allow the modelers to be as specific as they would like to be. The flow of computation is modeled by events that flow between components. One advantage of this approach is the control that this approach provides to the modeler. The language is sufficiently descriptive to support any level of detail that the modeler wishes to use, unlike the tools previously discussed, which have a fixed level of representation. The disadvantage is that these models are programs with all of the problems associated with that level of detail.

JMCH04.fm Page 145 Wednesday, December 27, 2000 10:17 AM

When the model and test cases are represented in a programming language, the test execution can be performed automatically. The representation language may be a general purpose programming language used to implement a high-level prototype or a special purpose architectural-description language such as Rapide, which is used to build a standard model. The level of detail represented in the prototype will determine how specific the testing can be.

Testing the Architecture

The Software Architecture Testing (SAT) [McGregor96] technique is a special type of guided inspection that requires the following usual steps in testing any product: (1) test cases are constructed; (2) the tests are conducted on the product; and (3) the results of the test are evaluated for correctness. This technique is a "testing" technique because there are very specific test cases, and there is the concept of an execution even if the execution is sometimes manual. The team that is assigned to drive this activity is divided as shown in Figure 4.30. We will provide additional detail on each of the steps.

Constructing Test Cases

Test cases for the architecture are constructed from the use cases as described previously. Each use case describes a family of scenarios that specifies the different types of results that can occur during a specific use of the system. The test cases for the architecture are defined at a higher level than the more detailed design models.

The test cases are essentially defined at a level that exercises the interfaces between subsystems. For example, for the game framework, the essential interface is between a model and a view. The model is divided among the `Puck`, `Paddle` and `BrickPile` classes. The view is concentrated in the `BricklesView` class.

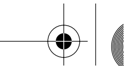
The Model/View architecture calls for most of the interaction to be from the view but with the model notifying the view when a change has occurred to the model. Since *Brickles* requires animation, we modified the architecture so that when the `BricklesView` object is created it is sent a series of messages that provide it with handles to the pieces of the model.

The basic architectural model is given in Figure 4.33. With the analysis out of the way, the test cases can be selected. The two basic operations are (1) setup of the system and (2) repainting the screen after a move has occurred. Unlike many systems built on Model/View, there is no need to consider the ability to add additional views. We could define a test case for each operation; however, a single **grand tour**[1] case can be defined in this case. Usually grand tours are too large and give little information if they fail, but in this case the second operation cannot be realized without the first so it is a natural conjunction.

---

1. A grand tour is a test case that combines a number of separate test cases into one run.

JMCH04.fm Page 146 Wednesday, December 27, 2000 10:17 AM



Test Execution

The tests are executed as described for each specific type of representation. We have used the UML notation so this will be an interactive session.

We execute the test case by constructing a message-sequence diagram. The diagram reflects preconditions for a test case. The `BricklesView` object is created followed by a `BricklesGame` object. As the `BricklesGame` object is created, it creates a `PlayField` object that turn creates `Puck`, `Paddle`, and `BrickPile` objects. The messages across the architectural boundaries are shown in bold italics in Figure 4.32.

Verification of Results

Usually for architectures, this step is fairly simple, even though for the detailed functionality of the final application it can be very difficult. When the output from the test is in the form of diagrams, the resulting diagrams must be verified after each test execution by domain experts. When the output is the result of an execution, the test results can be verified by having those domain experts construct event sequences that would be produced by an architecture that performs correctly. The interpretation of the evaluation criteria is given in Figure 4.29.

An Additional Example

The architecture of *Brickles* is obviously very simple so let's consider the typical three-layer architecture. Although the diagram in Figure 4.33 is greatly simplified, we can consider the types of test cases that would be effective. The client is intended to interact with a user, do computations needed to format presentations, and interact with the business model residing on the application server. The application server is intended to be the primary computational engine, and it also handles interactions with the client and database components. Finally, the database component provides persistence for the business objects from the application server.

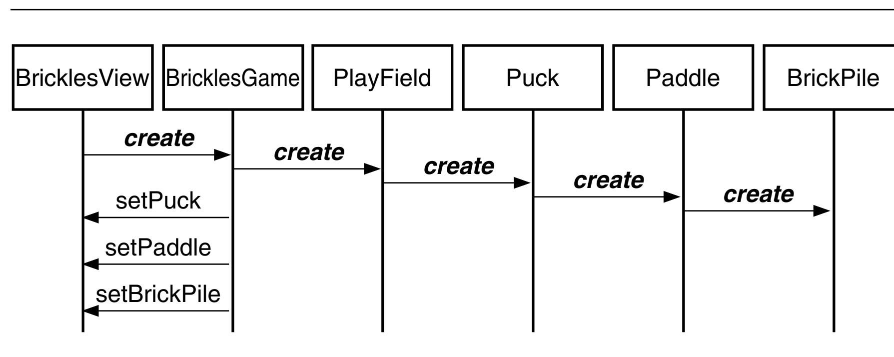
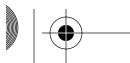
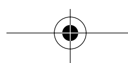
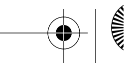


Figure 4.32 Test case execution

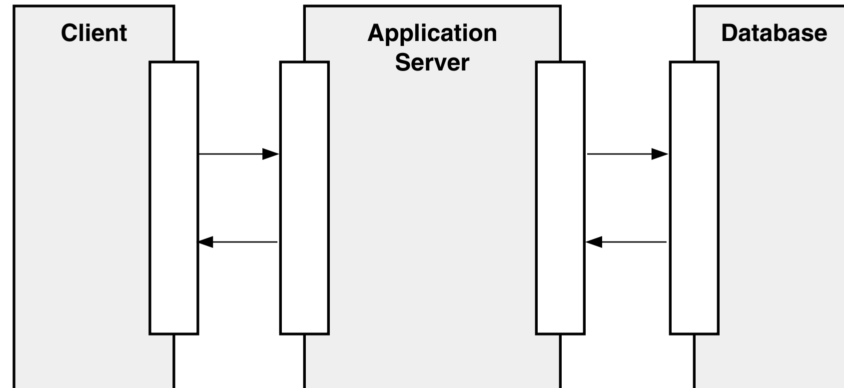


Figure 4.33 Three-tiered architecture

The most important scenarios for this type of architecture include multiple client/single server and multiple client/multiple server scenarios. The discussion in the next section provides a technique for structuring these tests so that they are repeatable and representative. Useful coverage of this architecture includes exercising various combinations of threads. Since these systems are usually distributed, we will defer further discussion until Chapter 8.

Evaluating Performance and Scalability

The architecture of a system should be evaluated beyond correctness, completeness, and consistency. Most architectures will have a specified set of quality attributes and these should also be evaluated. A system that presents animation, as does *Brickles*, must meet performance goals. The scenarios used as test cases for the basic inspection can also be used to analyze the expected performance for the architecture. The SAAM [Kazman94] approach uses a free-form analysis technique for analyzing performance. Software Architecture Testing (SAT) [McGregor96] uses the testing perspective to ensure that the important features of the architecture are investigated.

The test cases are symbolically executed and the message-sequence diagrams can be analyzed from a performance perspective. For the analysis, each connection between components in the architecture is assigned a "cost" that reflects the type of communication used by the connection. The number of messages in each scenario gives an indication of the relative performance although by itself the technique gives an order of magnitude to quantified value rather than a specific quantity. A more exact value can be computed by the following string:

$$\text{time to compute} = n_1c_1 + n_2c_2 + \ldots + n_mc_m$$

JMCH04.fm Page 148 Wednesday, December 27, 2000 10:17 AM

in which the c's are the connection types and the n's represent the number of each connection type in the scenario. If a use profile (see Use Profiles on page 130) is used to select a representative set of test cases, an accurate approximation to a typical user session can be computed. Worst and best case approximations can also be constructed.

Design alternatives can be evaluated by comparing the message-sequence diagrams and the relative quantities of messages. By using the same set of test cases, selected based on the use profile, a fair and realistic comparison can be made as to how the system will perform if constructed using each alternative.

The performance of distributed systems can also be analyzed in this manner by annotating those messages that will be interprocess and interprocessor. The test case approach using an use profile produces a representative performance measure.

The sequence diagrams can also be used to evaluate the scalability of the architecture. The following use profile indicates several types of users and different frequencies of operations in each:

$$userType = p_1s_1,\ p_2s_2,\ \ldots,\ p_ns_n$$
$$useProfile = q_1ut_1,\ q_2ut_2,\ \ldots,\ q_mut_m$$

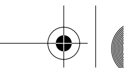
in which the p's and q's are the probability that a particular scenario and user type, respectively, will be selected.

A scalability test case is a hypothetical mix of actors that is different from the current use profile, that is, a set of values for the q's in the `useProfile` equation. Usually, the different types of users will remain constant, but the relative number changes. The computation given previously is used for each scenario and for each user type. Then the number of each user type is used to aggregate further. The resulting values can identify the intensity of use for specific messages.

Detailed Class Design Model

The detailed class design model populates the architectural model with classes that will implement the interfaces defined in the architecture. This model typically includes a set of class diagrams, the OCL pre- and postconditions for every method of every class, activity diagrams of significant algorithms, and state diagrams for each class. The detailed design model for *Brickles* is shown in Figure 2.18, and additional detail is shown in Figure 2.15 for one specific class.

The model evaluation criteria are specialized in Figure 4.34. The focus is on compliance with the architecture. This reinforces the idea that the architecture is the keystone of the product. This is also the place where components will be reused and inserted into the system. The specification of the component should be included in the execution trace to ensure there is no need for an adapter between the component and the application.



Criteria	Interpretation for Class Design Model
Completeness	Classes are defined for each interface in the architecture. The preconditions for each method specify sufficient information so that the user can safely use the method. The postconditions for a method show error conditions as well as the normally expected result.
Correctness	Each class accurately implements the semantics of an interface. For those classes that correspond to interfaces in the architecture, the class's specification must correspond to the interface specified by the architecture.
Consistency	The behaviors in the interface of each class provides either a single way to accomplish a task or, if there are multiple ways, they provide the same behavior but with different preconditions.

Figure 4.34 Criteria for the class design model inspection

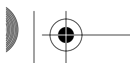
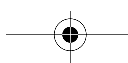
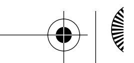
The roles are assigned in Figure 4.35. Notice that the architects have a role in testing the class diagram. The architects' responsibility to a project is to "enforce" the architecture. That is, the architect makes certain that developers do not violate the constraints imposed by the architecture. By selecting test cases and evaluating the results, the architects can gain detailed knowledge about the developers' implementation.

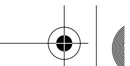
A detailed class design test case would look like this:

> *Assume that a puck is moving to the left and up, but will hit the left wall before hitting a brick. How will the puck's direction and speed be changed when it hits the wall? It is expected that when the puck is found against the left wall, the wall will create a* `Collision` *object that will be passed to the puck. The puck will modify its velocity and begin to move to the right and up. It will be moving at the same speed.*

Role in Inspection	Role in Project
Domain Expert	Domain expert; system engineer
Tester	Application developer; architect
Developer	Application developer

Figure 4.35 Roles in class design model inspection



The test cases at this level are very much like the final system test cases. There is so much detail available at this level that the testers have to be careful to record all the model elements that are touched by test cases. Figure 4.34 shows the diagram elements that must be coordinated during the guided inspection session. As the test progresses, the executors select methods that will be invoked, the state model of the receiver is checked to be certain that the target object can receive the message. The messages are then added to the sequence diagram and the state models are updated to reflect changes in state. When a state in a diagram is shaded, there is additional detail to the state but that information is not needed to evaluate the current test. Sequence diagrams will also have "dead-end" objects in which the testers will not attempt to examine the logic beyond that object.

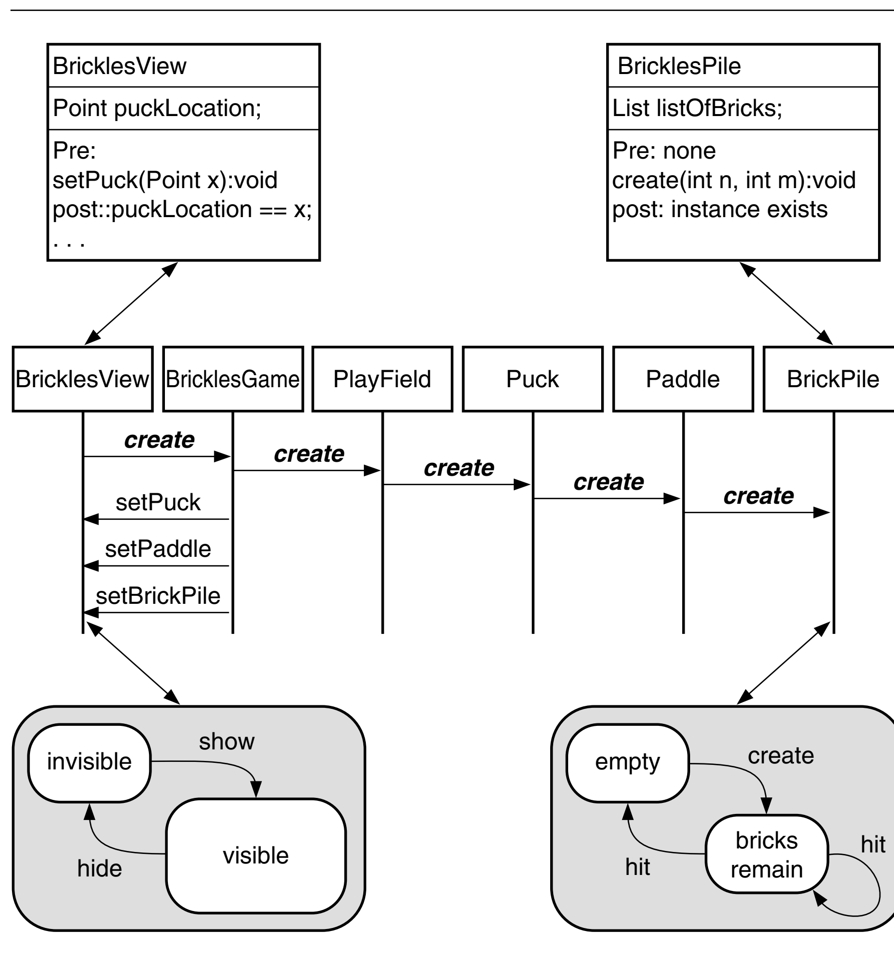
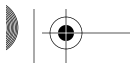
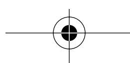
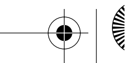


Figure 4.36 A test environment

JMCH04.fm Page 151 Wednesday, December 27, 2000 10:17 AM

This is the last step prior to implementation and code-based testing. Developers doing buddy testing of each other's code will benefit from coming back to the test cases created at this level of testing and translating these into class-level code tests.

Testing Again

We are assuming that you are using an iterative development process as we do. That means that these tests must be repeatable. We have tried to accomplish this by writing down formal test cases as opposed to simply thinking up scenarios during the inspection session.

On the second and succeeding iterations, we usually choose to reapply all those tests that were failed the last time and some of those that were passed. Tests may be added to cover the new features added. If any problems were discovered after the inspection was conducted, tests should be added to check for that problem as well.

Tip

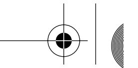
> Use guided inspection to transfer knowledge about the model under test. On a recent project, when the developer responsible for a specific piece of the design was leaving, we used a series of inspection sessions to bring other developers up to speed on their individual piece. A presentation by the developer, as opposed to an inspection, would have addressed the design in the way he knew it best, not in the way the other developers were viewing it.

Testing Models for Additional Qualities

Increasingly, projects are chartered to achieve more aggressive objectives such as the development of extensible designs, the design of reusable frameworks, or highly portable systems. The products of the analysis and design phases of these projects are most critical for achieving these types of objectives. In particular, the architecture is key to the success. Guided inspection can be used to ask metalevel questions about the system. In this mode, the test scenarios are developer actions on the system and not user actions. Instead of asking how the objects in the system would interact, the questions is, "how must the classes of the system be changed to provide the newly required behavior?"

The changes to the design or the revisions needed to produce a framework from an existing application can be captured as **change cases** [EcDe96]. A change case is a use case that is not a requirement of the system, but it is an anticipated change to the system. Guided inspection applies correctness, completeness, and consistency criteria to the current analysis and design models with the change cases as the source of test scenarios.

JMCH04.fm Page 152 Wednesday, December 27, 2000 10:17 AM



For example, if the project is to build a framework upon which future development will be based, it is not sufficient to test against the current uses for the system. Consider the change case shown in Figure 4.37. Test cases can give insight into the effort required to extend the framework by testing how complete the existing model is relative to the new requirements. The second change case, as shown in Figure 4.38, could be used to test the architecture. It would be used to determine how completely the existing architecture covers the new requirement.

Change Case #1

Actor: Player

Use Scenario: The player bounces the puck off several bricks. Some of the bricks break and disappear, only but two of the bricks crack.

Alternative Paths:

Exceptional Cases:

Frequency: Low

Criticality: Low

Risk: Medium

Figure 4.37 A change case for *Brickles*

Change Case #2

Actor: Player

Use Scenario: The player directs the puck against a set of nonmoving, nonbreakable barriers. The player must move the paddle more quickly in order to catch the puck as it bounces off the closer obstacles.

Alternative Paths:

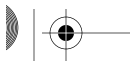
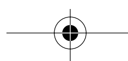
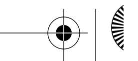
Exceptional Cases:

Frequency: Low

Criticality: Low

Risk: Medium

Figure 4.38 A second change case

The technique for testing these objectives can be viewed as a series of steps. Each of the steps is described and accompanied by an example.

- Explicitly state the objective that the change case will address.

  The design will be easily extensible to accommodate new games.

- Construct a "change case" including a specific scenario that illustrates the objective.

  The framework is to be used to implement pinball games that are user configurable. The obstacles to be available include posts, flippers, and bumpers.

- Create test cases by sampling from the range permitted by the change case.

  A pinball game is to be created. Figure 4.39 illustrates two new states that might be added to the ***Brickles*** state machine.

- Enumerate the work needed to achieve the objective by specifying the differences in state and behavior required for the new objective. This can be accomplished by identifying the new subclasses that must be defined.

  The `StationarySprite` class will be subclassed to provide the new obstacles. A `Ball` class will extend `MovableSprite` and `CollideWithBall` methods would be required for all sprites. Attributes will be added to give a specific point value to each obstacle. The `PinBall` subclass of `ArcadeGame` would add a `Score` attribute.

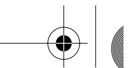
- Evaluate the current design relative to the design required to achieve the objective. Answer the following questions: "Are there fundamental concepts missing that would have to be added?" and "Are there contradictions between what exists and what would be added?"

  The necessary base classes and methods are present. The needed attributes can be added without conflict with existing attributes. However, the `Sprite` class must be modified.

- Repeat with additional test scenarios until all proposed changes are examined.

The output of this process is a set of potential changes needed to achieve the desired system quality, such as extensibility. The inspection searches for missing concepts and contradictions between what exists in the model currently and what would need to be added to the model in order to achieve the new objective. This technique can provide early feedback to the development team about fundamental weaknesses in the design.

JMCH04.fm Page 154 Wednesday, December 27, 2000 10:17 AM



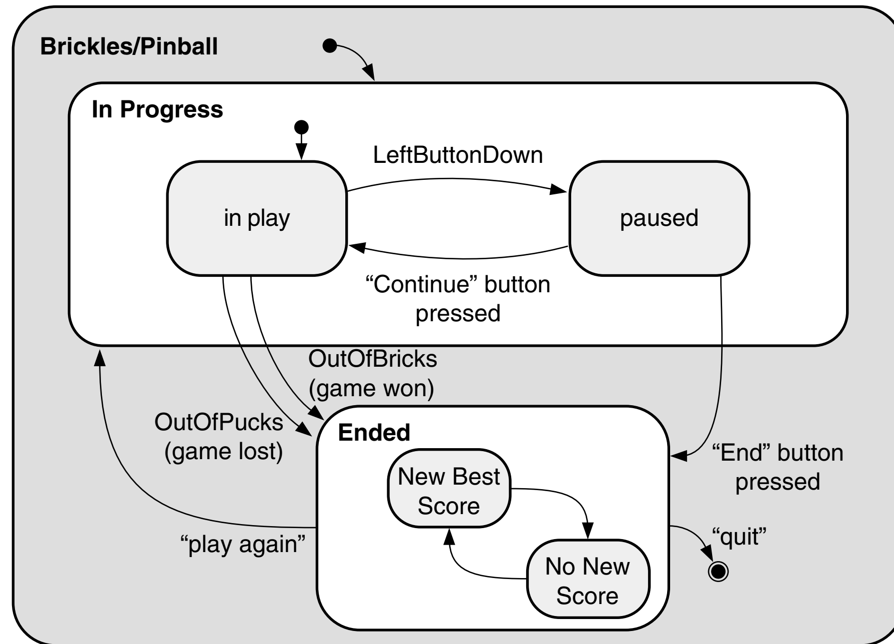


Figure 4.39 A state diagram for a pinball game

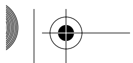
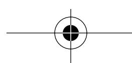
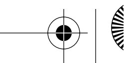
Summary

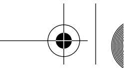
The techniques presented in this chapter are sometimes people intensive and scenarios should be systematically selected for maximum effectiveness. In particular choosing use cases that are less well understood or that represent high-risk situations is recommended. This maximizes the likelihood of finding errors and omissions that will have maximum impact on the quality of the system.

Figure 4.40 illustrates much of what we want you to get out of this chapter. The UML diagrams have been developed so that they are mutually supportive. The test cases developed in this chapter provide guidance for making a systematic search of the models for potential defects.

Why is this technique in a book on testing? First, it uses a testing perspective to focus the examination of the models. Second, much of this testing uses a system-wide scope and thus it is a natural role for the system testers. They can participate in the development process from the earliest phases if they have responsibility for developing test cases from the use cases. In this chapter, we have presented an approach that will identify defects early in the development process and will support the early involvement of the test community, and the class integration and system testers in the development project.

The checklist in Figure 4.41 summarizes the tasks described in this chapter.



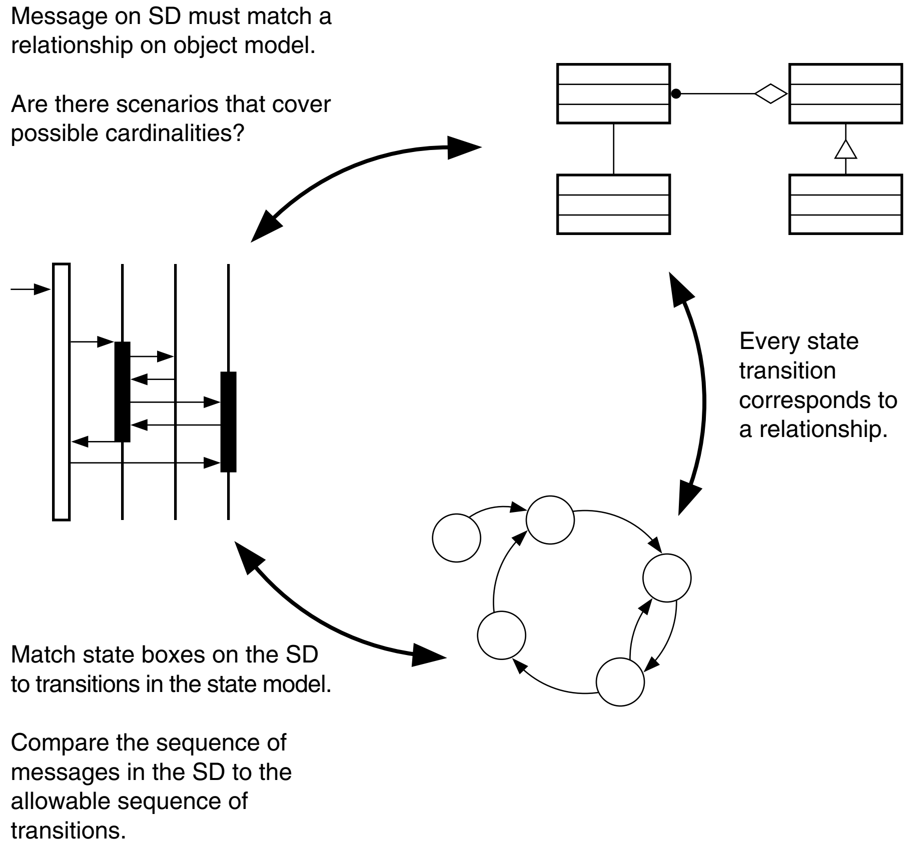
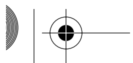
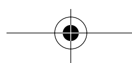
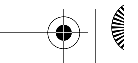
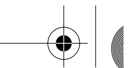


Figure 4.40 Consistency between diagrams

Model-Testing Checklist

The steps in this checklist are intended to ensure that all of the required activities in the guided inspection process are completed. A detailed process is defined in the addendum to this chapter.



Guided Inspection Process CheckList Step	*Done*
Decide how completeness, consistency and correctness will be judged for the particular use in the model under test (MUT).	
Determine which scenarios to sample from the use-case model to use as test cases.	
Create test cases by supplementing the scenarios with specific data.	
Select the model/notation that will record the results of each execution.	
Conduct tests.	
Evaluate the results of the test executions to determine which tests the model passed and which it failed.	
Record the results for use in guiding the repair and testing processes in the next iteration.	

Figure 4.41 A guided inspection process checklist

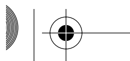
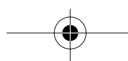
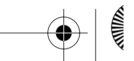
Exercises

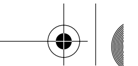
4-1. Using the use cases of your project as a starting point, you add the Frequency and Criticality fields to them. Perform the risk analysis for the use cases and fill in that field for each use case. Develop an ordering of uses that reflects how intensely each should be tested. Write test cases for the use cases in the model. Write one test case for the least important use case. Write additional test cases for each of the use cases.

4-2. Conduct a guided inspection session for the initial analysis model for your project. Generate a report that lists the discrepancies between the models.

4-3. Develop three scenarios from the use cases given in this chapter. Then, using the models given in Chapter 2 and Chapter 4, identify examples of incompleteness, inconsistency, and incorrectness.

4-4. Select the phase in your software in which development process, in your estimation, more defects are created than any other. Design a guided inspection checklist that will lead developers to finding the types of defects created in that phase.



Addendum: A Process Definition for Guided Inspection

Goal: To identify defects in artifacts created during the analysis and design phases of software construction.

Steps in the Process

1. Define the scope and depth of the guided inspection.
2. Identify the basis model(s) from which the material being inspected was created.
3. Assemble the guided inspection team.
4. Define a sampling plan and coverage criteria.
5. Create test cases from the bases.
6. Apply the tests to the material.
7. Gather and analyze test results.
8. Report and feedback.

Detailed Step Descriptions

1. Define the scope and depth of the review/inspection.

   Inputs:

   The project's position in the life cycle.

   The materials produced by the project (UML models, plans, use cases).

   Outputs:

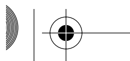
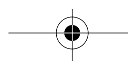
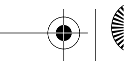
   A specific set of diagrams and documents that will be the basis for the evaluation

   Method:

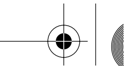
   Define the scope of the guided inspection to be the set of deliverables from a phase of the development process. Use the development-process information to identify the deliverables that will be produced by the phase of interest.

   Example:

   The project has just completed the domain-analysis phase. The development process defines the deliverable from this phase as a UML model containing domain-level use cases, static information such as class diagrams, and dynamic information such as sequence and state diagrams. The guided inspection will evaluate this model.

JMCH04.fm Page 158 Wednesday, December 27, 2000 10:17 AM



2. Identify the basis model(s) from which the material being inspected was created.

   Inputs:

   The scope of the guided inspection.

   The project's position in the life cycle.

   Outputs:

   The material from which the test cases will be constructed (the model under test—MUT).

   Method:

   Review the development process description to determine the inputs to the current phase. The basis model(s) should be listed as inputs to the current phase.

   Example:

   The inputs to the domain-analysis phase is the "knowledge of experts familiar with the domain." These mental models are the basis models for this guided inspection.

3. Assemble the guided inspection team.

   Inputs:

   The scope of the guided inspection.

   Available personnel.

   Outputs:

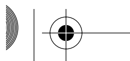
   A set of participants and their roles.

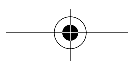
   Method:

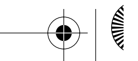
   Assign persons to fill one of three categories of roles: Administrative, Participant in creating the model to be tested, Objective observer of the model to be tested. Choose the objective observers from the customers of the model to be tested and the participants during the creation of the basis model.

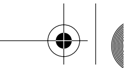
   Example:

   Since the model to be tested is a domain analysis model and the basis model is the mental models of the domain experts, the objective observers can be selected from other domain experts and/or from application analysts. The creation participants are members of the domain-modeling team. The administrative personnel can perhaps come from other interested parties or an office that provides support for the conduct of guided inspections.









4. Define a sampling plan and coverage criteria.

   Inputs:

   The project's quality plan.

   Outputs:

   A plan for how test cases will be selected.

   A description of what parts of the MUT will be covered.

   Method:

   Identify important elements of this MUT. Estimate the effort required to involve all of these in the guided inspection. If there are too many to cover, use information such as the RISK section of the use cases or the judgement of experts to prioritize the elements.

   Example:

   In a domain model there are static and dynamic models as well as use cases. At least one test case should be created for each use case. There should be sufficient test cases to take every "major" entity through all of its visible states.

5. Create test cases from the bases.

   Inputs:

   The sampling plan.

   MUT.

   Outputs:

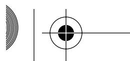
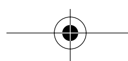
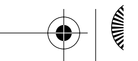
   A set of test cases.

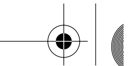
   Method:

   Obtain a scenario from the basis model. Determine the preconditions and inputs that are required to place the system in the correct state and to begin the test. Present the scenario to the "oracle" to determine the results expected from the test scenario. Complete a test case description for each test case.

   Example:

   A different domain expert than the one who supported the model creation would be asked to supply scenarios that correspond to uses of the system. The experts also provide what they would consider an acceptable response.



6. Apply the tests to the material.

Inputs:

Set of test cases.

MUT

Outputs:

Set of test results.

Method:

Apply the test cases to the MUT using the most specific technique available. For UML models in a static environment, such as Rational Rose, an interactive simulation session in which the Creators play the roles of the model elements is the best approach. If the MUT is represented by an executable prototype then the test cases are mapped onto this system and executed.

Example:

The domain-analysis model is a static UML model. A simulation session is conducted with the Observers feeding test cases to the Creators. The Creators provide details of how the test scenario would be processed through the model. Sequence diagrams document the execution of each test case. Use agreed-upon symbols or colors to mark each element that is touched by a test case.

7. Gather and analyze test results and coverage.

Inputs:

Test results in the form of sequence diagrams and pass/fail decisions. The marked-up model.

Outputs:

Statistics on percentage pass/fail.

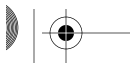
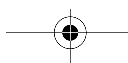
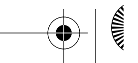
Categorization of the results.

Defect catalogs and defect reports.

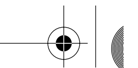
A judgement of the quality of the MUT and the tests.

Method:

Begin by counting the number of test cases that passed and how many have failed. Compare this ratio to other guided inspections that have been conducted in the organization. Compute the percentage of each type of element that has been used in executing the test cases. Use the marked-up model as the source of this data. Update the defect inventory with information about the failures from this test session.

JMCH04.fm Page 161 Wednesday, December 27, 2000 10:17 AM



Categorize the failed test cases. This can often be combined with the previous two tasks by marking paper copies of the model. Follow the sequence diagram for each failed test case and mark each message, class, and attribute touched by a failed test case.

Example:

For the domain-analysis model we should be able to report that every use case was the source of at least one test case, and that every class in the class diagram was used at least once. Typically, on the first pass, some significant states will be missed. This should be noted in the coverage analysis.

8. Report and feedback.

Inputs:

Test results.

Coverage information.

Outputs:

Information on what new tests should be created.

Test report.

Method:

Follow the standard format for a test report in your organization to document the test results and the analyses of those results. If the stated coverage goals are met then the process is complete. If not, use that report to return to Step 5 and proceed through the steps to improve the coverage level.

Example:

For the domain-analysis tests, some elements were found to be missing from the model. The failing tests might be executed again after the model has been modified.

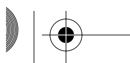
Roles in the Process

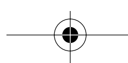
Administrator

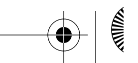
The administrative tasks include running the guided inspection sessions, collecting and disseminating the results, and aggregating metrics to measure the quality of the review. In our example, the administrative work could be done by personnel from a central office.

Creator

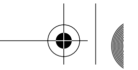
The persons who created the MUT. Depending on the form that the model takes, these people may “execute” the symbolic model on the test cases or they may assist in translating the test cases into a form







JMCH04.fm Page 162 Wednesday, December 27, 2000 10:17 AM



that can be executed with whatever representation of the model is available. In our example the modelers who created the domain model would be the "creators."

Observer

Persons in this role create the test cases that are used in the guided inspection. In our example they would be domain experts and preferably experts who were not the source of the information that was used to create the model initially.