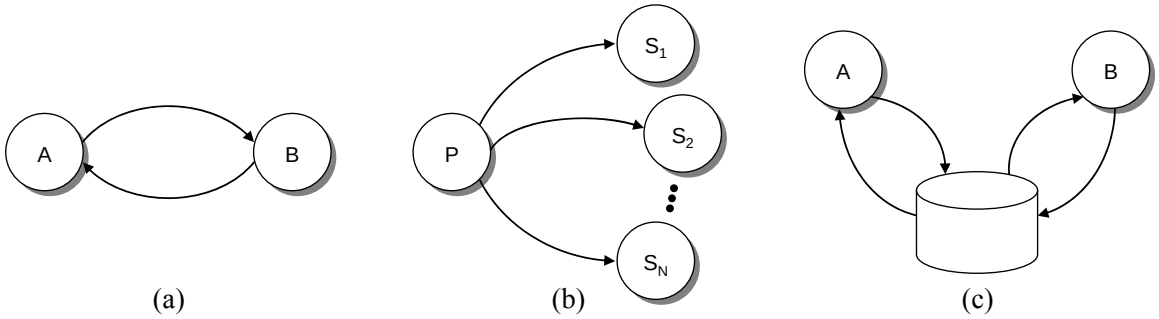


Figure 2-37: Example object communication patterns. (a) One-to-one direct messages. (b) One-to-many untargeted messages. (c) Via a shared data element.

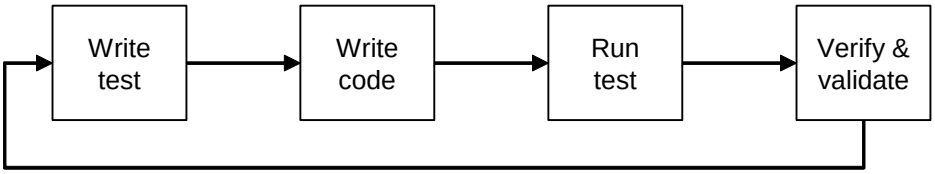


Figure 2-38: Test-driven implementation.

Note that objects almost never play an exclusive role; several roles are usually imparted to different degree in each object.

Object Communication Patterns

Communication pattern is a *message-sending relation* imposed on a set of objects. As with any relation, it can be one-to-one or one-to-many and it can be deterministic or random (Section 3.1.1). Some of these patterns are illustrated in Figure 2-37.

Object-oriented design, particularly design patterns, is further elaborated in Chapter 5.

2.6.3 Why Software Engineering Is Difficult (3)

Another key cause is the lack of analytical methods for software design. Software engineers are aiming at optimal designs, but quantitative criteria for optimal software design are largely unknown. Optimality criteria appear to be mainly based upon judgment and experience.

2.7 Test-driven Implementation

"The good news about computers is that they do what you tell them to do. The bad news is that they do what you tell them to do." —Ted Nelson

Given a feature selected for implementation, test-driven implementation works by writing the code for tests, writing the code that implements the feature, running the tests, and finally verifying and validating the test results (Figure 2-38). If the results meet the expectations, we move onto the next feature; otherwise, we need to debug the code, identify and fix the problem, and test again.

2.7.1 Overview of Software Testing

"Testing shows the presence, not the absence of bugs." —Edsger W. Dijkstra

Testing is often viewed as executing a program to see if it produces the correct output for a given input. This implies testing the end-product, the software itself, which in turn means that testing activities are postponed until late in the lifecycle. This is wrong because experience has shown that errors introduced during the early stages of software lifecycle are the costliest and most difficult to discover. A more general definition is that testing is the process of finding faults in software artifacts, such as UML diagrams or code. A **fault**, also called "defect" or "bug," is an erroneous hardware or software element of a system that can cause the system to *fail*, i.e., to behave in a way that is not desired or even harmful. We say that the system experienced *failure* because of an inbuilt fault.

Any software artifact can be tested, including requirements specification, domain model, and design specification. Testing activities should be started as early as possible. An extreme form of this approach is *test-driven development* (TDD), one of the practices of Extreme Programming (XP), in which development *starts* with writing tests. The form and rigor of testing should be adapted to the nature of the artifact that is being tested. Testing of design sketches will be approached differently than testing a software code.

Testing works by *probing* a program with different combinations of inputs to detect faults. Therefore, testing shows only the presence of faults, not their absence. Showing the absence of faults requires exhaustively trying all possible combinations of inputs (or following all possible paths through the program). The number of possible combinations generally grows exponentially with software size. However, it is not only about inadvertent bugs—a bad-intended programmer might have introduced purposeful malicious features for personal gain or revenge, which are activated only by a very complex input sequence. Therefore, it is impossible to test that a program will work correctly for all imaginable input sequences. An alternative to the brute force approach of testing is to prove the correctness of the software by *reasoning* (or, theorem proving). Unfortunately, proving correctness generally cannot be automated and requires human effort. In addition, it can be applied only in the projects where the requirements are specified in a formal (mathematical) language. We will discuss this topic further in Chapter 3.

A key tradeoff of testing is between testing as many possible cases as possible while keeping the economic costs limited. Our goal is to find faults as cheaply and quickly as possible. Ideally, we would design a single "right" test case to expose each fault and run it. In practice, we have to run many "unsuccessful" test cases that do not expose any faults. Some strategies that help keep costs down include (*i*) complementing testing with other methods, such as design/code review, reasoning, or static analysis; (*ii*) exploiting automation to increase coverage and frequency of testing; and (*iii*) testing early in the lifecycle and often. Automatic checking of test results is

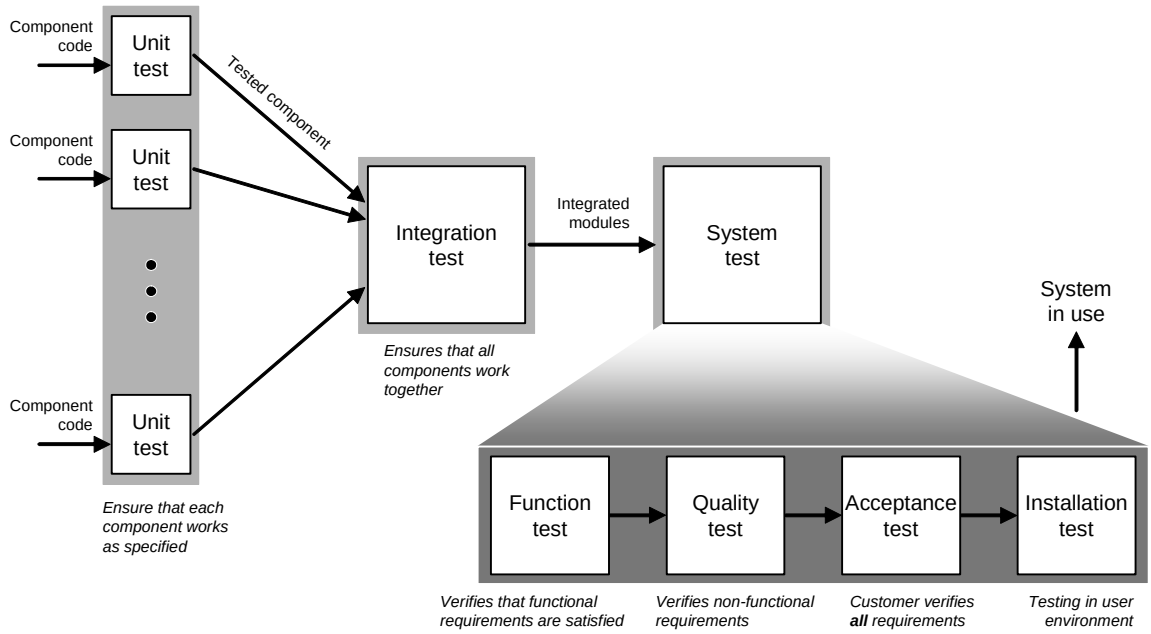


Figure 2-39:Logical organization of software tests.

preferred to keep the costs low, but may not always be feasible. For example, how to check the display content of a graphical user interface?

Testing is usually guided by the hierarchical structure of the system (software architecture, Section 2.3) as designed in the analysis and design phases (Figure 2-39). We may start by testing individual components, which is known as **unit testing**. These components are incrementally integrated into a system. Testing the composition of the system components is known as **integration testing**. **System testing** ensures that the whole system complies with the functional and non-functional requirements. The customer performs **acceptance testing** of the whole system. (Acceptance tests and examples are described in Sections 2.2 and 2.4, when describing requirements engineering.) As always, the logical organization does not imply that testing steps should be ordered in time as shown in Figure 2-39. Instead, the development lifecycle evolves incrementally and iteratively, and corresponding cycles will occur in testing as well.

Unit testing finds differences between the object design model and its corresponding implementation. There are several benefits of focusing on individual components. One is the common advantage of the divide-and-conquer approach—it reduces the complexity of the problem and allows us to deal with smaller parts of the system separately. Second, unit testing makes it easier to locate and correct faults because only few components are involved in the process. Lastly, unit testing supports division of labor, so several team members can test different components in parallel. Practical issues with unit testing are described in Section 2.7.3.

Regression testing seeks to expose new errors, or "regressions," in existing functionality after changes have been made to the system. A new test is added for every discovered fault, and tests are run after every change to the code. Regression testing helps to populate test suite with good test cases, because every regression test is added *after* it uncovered a fault in one version of the code. Regression testing protects against reversions that reintroduce faults. Because the fault that resulted in adding a regression test already happened, it may be an easy error to make again.

Another useful distinction between testing approaches is what document or artifact is used for designing the test cases. **Black box testing** refers to analyzing a running program by probing it with various inputs. It involves choosing test data only from the specification, without looking at the implementation. This testing approach is commonly used by customers, for example for acceptance testing. **White box testing** chooses test data with knowledge of the implementation, such as knowledge of the system architecture, used algorithms, or program code. This testing approach assumes that the code implements all parts of the specification, although possibly with bugs (programming errors). If the code omitted a part of the specification, then the white box test cases derived from the code will have incomplete coverage of the specification. White box tests should not depend on specific details of the implementation, which would prevent their reusability as the system implementation evolves.

2.7.2 Test Coverage and Code Coverage

Because exhaustive testing often is not practically achievable, a key issue is to know when we have done enough testing. **Test coverage** measures the degree to which the specification or code of a software program has been exercised by tests. In this section we interested in a narrower notion of **code coverage**, which measures the degree to which the *source code* of a program has been tested. There are a number of code coverage criteria, including equivalence testing, boundary testing, control-flow testing, and state-based testing.

To select the test inputs, one may make an *arbitrary* choice of what one "feels" should be appropriate input values. A better approach is to select the inputs *randomly* by using a random number generator. Yet another option is choosing the inputs *systematically*, by partitioning large input space into a few representatives. Arbitrary choice usually works the worst; random choice works well in many scenarios; systematic choice is the preferred approach.

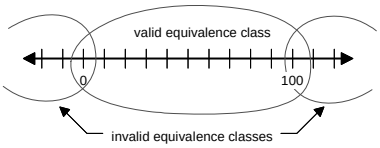
Equivalence Testing

Equivalence testing is a black-box testing method that divides the space of all possible inputs into equivalence groups such that the program "behaves the same" on each group. The goal is to reduce the total number of test cases by selecting representative input values from each equivalence group. The assumption is that the system will behave similarly for all inputs from an equivalence group, so it suffices to test with only a single element of each group. Equivalence testing has two steps: (*i*) partitioning the values of input parameters into equivalence groups and (*ii*) choosing the test input values.

The trouble with this approach is that it is just as hard to find the equivalence classes of inputs as it is to prove correctness. Therefore, we use heuristics (rules of thumb that are generally useful but do not guarantee correctness) to select a set of test cases. We are essentially guessing based on experience and domain knowledge, and hoping that at least one of the selected test cases belongs to each of the true (unknown) equivalence classes.

Partitioning the values of input parameters into equivalence classes may be performed according to the following heuristics:

- For an input parameter specified over a range of values, partition the value space into one valid and two invalid equivalence classes. For example, if the allowed input values are integers between 0 and 100, the valid equivalence class



contains integers between 0 and 100, one invalid equivalence class contains all negative integers, and the other invalid equivalence class contains all integers greater than 100.

- For an input parameter specified with a single value, partition the value space into one valid and two invalid equivalence classes. For example, if the allowed value is a real number 1.4142, the valid equivalence class contains a single element {1.4142}, one invalid equivalence class contains all real number smaller than 1.4142, and the other invalid equivalence class contains all real number greater than 1.4142.
- For an input parameter specified with a set of values, partition the value space into one valid and one invalid equivalence class. For example, if the allowed value is any element of the set {1, 2, 4, 8, 16}, the valid equivalence class contains the elements {1, 2, 4, 8, 16}, and the invalid equivalence class contains all other elements.
- For an input parameter specified as a Boolean value, partition the value space into one valid and one invalid equivalence class (one for TRUE and the other for FALSE).

Equivalence classes defined for an input parameter must satisfy the following criteria:

1. *Coverage*: Every possible input value belongs to an equivalence class.
2. *Disjointedness*: No input value belongs to more than one equivalence class.
3. *Representation*: If an operation is invoked with one element of an equivalence class as an input parameter and returns a particular result, then it must return the same result if any other element of the class is used as input.

If an operation has more than one input parameter, we must define new equivalence classes for combinations of the input parameters (known as Cartesian product or cross product, see Section 3.2.1).

For example, consider testing the Key Checker's operation `checkKey(k : Key) : boolean`. As shown in Figure 2-35, the class Key has three string attributes: `code`, `timestamp`, and `doorLocation`. The operation `checkKey()` as implemented in Listing 2-4 does not use `timestamp`, so its value is irrelevant. However, we need to test that the output of `checkKey()` does not depend on the value of `timestamp`. The other two attributes, `code` and `doorLocation`, are specified with a set of values for each. Suppose that the system is installed in an apartment building with the apartments numbered as {196, 198, 200, 202, 204, 206, 208, 210}. Assume that the attribute `doorLocation` takes the value of the associated apartment number. On the other hand, the tenants may have chosen their four-digit access codes as {9415, 7717, 8290, …, 4592}. Although a `code` value "9415" and `doorLocation` value "198" are each valid separately, their combination is invalid, because the `code` value for the tenant in apartment 198 is "7717."

Therefore, we must create a cross product of `code` and `doorLocation` values and partition this value space into valid and invalid equivalence classes. For the pairs of test input values chosen from the valid equivalence class, the operation `checkKey()` should return the Boolean value TRUE. Conversely, for the pairs of test input values from invalid equivalence classes it should return FALSE.

When ensuring test coverage, we should consider not only the current snapshot, but also historic snapshots as well. For example, when testing the Key Checker's operation `checkKey()`, the

previously-valid keys of former tenants of a given apartment belong to an invalid equivalence class, although in the past they belonged to the valid equivalence class. We need to include the corresponding test cases, particularly during integration testing (Section 2.7.4).

Boundary Testing

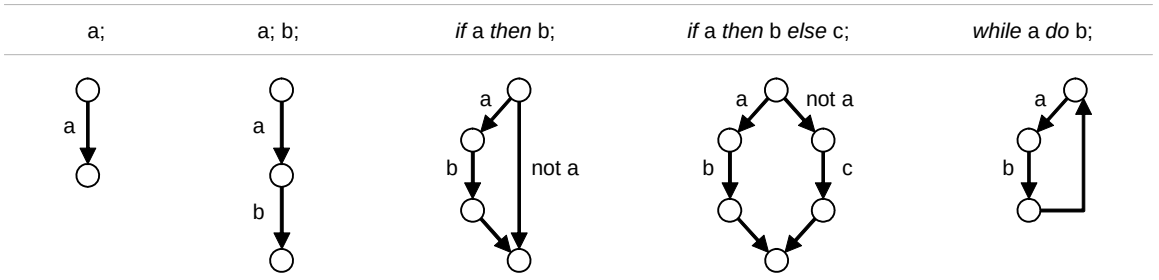
Boundary testing is a special case of equivalence testing that focuses on the boundary values of input parameters. After partitioning the input domain into equivalence classes, we test the program using input values not only "inside" the classes, but also at their boundaries. Rather than selecting any element from an equivalence class, boundary testing selects elements from the "edges" of the equivalence class, or "outliers," such as zero, min/max values, empty set, empty string, and null. Another frequent "edge" fault results from the confusion between > and >=. The assumption behind this kind of testing is that developers often overlook special cases at the boundary of equivalence classes.

For example, if an input parameter is specified over a range of values from *a* to *b*, then test cases should be designed with values *a* and *b* as well as just above and just below *a* and *b*.

Control Flow Testing

Statement coverage selects a test set such that every elementary statement in the program is executed at least once by some test case in the test set.

Edge coverage selects a test set such that every edge (branch) of the control flow is traversed at least once by some test case. We construct the *control graph* of a program so that statements become the graph edges, and the nodes connected by an edge represent entry and exit to/from the statement. A sequence of edges (without branches) should be collapsed into a single edge.



Condition coverage (also known as *predicate coverage*) selects a test set such that every condition (Boolean statement) takes TRUE and FALSE outcomes at least once in some test case.

Path coverage determines the number of distinct paths through the program that must be traversed (travelled over) at least once to verify the correctness. This strategy does not account for loop iterations or recursive calls. Cyclomatic complexity metric (Section 4.2.2) provides a simple way of determining the number of independent paths.

State-based Testing

State-based testing defines a set of abstract states that a software unit can take and tests the unit's behavior by comparing its actual states to the expected states. This approach has become popular with object-oriented systems. The *state* of an object is defined as a constraint on the values of object's attributes. Because the methods use the attributes in computing the object's behavior, the behavior depends on the object state.

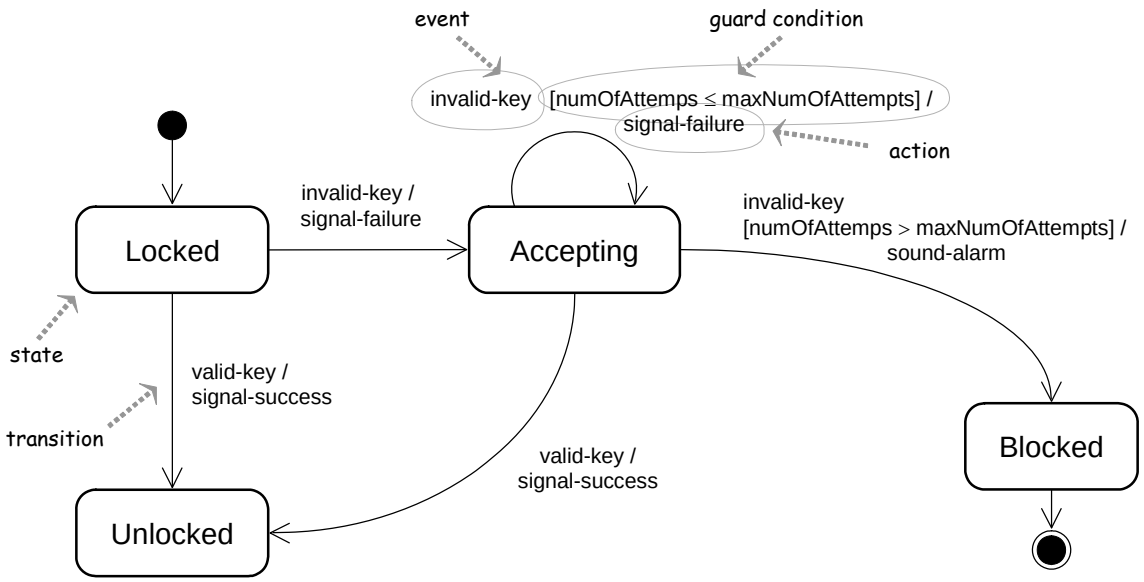


Figure 2-40: UML state diagram for the Controller class in Figure 2-35. The notation for UML state diagrams is introduced in Section 3.2.

The first step in using state-based testing is to derive the state diagram for the tested unit. We start by defining the states. Next, we define the possible transitions between states and determine what triggers a transition from one state to another. For a software class, a state transition is usually triggered when a method is invoked. Then we choose test values for each individual state.

The second step is to initialize the unit and run the test. The test driver exercises the unit by calling methods on it, as described in Section 2.7.3. When the driver has finished exercising the unit, assuming no errors have yet occurred, the test then proceeds to compare the actual state of the unit with its expected state. If the unit reached the expected state, the unit is considered correct regardless of how it got to that state.

Assume that we are to test the Controller class of our safe home access case study (the class diagram shown in Figure 2-35). The process of deriving the state diagrams and UML state diagram notation are described in Chapter 3. A key responsibility of the Controller is to prevent the dictionary attacks by keeping track of unsuccessful attempts because of an invalid key. Normally, we assume that the door is locked (as required by REQ1 in Table 2-1). The user unlocks the door by providing a valid key. If the user provided an invalid key, the Controller will allow up to maxNumOfAttempts unsuccessful attempts, after which it should block and sound alarm. Therefore, we identify the following elements of the state diagram (Figure 2-40):

- Four states { Locked, Unlocked, Accepting, Blocked }
- Two events { valid-key, invalid-key }
- Five valid transitions { Locked→Unlocked, Locked→Accepting, Accepting→Accepting, Accepting→Unlocked, Accepting→Blocked }

A test set consists of scenarios that exercise the object along a given path through the state diagram. In general the number of state diagram elements is

$$\text{all-events, all-states} \leq \text{all-transitions} \leq \text{all-paths}$$

Because the number of possible paths in the state diagram is generally infinite, it is not practical to test each possible path. Instead, we ensure the following coverage conditions:

- Cover all identified states at least once (each state is part of at least one test case)
- Cover all valid transitions at least once
- Trigger all invalid transitions at least once

Testing all valid transitions implies (subsumes) all-events coverage, all-states coverage, and all-actions coverage. This is considered a minimum acceptable strategy for responsible testing of a state diagram. Note that all-transitions testing is *not* exhaustive, because exhaustive testing requires that every path over the state machine is exercised at least once, which is usually impossible or at least unpractical.

2.7.3 Practical Aspects of Unit Testing

Executing tests on single components (or "units") or a composition of components requires that the tested thing be isolated from the rest of the system. Otherwise we will not be able to localize the problem uncovered by the test. But system parts are usually interrelated and cannot work without one another. To substitute for missing parts of the system, we use test drivers and test stubs. A **test driver** simulates the part of the system that invokes operations on the tested component. A **test stub** is a minimal implementation that simulates the components which are called by the tested component. The thing to be tested is also known as the **fixture**.

A stub is a trivial implementation of an interface that exists for the purpose of performing a unit test. For example, a stub may be hard-coded to return a fixed value, without any computation. By using stubs, you can test the interfaces without writing any *real* code. The implementation is really not necessary to verify that the interfaces are working properly (from the client's perspective—recall that interfaces are meant for the client object, Section 1.4). The driver and stub are also known as **mock objects**, because they pretend to be the objects they are simulating.

Each testing method follows this cycle:

1. Create the thing to be tested (fixture), the test driver, and the test stub(s)
2. Have the test driver invoke an operation on the fixture
3. Evaluate that the results are as expected

More specifically, a **unit test case** comprises three steps performed by the test driver:

1. *Setup* objects: create an object to be tested and any objects it depends on, and set them up
2. *Act* on the tested object
3. *Verify* that the outcome is as expected

Suppose you want to test the Key Checker class of the safe-home-access case study that we designed in Section 2.6. Figure 2-41(a) shows the relevant excerpt sequence diagram extracted from Figure 2-33. Class `Checker` is the tested component and we need to implement a *test driver* to substitute `Controller` and *test stubs* to substitute `KeyStorage` and `Key` classes.

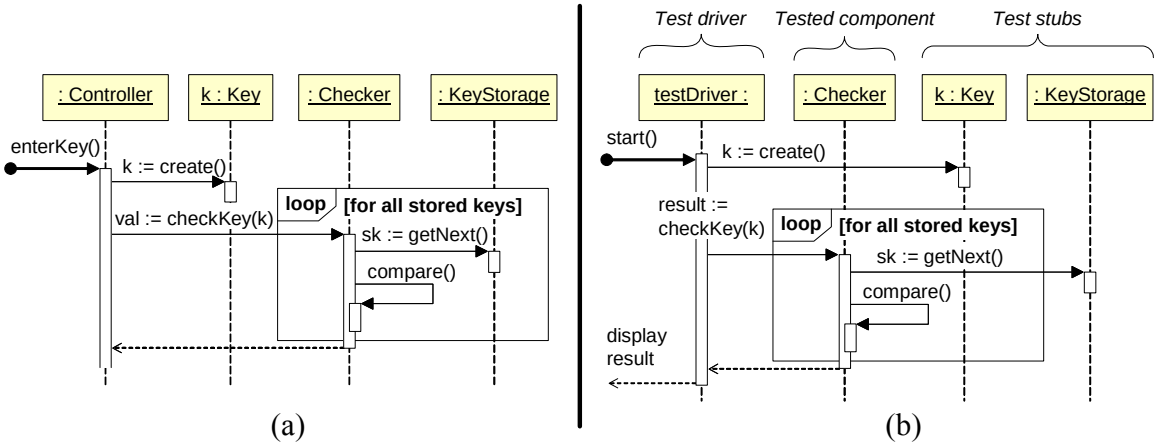


Figure 2-41: Testing the Key Checker's operation `checkKey()` (use case Unlock). (a) Relevant part of the sequence diagram excerpted from Figure 2-33. (b) Test stubs and drivers for testing the Key Checker.

As shown in Figure 2-41(b), the test driver passes the test inputs to the tested component and displays the results. In JUnit testing framework for Java, the result verification is done using the `assert*()` methods that define the expected state and raise errors if the actual state differs. The test driver can be any object type, not necessarily an instance of the `Controller` class. Unlike this, the test stubs must be of the same class as the components they are simulating. They must provide the same operation APIs, with the same return value types. The implementation of test stubs is a nontrivial task and, therefore, there is a tradeoff between implementing accurate test stubs and using the actual components. That is, if `KeyStorage` and `Key` class implementations are available, we could use them when testing the Key Checker class.

Listing 2-1: Example test case for the Key Checker class.

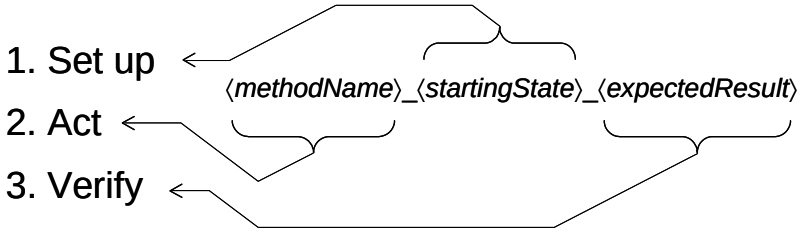
```
public class CheckerTest {
  // test case to check that invalid key is rejected
  @Test public void
    checkKey_anyState_invalidKeyRejected() {

    // 1. set up
    Checker checker = new Checker( /* constructor params */ );

    // 2. act
    Key invalidTestKey = new Key( /* setup with invalid code */ );
    boolean result = checker.checkKey(invalidTestKey);

    // 3. verify
    assertEqual(result, false);
  }
}
```

We use the following notation for methods that represent test cases (see Listing 2-1):



where *methodName* is the name of the method (i.e., event) we are testing on the tested object; *startingState* are the conditions under which the tested method is invoked; and, *expectedResult* is what we expect the tested method to produce under the specified condition. In our example, we are testing `Checker`'s method `checkKey()`. The `Checker` object does not have any attributes, so it is always in an initial state. The expected result is that `checkKey()` will reject an invalid key. Thus the test case method name `checkKey_anyState_invalidKeyRejected()`.

Testing objects with different states is a bit more complex, because we must bring the object to the tested state and in the end verify that the object remains in an expected state. Consider the `Controller` object and its state diagram shown in Figure 2-40. One test case needs to verify that when `Controller` receives maxNumOfAttempts invalid keys, it correctly transitions to the Blocked state.

Listing 2-2: Example test case for the Controller class.

```
public class ControllerTest {
  // test case to check that the state Blocked is visited
  @Test public void
    enterKey_accepting_toBlocked() {

    // 1. set up: bring the object to the starting state
    Controller cntrl = new Controller( /* constructor params */ );
    // bring Controller to the Accepting state, just before it blocks
    Key invalidTestKey = new Key( /* setup with invalid code */ );
    for (i=0; i < cntrl.getMaxNumOfAttempts(); i++) {
        cntrl.enterKey(invalidTestKey);
    }
    assertEqual(   // check that the starting state is set up
        cntrl.getNumOfAttempts(), cntrl.getMaxNumOfAttempts() – 1
    );

    // 2. act
    cntrl.enterKey(invalidTestKey);

    // 3. verify
    assertEqual(   // the resulting state must be "Blocked"
        cntrl.getNumOfAttempts(), cntrl.getMaxNumOfAttempts()
    );
    assertEqual(cntrl.isBlocked(), true);
  }
}
```

It is left to the reader to design the remaining test cases and ensure the coverage conditions (Section 2.7.2).

A key challenge of unit testing is to sufficiently isolate the units so that each unit can be tested individually. Otherwise, you end up with a "unit" test that is really more like an integration test. The most important technique to help achieve this isolation is to program to interfaces instead of concrete classes.

2.7.4 Integration and Security Testing

In traditional methods, testing takes place relatively late in the development lifecycle and follows the logical order Figure 2-39. Unit testing is followed by integration testing, which in turn is followed by system testing. Integration testing works in a step-by-step fashion by linking together individual components ("units") and testing the correctness of the combined component. Components are combined in a *horizontal fashion* and integration processes in different direction, depending on the horizontal integration testing strategy.

In agile methods, testing is incorporated throughout the development cycle. Components are combined in a *vertical fashion* to implement an end-to-end functionality. Each vertical slice corresponds to a user story (Section 2.2.3) and user stories are implemented and tested in parallel.

Horizontal Integration Testing Strategies

There are various ways to start by combining the tested units. The simplest, known as **"big bang" integration** approach, tries linking all components at once and testing the combination.

Bottom-up integration starts by combining the units at the lowest level of hierarchy. The "hierarchy" is formed by starting with the units that have no dependencies to other units. For example, in the class diagram of Figure 2-35, classes `PhotoSObsrv`, `Logger`, and `DeviceCtrl` do not have navigability arrow pointing to any other class—therefore, these three classes form the bottommost level of the system hierarchy (Figure 2-42(a)). In bottom-up integration testing, the bottommost units ("leaf units") are tested first by unit testing (Figure 2-42(b)). Next, the units that have navigability to the bottommost units are tested in combination with the leaf units. The integration proceeds up the hierarchy until the topmost level is tested. There is no need to develop test stubs: The bottommost units do not depend on any other units; for all other units, the units on which the currently tested unit depends on are already tested. We do need to develop test drivers for bottom-up testing, although these can be relatively simple. Note that in real-world systems unit hierarchy may not necessarily form a "tree" structure, but rather may include cycles making it difficult to decide the exact level of a unit.

Top-down integration starts by testing the units at the highest level of hierarchy that no other unit depends on (Figure 2-42(c)). In this approach, we never need to develop test drivers, but we do need test stubs.

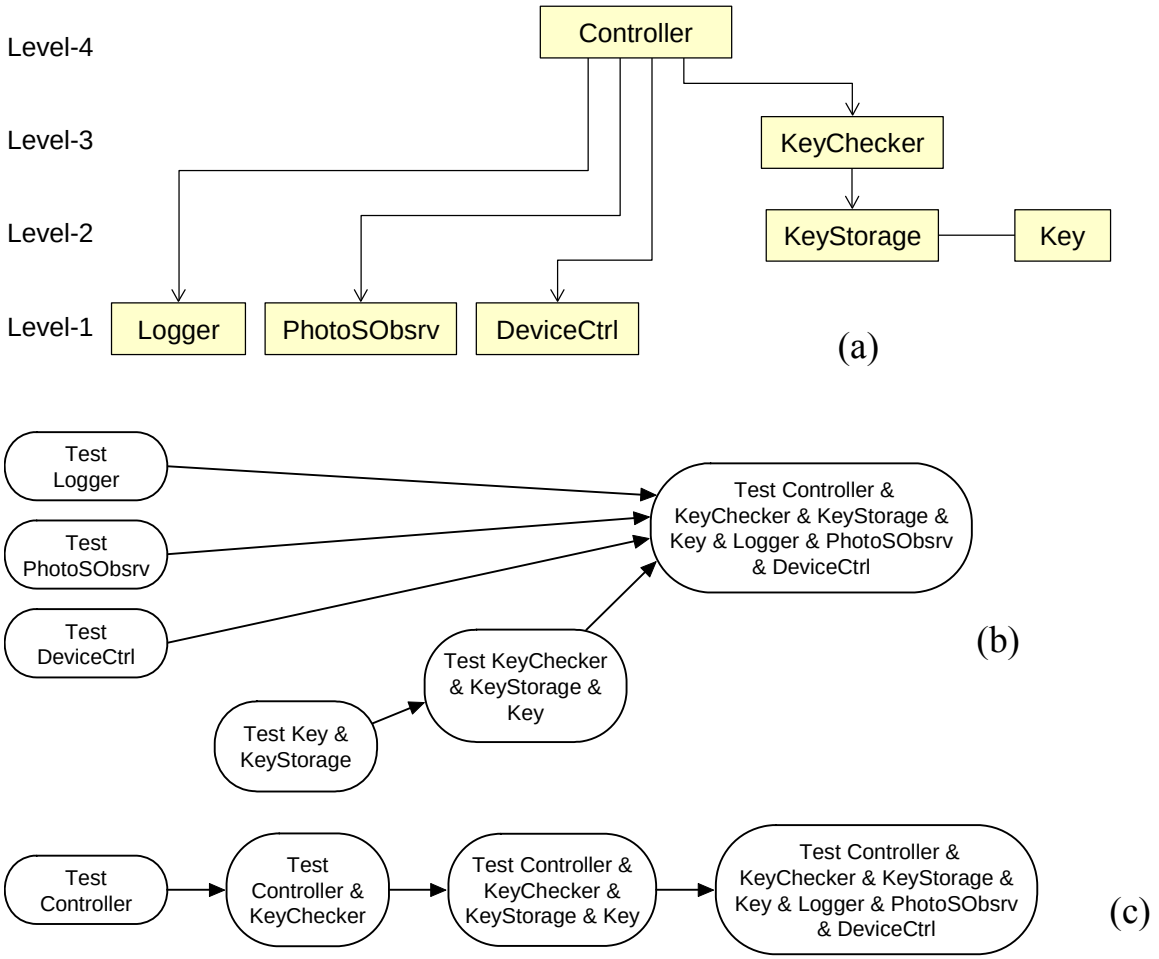


Figure 2-42: Integration testing strategies for the system from Figure 2-35. (a) Units hierarchy; (b) Bottom-up integration testing; (c) Top-down integration testing.

Sandwich integration approach combines top-down and bottom-up by starting from both ends and incrementally using components of the middle level in both directions. The middle level is known as the *target level*. In sandwich testing, usually there is need to write stubs for testing the top components, because the actual components from the target level can be used. Similarly, the actual target-level components are used as drivers for bottom-up testing of low-level components. In our example system hierarchy of Figure 2-42(a), the target layer contains only one component: Key Checker. We start by top-down testing of the Controller using the Checker. In parallel, we perform bottom-up testing of the Key Storage again by using the Checker. Finally, we test all components together.

There are advantages and drawbacks of each integration strategy. Bottom-up integration is suitable when the system has many low-level components, such as utility libraries. Moving up the hierarchy makes it easier to find the component-interface faults: if a higher-level component violates the assumption made by a lower-level component, it is easier to find where the problem is. A drawback is that the topmost component (which is usually the most important, such as user interface), is tested last—if a fault is detected, it may lead to a major redesign of the system.

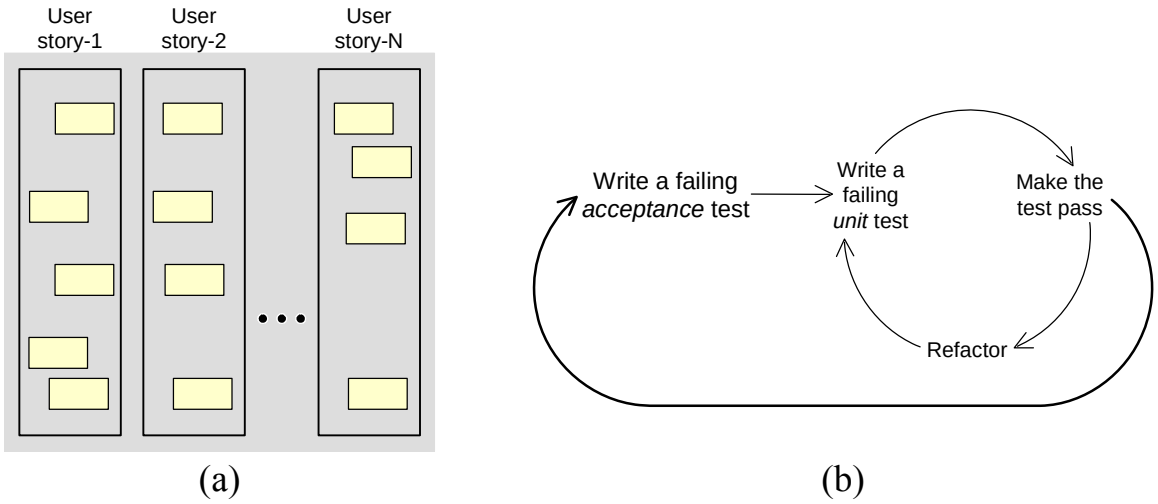


Figure 2-43: Vertical integration in agile methods develops functional vertical slices (user stories) in parallel (a). Each story is developed in a cycle that integrates unit tests in the inner feedback loop and the acceptance test in the outer feedback loop (b).

Top-down integration has the advantage of starting with the topmost component (usually the user interface, which means possibility of early end-user involvement). The test cases can be derived directly from the requirements. Its disadvantage is that developing test stubs is time consuming and error prone.

The advantages of sandwich testing include no need to write stubs or drivers and the ability of early testing of the user interface and thus early involvement of end users. A drawback is that sandwich testing does not thoroughly test the units of the target (middle) level before integration. This problem can be remedied by the *modified sandwich testing* that tests the lower, middle, and upper levels individually before combining them in incremental tests with one another.

Vertical Integration Testing Strategies

Agile methods use the *vertical integration* approach to develop the user stories in parallel (Figure 2-43(a)). Each story is developed in a feedback loop (Figure 2-43(b)), where the developers use unit tests in the inner loop and the customer runs the acceptance test in the outer loop. Each cycle starts with the customer/user writing the acceptance test that will test a particular user story. Based on the acceptance test, the developer writes the unit tests and develops only the code that is relevant, i.e., needed to pass the unit tests. The unit tests are run on daily basis, soon after the code is written, and the code is committed to the code base only after it passes the unit tests. The acceptance test is run at the end of each cycle (order of weeks or months).

The advantage of vertical integration is that it yields a working deliverable quickly. A potential drawback is that because each subsystem (vertical slice—user story) is developed independently, the system may lack uniformity and "grand design." Therefore, the system may need a major redesign late in the development cycle.

Security Testing

Functional testing is testing for "positives"—that the required features and functions are correctly implemented. However, a majority of security defects and vulnerabilities are not directly related to security functionality, such as encryption or privilege management. Instead, security issues involve often unexpected but intentional misuses of the system discovered by an attacker. Therefore, we also need to test for "negatives," such as abuse cases, to determine how the system behaves under attack. Security tests are often driven by known attack patterns.

2.7.5 Test-driven Implementation

"Real programmers don't comment their code. If it was hard to write, it should be hard to understand."
—Unknown

This section shows how the designed system might be implemented. (The reader may wish to review the Java programming refresher in Appendix A before proceeding.) One thing that programmers often neglect is that the code must be elegant and readable. This is not for the sake of the computer which will run the code, but for the sake of humans who will read, maintain, and improve on the original code. I believe that writing good comments is at least as difficult as writing good code. It may be even more important, because comments describe the developer's *intention*, while the code expresses only what the developer did. The code that lacks aesthetics and features poor writing style in comments is likely to be a poor quality code.[11] In addition to comments, languages such as Java and C# provide special syntax for writing the documentation for classes and methods. Javadoc is a tool for generating API documentation in HTML format from documentation comments in source code. Sandcastle is the equivalent tool for C#.

The hardware architecture of our system-to-be is described in Section [@@@] (Figure 2-7).

The following code uses threads for concurrent program execution. The reader not familiar with threads should consult Section 5.3.

The key purpose of the main class is to get hold of the external information: the table of valid keys and a connection to the embedded processor that controls the devices. Following is an implementation for the main system class.

Listing 2-3: Implementation Java code of the main class, called `HomeAccessControlSystem`, of the case-study home-access system.

```
import java.io.IOException;
import java.io.InputStream;
import java.util.TooManyListenersException;
import javax.comm.CommPortIdentifier;
import javax.comm.NoSuchPortException;
```

[11] On a related note, writing user messages is as important. The reader may find that the following funny story is applicable to software products way beyond Microsoft's: "*There was once a young man who wanted to become a great writer and to write stuff that millions of people would read and react to on an emotional level, cry, howl in pain and anger, so now he works for Microsoft, writing error messages.*" [Source: A Prairie Home Companion, February 3, 2007. Online at: http://prairiehome.publicradio.org/programs/2007/02/03/scripts/showjokes.shtml]

```
import javax.comm.SerialPort;
import javax.comm.SerialPortEvent;
import javax.comm.SerialPortEventListener;
import javax.comm.UnsupportedCommOperationException;

public class HomeAccessControlSystem extends Thread
        implements SerialPortEventListener {
    protected Controller ctrler_;  // entry point to the domain logic
    protected InputStream inputStream_;  // from the serial port
    protected StringBuffer key_ = new StringBuffer(); // user key code
    public static final long keyCodeLen_ = 4;  // key code of 4 chars

    public HomeAccessControlSystem(
        KeyStorage ks, SerialPort ctrlPort
    ) {
        try {
            inputStream_ = ctrlPort.getInputStream();
        } catch (IOException e) { e.printStackTrace(); }

        LockCtrl lkc = new LockCtrl(ctrlPort);
        LightCtrl lic = new LightCtrl(ctrlPort);
        PhotoObsrv sns = new PhotoObsrv(ctrlPort);
        AlarmCtrl ac = new AlarmCtrl(ctrlPort);

        ctrler_ =
            new Controller(new KeyChecker(ks), lkc, lic, sns, ac);

        try {
            ctrlPort.addEventListener(this);
        } catch (TooManyListenersException e) {
            e.printStackTrace(); // limited to one listener per port
        }
        start(); // start the thread
    }

    /** The first argument is the handle (filename, IP address, ...)
     * of the database of valid keys.
     * The second arg is optional and, if present, names
     * the serial port. */
    public static void main(String[] args) {
        KeyStorage ks = new KeyStorage(args[1]);

        SerialPort ctrlPort;
        String portName = "COM1";
        if (args.length > 1) portName = args[1];
        try {      // initialize
            CommPortIdentifier cpi =
                CommPortIdentifier.getPortIdentifier(portName);
            if (cpi.getPortType() == CommPortIdentifier.PORT_SERIAL) {
                ctrlPort = (SerialPort) cpi.open();

                // start the thread for reading from serial port
                new HomeAccessControlSystem(ks, ctrlPort);
        } catch (NoSuchPortException e) {
            System.err.println("Usage: ... ... port_name");
        }
```

```
        try {
            ctrlPort.setSerialPortParams(
                9600, SerialPort.DATABITS_8, SerialPort.STOPBITS_1,
                SerialPort.PARITY_NONE
            );
        } catch (UnsupportedCommOperationException e) {
            e.printStackTrace();
        }
    }

    /** Thread method; does nothing, just waits to be interrupted
     * by input from the serial port. */
    public void run() {
        while (true) { // alternate between sleep/awake periods
            try { Thread.sleep(100); }
            catch (InterruptedException e) { /* do nothing */ }
        }
    }

    /** Serial port event handler
     * Assume that the characters are sent one by one, as typed in. */
    public void serialEvent(SerialPortEvent evt) {
        if (evt.getEventType() == SerialPortEvent.DATA_AVAILABLE) {
            byte[] readBuffer = new byte[5]; // 5 chars, just in case

            try {
                while (inputStream_.available() > 0) {
                    int numBytes = inputStream_.read(readBuffer);
                    // could check if "numBytes" == 1 ...
                }
            } catch (IOException e) { e.printStackTrace(); }
            // append the new char to the user key
            key_.append(new String(readBuffer));

            if (key_.length() >= keyCodeLen_) {  // got the whole key?
                // pass on to the Controller
                ctrler_.enterKey(key_.toString());
                // get a fresh buffer for a new user key
                key_ = new StringBuffer();
            }
        }
    }
}
```

The class `HomeAccessControlSystem` is a thread that runs forever and accepts the input from the serial port. This is necessary to keep the program alive, because the main thread just sets up everything and then terminates, while the new thread continues to live. Threads are described in Section 5.3.

Next shown is an example implementation of the core system, as designed in Figure 2-33. The coding of the system is directly driven by the interaction diagrams.

Listing 2-4: Implementation Java code of the classes `Controller`, `KeyChecker`, and

LockCtrl.

```
public class Controller {
    protected KeyChecker checker_;
    protected LockCtrl lockCtrl_;
    protected LightCtrl lightCtrl_;
    protected PhotoObsrv sensor_;
    protected AlarmCtrl alarmCtrl_;
    public static final long maxNumOfAttempts_ = 3;
    public static final long attemptPeriod_ = 600000; // msec [=10min]
    protected long numOfAttempts_ = 0;

    public Controller(
        KeyChecker kc, LockCtrl lkc, LightCtrl lic,
        PhotoObsrv sns, AlarmCtrl ac
    ) {
        checker_ = kc;
        lockCtrl_ = lkc; alarmCtrl_ = ac;
        lightCtrl_ = lic; sensor_ = sns;
    }

    public enterKey(String key_code) {
        Key user_key = new Key(key_code)
        if (checker_.checkKey(user_key)) {
            lockCtrl_.setArmed(false);
            if (!sensor_.isDaylight()) { lightCtrl_.setLit(true); }
            numOfAttempts_ = 0;
        } else {
            // we need to check the attempt period as well, but ...
            if (++numOfAttempts_ >= maxNumOfAttempts_) {
                alarmCtrl_.soundAlarm();
                numOfAttempts_ = 0; // reset for the next user
            }
        }
    }
}
```

```
import java.util.Iterator;

public class KeyChecker {
    protected KeyStorage validKeys_;

    public KeyChecker(KeyStorage ks) { validKeys_ = ks; }

    public boolean checkKey(Key user_key) {
        for (Iterator e = validKeys_.iterator(); e.hasNext(); ) {
            if (compare((Key)e.next(), user_key) { return true; }
        }
        return false;
    }

    protected boolean compare(Key key1, Key key2) {

    }
}
```

```
import javax.comm.SerialPort;
```

```
public class LockCtrl {
    protected boolean armed_ = true;

    public LockCtrl(SerialPort ctrlPort) {
    }
}
```

In Listing 2-4 I assume that KeyStorage is implemented as a list, `java.util.ArrayList`. If the keys are simple objects, e.g., numbers, then another option is to use a hash table, `java.util.HashMap`. Given a key, KeyStorage returns a value of a valid key. If the return value is `null`, the key is invalid. The keys must be stored in a persistent storage, such as relational database or a plain file and loaded into the KeyStorage at the system startup time, which is not shown in Listing 2-4.

The reader who followed carefully the stepwise progression from the requirements from the code may observe that, regardless of the programming language, the code contains many details that usually obscure the high-level design choices and abstractions. Due to the need for being precise about every detail and unavoidable language-specific idiosyncrasies, it is difficult to understand and reason about software structure from code only. I hope that at this point the reader appreciates the usefulness of traceable stepwise progression and diagrammatic representations.

2.7.6 Refactoring: Improving the Design of Existing Code

A **refactoring** of existing code is a transformation that improves its design while preserving its behavior. Refactoring changes the internal structure of software to make it easier to understand and cheaper to modify that does not change its observable behavior. The process of refactoring involves removing duplication, simplifying complex logic, and clarifying unclear code. Examples of refactoring include small changes, such as changing a variable name, as well as large changes, such as unifying two class hierarchies.

Refactoring applies sequences of low-level design transformations to the code. Each transformation improves the code by a small increment, in a simple way, by consolidating ideas, removing redundancies, and clarifying ambiguities. A major improvement is achieved gradually, step by step. The emphasis is on tiny refinements, because they are easy to understand and track, and each refinement produces a narrowly focused change in the code. Because only small and localized block of the code is affected, it is less likely that a refinement will introduce defects.

Agile methods recommend test-driven development (TDD) and continuous refactoring. They go together because refactoring (changing the code) requires testing to ensure that no damage was done.

Using Polymorphism Instead of Conditional Logic

An important feature of programming languages is the *conditional*. This is a statement that causes another statement to execute only if a particular condition is true. One can use simple "sentences" to advise the computer, "Do these fifteen things one after the other; if by then you still haven't achieved such-and-such, start all over again at Step 5." Equally, one can readily symbolize a complex conditional command such as: "If at that particular point of runtime, *this* happens, then do so-and-so; but if *that* happens, then do such-and-such; if anything else happens, whatever it is, then do thus-and-so." Using the language constructs such as IF-THEN-ELSE, DO-WHILE, or SWITCH, the occasion for action is precisely specified. The problem with conditionals is that they make code difficult to understand and prone to errors.

Polymorphism allows avoiding explicit conditionals when you have objects whose behavior varies depending on their types. As a result you find that `switch` statements that switch on type codes or if-then-else statements that switch on type strings are much less common in an object-oriented program. Polymorphism gives you many advantages. The biggest gain occurs when this same set of conditions appears in many places in the program. If you want to add a new type, you have to find and update all the conditionals. But with subclasses you just create a new subclass and provide the appropriate methods. Clients of the class do not need to know about the subclasses, which reduces the dependencies in your system and makes it easier to update.

some conditionals are needed, like checks for boundary conditions, but when you keep working with similar variables, but apply different operations to them based on condition, that is the perfect place for polymorphism and reducing the code complexity. Now there are usually two types of conditionals you can't replace with Polymorphism. Those are comparatives (>, <) (or working with primitives, usually), and boundary cases, sometimes. And those two are language

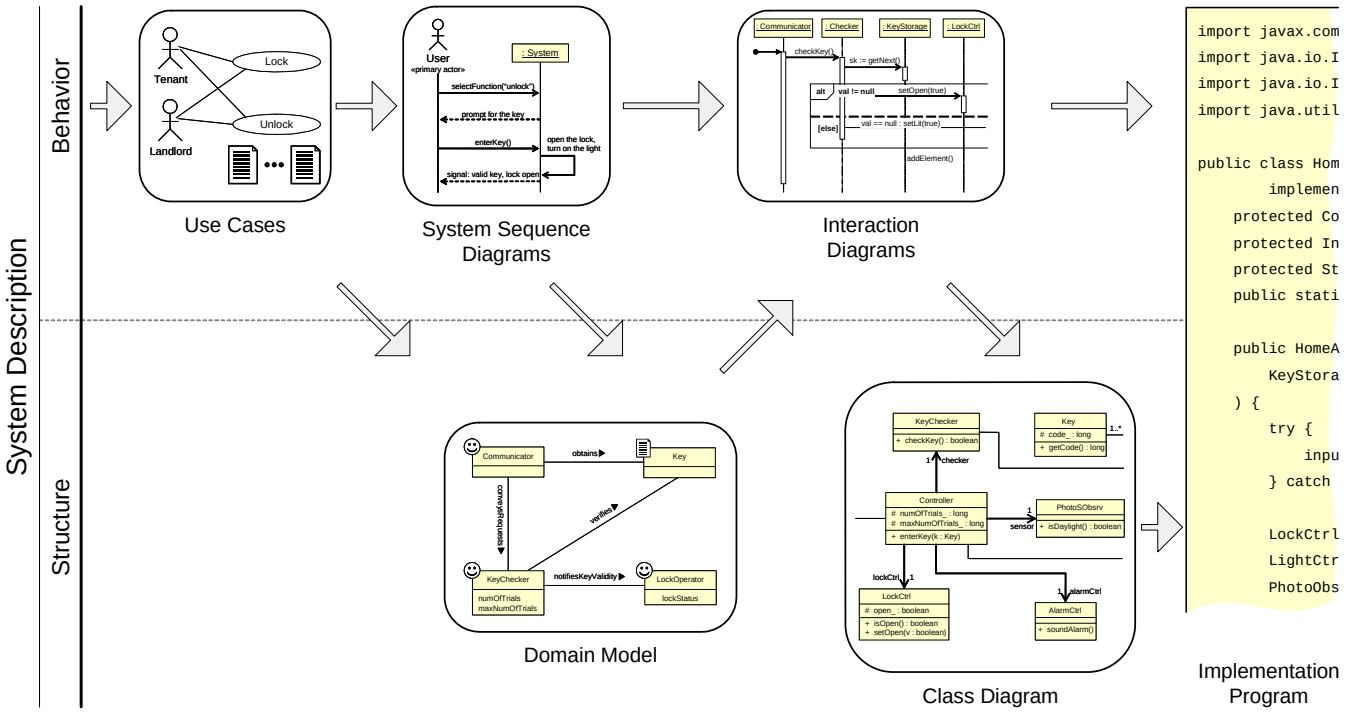


Figure 2-44: Summary of a single iteration of the software development lifecycle. The activity alternates between elaborating the system's behavior vs. structure. Only selected steps and artifacts are shown.

specific as well, as in Java only. Some other languages allow you to pass closures around, which obfuscate the need for conditionals.

2.8 Summary and Bibliographical Notes

"Good judgment comes from experience, and experience comes from bad judgment."
—Frederick P. Brooks

This chapter presents incremental and iterative approach to software design and gradually introduces software engineering techniques using a running case study. Key phases of the process are summarized in Figure 2-44. (Note that *package diagram*, which is a structural description, is not shown for the lack of space.) To ensure meaningful correspondence between the successive software artifacts, we maintain traceability matrices across the development lifecycle. The traceability matrix links requirements, design specifications, hazards, and validation. Traceability among these activities and documents is essential.

Figure 2-44 shows only the logical order in which activities take place and does not imply that software lifecycle should progress in one direction as in the waterfall method. In practice there is significant intertwining and backtracking between the steps and Figure 2-44 shows only *one iteration* of the process. The sequential presentation of the material does not imply how the actual