Chapter 5

Working with Code

“A designer knows [s]he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away.”
—Antoine de Saint-Exupery

Effective software development requires a mastery of both the architecture and specific strategies for improving, testing, and debugging the code base.

This chapter introduces these strategies, which can be used either for correcting a defect or adding a new feature to the software. Both these activities require the use of refactoring, testing, and debugging tools and techniques.¹

When we work with code, we use both synthesis (writing new code) and analysis (criticizing and revising existing code). Four central activities are associated with code analysis:

- Identifying “bad smells,” or poorly written code fragments
- Refactoring
- Testing
- Debugging

These are strongly interrelated activities in the software development process (see Figure 5.1). One cannot usually occur in isolation.

The following sections present and illustrate some basic techniques associated with each of these activities.

¹To facilitate hands-on engagement with refactoring, testing, and debugging, readers are encouraged to download and install a sandbox version of their project’s code base. The RMH Homebase release 1.5 code base is used in the examples in this chapter, and it can be downloaded from the book’s Web site myopensoftware.org/textbook. A sandbox version of a code base is an implementation where developers can “play” with the code base and the run-time system in order to understand it better. Since developers are not using the live version of the software, errors that they make while playing in the sandbox have no disastrous side effects on users.
5.1 Bad Smells and Metrics

Before adding new features to an existing code base, the code itself needs to be examined for quality and receptivity to those new features. Many times, such an examination reveals bugs that need to be fixed, or at least organizational characteristics that need to be improved, before the new features can be added.

This section discusses two strategies for finding code that needs to be improved: identifying bad smells and using software metrics.

5.1.1 Identifying Bad Smells

The code we read while testing or preparing to add a new feature to a software product may have many “bad smells” (a term coined by Kent Beck in [Fow00], p. 75). A bad smell is a segment of code that doesn’t read clearly and concisely, and hence probably can be improved.

Such code may have been developed by a novice, a person not familiar with standards for good programming practice, or someone interested only in “making the program work” rather than “making the program readable.” In any case, the original author of the smelly code is likely not within shouting distance. So it falls to the current developer to remove bad smells from the code so that future developers will have an easier time understanding it.

To illustrate this idea, consider the PHP program text in Figure 5.2, which contains several duplicate copies of a very technical piece of text. The removal of this bad smell by refactoring renders the code more readable, as illustrated in Figure 5.3.

Several different types of bad smells can occur in programs. Here are a few common types of bad smells that commonly occur in existing software artifacts (see Fowler [Fow00] for a more detailed discussion):

- Poorly Named Variable, Method, or Class—the variable, method, or class name does not clearly represent its purpose in the code. For example, the name $sch is a poor name for a variable instead of $schedule.
Duplicate Code—the same sequence of expressions or instructions appears in several places. For example, see Figure 5.2.

Long Method—a method contains a lot of code that accomplishes several sub-tasks. For example, writing a method `AvMaxMin($list)` is a poor way to compute the average, maximum, and minimum value in a list, compared with writing three separate methods.

Large Class—a class has an unusually large and diverse collection of instance variables or methods. Often this signals a lack of cohesion for a class, in the sense that it is trying to represent more than one object at one time. For example, the class `AlarmClock` might have features of an alarm and other features of a clock. A better design would define two classes, `Alarm` and `Clock`, and specify that a `Clock` may have an `Alarm` as a component.

Too Many/Too Few Comments—too many comments can hide bad code (i.e., they can be “used as a deodorant”), while too few can make code difficult to read. As a rule of thumb, use the guidance discussed in Section 2.1.3 for inserting an appropriate level of commentary in the code base.

Data Clumps—the same three or four variables appear together in several different places. This may signal an opportunity for defining, or “extracting,” a new class that has these variables as instance variables.

Parallel Inheritance Hierarchies—each time a sub-class is added to one hierarchy, it must be added to other hierarchies. This may signal a need to reorganize the hierarchies so that the sub-class is added only once.

Feature Envy—a method requires lots of information from a different class. Perhaps that method should be embedded inside that class.

```c
/**
 * process_form sanitizes data, concatenates needed data, and enters it all into a database
 */
function process_form(){
  //step one: sanitize data by replacing HTML entities and escaping the ' character
  $first_name = trim(htmlspecialchars(str_replace("\", \\\", $POST['first_name'])));
  $last_name = trim(htmlspecialchars(str_replace("\", \\\", $POST['last_name'])));
  $address = trim(htmlspecialchars(str_replace("\", \\\", $POST['address'])));
  $city = trim(htmlspecialchars(str_replace("\", \\\", $POST['city'])));
  $state = trim(htmlspecialchars(str_replace("\", \\\", $POST['state'])));
  $zip = trim(htmlspecialchars(str_replace("\", \\\", $POST['zip'])));
  $phone1 = trim(htmlspecialchars(str_replace("\", \\\", $POST['phone1'])));
  $phone2 = trim(htmlspecialchars(str_replace("\", \\\", $POST['phone2'])));
  $private_notes = trim(htmlspecialchars($POST['private_notes']));
  $public_notes = trim(htmlspecialchars($POST['public_notes']));
  $my_notes = trim(htmlspecialchars($POST['my_notes']));
}
```

FIGURE 5.2: Example bad smell—duplicate code.
Software Development: An Open Source Approach

- **Primitiv e Obsession**—the code is reluctant to use classes instead of primitive data types. This signals an opportunity to simplify the code and improve its reliability by using classes rather than primitives.

- **Lazy Class**—a class that no longer “pays its way”; it is seldom used and its methods are seldom called. This usually suggests replacing those calls by calls to methods in other classes, and then removing the class from the code base.

- **Speculative Generality**—inserting features into the code for functionality that is not part of the current requirements. Generally, these clutter the code and should be removed.

- **Temporary Fields**—instance variables in a class that are set only in certain circumstances. All instance variables should be set and accessible all the time.

- **Inappropriate Intimacy**—pairs of classes that know too much about each other’s private details. This usually suggests making each class’s instance variables private and adding appropriate methods to access or modify them as needed by the other class in the pair.

- **Incomplete Class**—one that doesn’t do everything you need. The fix is to complete the class’s functionality by adding methods that do everything that is needed.

- **Refused Bequest**—a sub-class ignores most of the functionality provided by its superclass. The fix is to remove methods from the sub-class that duplicate that functionality.

### 5.1.2 Software Metrics

Another way to evaluate the quality of a code base is to calculate its so-called “software metrics.” A metric is a quantification of some aspect of the syntactic structure of a program, which aims to expose potential weaknesses (bugs, or risks of run-time failures) in that structure.

Several different metrics can be used to measure the quality of a program. If the metrics of a code base are not all within a normal range, this may indicate poor code quality. Here is a short list of metrics that can help measure the quality of a Java program:

- **Count Metrics**—the number of packages, classes, instance variables and methods for each class, and parameters and lines of code for each method. Good design tries to minimize these numbers in accordance with the nature of the application and its individual classes.

- **Clarity Metric**—“McCabe cyclometric complexity” counts the number of distinct paths through each method, considering all its if, for, while,
do, case, and catch statements. A good design tries to keep this number under 10 for every method in the code base.

- Cohesion Metric—lack of cohesion of methods (LCOM) in a class with \( m \) methods means that the ratio of the average number of methods accessing each instance variable to \( m - 1 \) is near 1. That is, a class is completely cohesive if all its methods access all its instance variables, which means that \( LCOM = 0 \). So the more cohesive the class, the closer \( LCOM \) is to 0, which is another indicator of good design.

- Coupling Metrics—indicate how strongly the packages in a code base are interdependent. Good design tries to minimize coupling.

Metrics calculators are available as a plugin for any Eclipse/Java IDE.\(^2\) When metrics are enabled, they are recalculated automatically whenever the code base is changed. Unfortunately, no similar plugin for Eclipse/PHP is available at this writing.

Bad smells and metrics tend to be complementary, in the sense that each metric corresponds to a particular bad smell that can be detected by examining the code.

The main advantage of using metrics over looking for bad smells is that they can be quickly applied to a very large code base, highlighting areas that need a closer look. That is, reading all the text of a million-line code base, without the aid of metrics, would be a very tedious exercise.

5.2 Refactoring

Once a bad smell is detected, or a metric reveals a probable instance of poor coding, the offending code segment needs to be improved. Modifying code to improve its readability, efficiency, or modifiability is called refactoring.

No new features should be added during refactoring: the goal of refactoring is simply to improve the quality of the code without changing its observable behavior (i.e., its functionality in the eyes of the user). Often refactoring substantially reduces the size of the code base. Refactoring may also transform complex structures into simpler ones that are easier to maintain and understand.

What particular types of refactoring can be done? One kind of refactoring is to eliminate instances of duplicate code, which usually involves defining a new function with one copy of that code and replacing all the duplicate copies.

---

\(^2\)The use of Eclipse/Java metrics is illustrated at the book's Web site myopensoftware.org/textbook.
FIGURE 5.3: Example bad smell removal.

by a function call. The bigger the size of the duplicated code, the worse it smells! Figure 5.3 shows how creating a new function can eliminate the bad smell of duplicate code that appears in Figure 5.2.

This type of refactoring is called “method extraction.” Many other types of refactoring are also possible.\(^3\) To support this activity, the integrated development environment (IDE) can be configured to perform some types of refactoring automatically.

Here is a short list of important refactoring types that can be useful for developers working with a code base as they begin to add new features or track down bugs:

**Renaming** a variable, method, or class to add clarity for the reader and consistency with the user’s domain and design document.

**Extracting a method** to eliminate duplicate code, and then replace each duplicate appearance by a call to that method.

**Extracting methods** to reduce the size of a long method. That is, logically divide the long method into smaller segments, extracting each segment as a separate method, and then replace the original segments by a series of calls.

**Reorganizing a class** to improve its logical cohesion. If two or more kinds of objects are encapsulated by the class, break it into two or more separate classes.

**Unifying data clumps** If a clump of data declarations characterizes an object, define a new class with them as its instance variables. Each original

\(^3\)In fact, entire books have been written on this subject (for example, [Fow00]).
appearance of these variables can be replaced by a call to a new method, which can be defined within the new class.

**Removing a parameter** Sometimes a parameter is superfluous, for example when it is functionally dependent on one or more other parameters. Such a parameter can be eliminated.

**Simplifying conditionals** Sometimes a collection of conditionals can become so deeply nested that the underlying logic becomes impossible to understand. Such a nest can be simplified by rethinking and disentangling the logic.

The following additional types of refactoring rely on the principles of software architecture—layers, cohesion, and coupling—that were discussed in Chapter 4.

**Removing Useless Code** When reviewing the code base alongside the design document, look for code that has no purpose in relation to the existing requirements or the rest of the code base. This may take the form of a function, variable, or an entire class or module that is never referenced. All such code should be removed.

**Removing Violations to Layering Principles** When layer, cohesion, and coupling principles are violated, refactoring should be done to remove the violations. This often involves other refactorings, such as extracting a new method or function.

**Merging Similar Functions/Modules** During different stages in development, a developer may not be aware that a function/module already exists that (nearly) fulfills a certain requirement, and thus may reimplement that function/module. When this redundancy is discovered later, it becomes a candidate for refactoring.

**Separating Model, View, and Controller Code** In a graphical user interface (GUI) module, confusion can occur if the model, view, and controller code are intertwined. Separating the code for these three may help clarify the code and make it more robust. We shall discuss this activity in more detail in Chapter 8.

Note that some refactoring activities may combine two or more of these techniques in a single step. For example, when combining two similar methods into one, we may also eliminate or add a parameter to clarify the new code. Note finally that many of these refactorings tend to reduce the size of the code base rather than enlarge it. In general, a good architecture is often one

---

4A modern IDE, such as Eclipse, will automatically flag instances of unused code sections or unreferenced variables, so that they can be easily spotted and removed.
that has less code rather than more. Effective software, like effective writing, has little tolerance for redundancy and superfluity. Remember the words of Saint-Exupery!

The refactoring examples in the following sections are all taken from the RMH Homebase release 1.5 code base.

5.2.1 Example 1: Removing Useless Functions

A function is useless if it is not called from anywhere else in the code base. Identifying useless functions is relatively easy with a modern IDE. One way to do this is to highlight the function name and then search the code base (project) for all references to that name. If there are none, the function is useless.

For example, consider the Shift class in RMH Homebase. It has 24 functions and constructors. Utilizing the search tools provided in our IDE, we find that the following functions are not called from anywhere else in the code base, and therefore can be safely eliminated:

```
fill_vacancy
find_person
remove_person
```

The reason these can be removed is that the responsibilities for adding and removing individual persons in a shift are handled directly by the dbShifts module. The only services the dbShifts.php module requires from the Shift module are adding and removing all the persons from a shift at once, using the get_persons and assign_persons functions.

Why were these three functions included in the Shift class in the first place? Probably because the Shift class was developed before, and independently from, the development of the dbShifts database module and the editShift user interface module. Now is a good time to clean house and remove them; in preparing to add new features to the code base, we want it to be as lean as possible at the outset.

5.2.2 Example 2: Removing a Layering Violation

In Chapter 4, we introduced an example in the editShift module from RMH Homebase that violates the layering principle (recall Figure 4.4). We return to that example now and illustrate how we can remove that violation by extracting a new method.

The code shown in Figure 4.4 builds an array of all persons who have/do not have “family_room” in their type and are available on a certain $day and $time. This code, which occurs in a user interface module, is directly executing a database query rather than calling a function in a database module, thus violating the layering principle.
Looking further into the editShift.php module, we see a second layering violation of the same sort:

```php
connect();
$query="SELECT * FROM dbPersons WHERE " .
"(type LIKE '%volunteer%' OR type LIKE '%sub%') " .
"ORDER BY last_name,first_name";
$result=mysql_query($query);
mysql_close();
```

This code builds an array of all persons who have “volunteer” or “sub” in their type, but for no particular day or time. Both sections return the array ordered by last_name and first_name.

It would be good to try to remove both these violations at once. To do this, we need to find or define one or more functions in the dbPersons module that satisfy both needs and then replace these two sections of code by appropriate function calls.

The dbPersons module already has a few search functions predefined. These are:

- `getall_persons()` : Find all persons.
- `getall_type($t)` : Find all persons of type $t.
- `getall_available($t, $day, $time)` : Find all persons of type $t who are available on $day and $time.
- `getall_availables($t, $u, $day, $time)` : Find all persons of type $t and $u who are available on $day and $time.
- `getall_scheduled($t, $group, $day, $time)` : Find all persons of type $t who are scheduled in $group, $day, and $time.

A quick look at the utility of these search functions reveals that only the first two are called from anywhere else in the code base. The other three can thus be removed. In their place, we can add two new search functions to the dbPersons module that address our newfound refactoring needs. These new functions are shown in Figures 5.4 and 5.5.

Now returning to the code in editShift.php, we can replace the first violation of the layering principle by the following call:

```php
$result = getall_available($fam, $day, $time);
```

and the second violation by the following call:

```php
$result = getall_eithertype("volunteer", "sub");
```

Now these layering violations have been eliminated from the editShift module and two new useful search functions have replaced three useless ones in the dbPersons module. Since either call returns $result === false if the search returns nothing, the remaining code in the editShift module is unaffected.
function getallAvailable($venue, $day, $time) {
    connect();
    if ($venue == "Fam")
        $query="SELECT * FROM dbPersons
                    WHERE (type LIKE "%family_room%")
                       " AND availability LIKE "%" . $day . $time . "%" 
                       ORDER BY last_name,first_name";
    else
        $query="SELECT * FROM dbPersons
                    WHERE (NOT (type LIKE "%family_room%"))
                       " AND availability LIKE "%" . $day . $time . "%" 
                       ORDER BY last_name,first_name";
    $result=mysql_query($query);
    mysql_close();
    return $result;
}

/*
 * get all available volunteers with either type $t or type $u
 * return false if none available
 */
function getall_eithertype ($t, $u) {
    connect();
    $query="SELECT * FROM dbPersons WHERE (type LIKE "%" .
        $t . "%" OR type LIKE "%" . $u . 
        "%") ORDER BY last_name,first_name";
    $result=mysql_query($query);
    mysql_close();
    return $result;
}

FIGURE 5.4: A new search function for the dbPersons module.

FIGURE 5.5: A second new search function for the dbPersons module.
5.3 Testing

Developing and using an effective test suite is a key step in software development. Traditional views place this activity sometime after the coding is done. That is the wrong placement.

A more aggressive approach to developing a test suite is to do it hand-in-hand with the coding process. A most aggressive approach to developing a test suite is to do it hand-in-hand with requirements gathering, and before any coding is done. This approach is advocated by the agile philosophy, and it is called test-driven development or TDD.

TDD is especially valuable because it keeps the client in the game, so to speak. That is, discussions between developers and clients can evoke examples of system and user interface behavior before the system is actually built. These examples thus become tests that drive the writing of the initial code for the new software.

When a test suite is developed either during or before the coding process, the test cases can motivate effective and focused code development. The code is clearly a prototype at this stage, since it is designed to respond specifically to the tests at hand. However, once an iteration or two of this agile process has taken place, the code (and the tests) will begin to mature and become more generally applicable.

A good development environment, such as Eclipse, supports this parallel process of test suite development alongside coding. Unit testing tools, like PHPUnit for PHP or JUnit for Java, can be easily integrated within the Eclipse environment.

The organization of the code base should also support unit testing, in the sense that a separate directory of test cases should live alongside the code directories themselves. Keeping the test modules together with the code base is good practice, even though it adds some storage overhead to the system as a whole. This practice facilitates unit testing of existing code whenever refactoring is done or new functionality is added.

For example, RMH Homebase has a collection of unit tests in the directory tests. Each test is aligned with a particular class or database module in the system. This directory has no unit tests for the user interface modules. Instead, each of those modules has been tested by exercising all the paths through the forms using a hypothetical database of users and a master schedule.

Each unit test usually applies to a single class or module. For example, Figure 5.6 shows the methods to be tested that are defined inside the Shift class in RMH Homebase.

The unit test itself contains a series of assert statements that must all be true for the code to “pass” the test. Each assert statement typically exercises
A unit test should have at least one method call for each non-trivial method in the module or class. For example, Figure 5.7 shows some elements of the unit test that were written to exercise the methods inside the Shift class shown in Figure 5.6.

```
class testShift extends UnitTestCase {
    function testShiftModule() {
        $noonshift = generate_new_shift("03-28-08-12-3",3);
        $this->assertEqual($noonshift->get_name(), "12-3");
        $this->assertTrue($noonshift->get_id() == "03-28-08-12-3");
        $this->assertTrue($noonshift->num_vacancies() == 3);
        $this->assertTrue($noonshift->get_day() == "Fri");
        $this->assertFalse($noonshift->has_sub_call_list());
        ...
```

FIGURE 5.7: Elements of a unit test for a PHP class or module.
class AllTests extends GroupTest {
    function AllTests() {
        ...
        $this->addTestFile(dirname(__FILE__).'/testShift.php');
        ...
        $this->addTestFile(dirname(__FILE__).'/testdbShifts.php');
        ...
        $this->addTestFile(dirname(__FILE__).'/testeditShift.php');
        ...
        echo("\nAll tests complete");
    }
}

FIGURE 5.8: A series of unit tests.

FIGURE 5.9: Results of running a series of unit tests.

5.3.1 Unit Testing Tools

Software tools for unit testing provide a convenient way to package a sequence of unit tests and run them all at once, relieving the developer of the effort that would be required to run each unit test individually.

In PHP, for example, one such tool is called SimpleTest. A series of unit tests for RMH Homebase code base is shown in Figure 5.8. Here, the ellipses (...) are placeholders for tests developed for the remaining classes, modules, and use cases indicated in the above list.

The results of running the three tests discussed in the foregoing section are displayed in Figure 5.9. If this figure were in color, the horizontal bar in the middle would be green when all tests had run with no failures or errors. A blue bar would indicate that one of the assert statements had failed, while a red bar would indicate that a run-time error had occurred in the code itself. So this feedback provides the developer with quick visual information about the status of the code base after exercising it with a complete test suite.

As a final note on testing, it is important to acknowledge its fallibility. That is, systematic unit testing does not guarantee that the code is free of errors. Only a strategy that uses formal methods would guarantee that.\footnote{The study of formal methods is beyond the scope of the present text. For more information, see [reference].}
5.3.2 Test Case Design

A well-architected software system must be hospitable to a rigorous testing strategy. To facilitate development, we need to design a collection, or suite, of test cases that cover all the modules and functions in the system. A Multi-Tier software architecture can provide a clear framework for developing such a test suite.

If the software we are writing is an extension of an existing system, the existing code base should ideally contain a complete suite of good unit tests. If it does not, our first step is to complete the test suite for the existing system—perhaps not a simple task in itself. Thus, when we build the test suite for the new system, it will be anchored in a reliable initial test suite.

It may be a surprise to suggest that a test suite for adding new features to a code base can be developed before any new code is written to implement those features. This process is formally known as test-driven development (TDD). The motivation for this horse-before-the-cart approach is twofold:

1. New code is better focused when it responds to specific use-case-driven examples in the form of test cases.

2. Users can suggest the best test cases because they have the clearest idea of the new application’s use cases.

The pre-existence of a test suite provides programmers with “live” goals for the code to achieve as they write, along with a suite of realistic examples that can test the validity of the new code itself. How do we begin designing a useful test suite for a code base? Here is the beginning of a discussion of this very complex question\(^6\):

A test suite is a collection of “unit tests.” The suite should have one unit test for every module or class in the code base and one unit test for each use case.

Each unit test should contain a group of calls that exercise all of the module’s functions (and constructors, in the case of a class). For each such call, the unit test contains an assertion that delivers “true” exactly when that call is successful, and “false” otherwise.

A unit test should aim for 100% code coverage and 100% use case coverage. That is, every line of code in each module being tested

\(^6\) For a more thorough treatment of software testing, readers may explore the subject more deeply by starting at [en.wikipedia.org/wiki/Software_testing](http://en.wikipedia.org/wiki/Software_testing). The discussion on these pages identifies only the beginning stages in designing unit tests, and should not be considered a complete treatment of software testing.
Working with Code

should be exercised by at least one call in the unit test. Moreover, every step of each use case being tested should also be covered.\footnote{In fact, aiming for 100\% code coverage is often a weak testing strategy. For example, there may be boundary conditions occurring in the use case that are not covered by the code or identified in the use case itself.}

Designing a unit test for a domain class or database module begins by writing at least one call for every function and constructor in that class or module.

Designing a unit test for a user interface module is more difficult, since each user interface module contains a mix of PHP and HTML code that supports a particular user form. Nevertheless, the unit test must begin with a test for every user option that appears on the form. Usually, these options appear as buttons, menus, text boxes, or other widgets common to an interactive Web page.

To unit test a use case, we begin by identifying as a “unit” those user interface elements—forms and associated modules—that combine to implement that use case. The question our unit test addresses is, “Does that unit correctly and fully support a user executing all the steps of that use case?”

In this sense, the complete set of unit tests stands as a basis for “integration testing” or “acceptance testing” the system in the eyes of the user. Implicit in this definition is the assumption that the unit testing of the use cases follows the separate testing of all the underlying domain classes, database modules, and user interface forms individually.

Examining the code base for \textit{RMH Homebase} release 1.5, we see that it has a test suite (the \texttt{tests} directory), and the test suite has unit tests for many of the classes, modules, and use cases implemented by the code base. To test the entire system, a complete collection of unit tests would need to be designed and exercised.

Since \textit{RMH Homebase} has at least 15 classes and modules (see Figure 4.3) and 9 use cases, a complete test suite should contain at least 24 unit tests. In the sections below, we use this code base to illustrate the design of unit tests for a domain class, a database module, a user interface module, and a use case.

5.3.2.1 Example 1: Testing a Domain Class

As our first unit testing example, consider the design of a unit test for the \texttt{Shift} class. A complete unit test for this class should exercise every function and constructor in the class. A partial unit test for this class is shown in Figure 5.10.

The \texttt{Shift} class (summarized in Figure 5.6) has 20 different functions (17 after refactoring), each of which should be invoked during the unit test to be sure that it delivers the correct results. That is the purpose of the \texttt{assertTrue} statements in Figure 5.10, each of which succeeds if and only if its argument...
class testShift extends UnitTestCase {
    function testShiftModule() {
        $noonshift = generate_new_shift("03-28-08-12-3",3);
        $this->assertEqual($noonshift->get_name(), "12-3");
        $this->assertTrue($noonshift->get_id() == "03-28-08-12-3");
        $this->assertTrue($noonshift->num_vacancies() == 3);
        $this->assertTrue($noonshift->get_day() == "Fri");
        $this->assertFalse($noonshift->has_sub_call_list());
        $persons = array();
        $persons[] = "alex1234567890+alex+jones";
        $noonshift->assign_persons($persons);
        $noonshift->ignore_vacancy();
        $persons[] = "malcom1234567890+malcom+jones";
        $noonshift->assign_persons($persons);
        $noonshift->ignore_vacancy();
        $persons[] = "nat1234567890+nat+jones";
        $noonshift->assign_persons($persons);
        $noonshift->ignore_vacancy();
        $this->assertTrue($noonshift->num_vacancies() == 0);
        $noonshift->add_vacancy();
        $this->assertTrue($noonshift->num_slots() == 4);
        $noonshift->ignore_vacancy();
        $this->assertTrue($noonshift->num_slots() == 3);
        $noonshift->set_notes("Hello 12-3 shift!");
        $this->assertTrue($noonshift->get_notes() == "Hello 12-3 shift!");
    echo ("testShift complete");
    }
}

FIGURE 5.10: A partial unit test for the Shift class.
delivers a true result. Altogether, this partial unit test contains calls to 10 different functions in the Shift class.

Running a unit test can produce one of three outcomes:

**Success** All the assertions in the unit test are true.

**Failure** One or more of the assertions in the unit test is false.

**Error** An error occurred in the code base being tested.

A unit test succeeds, of course, only when the Success outcome is achieved. In either of the other two cases, work remains to be done to understand why a particular test failed or what is wrong with the underlying code base.

5.3.2.2 Example 2: Testing a Database Module

Once the Shift class is fully tested, we can move on to design a unit test for the dbShifts module. The dbShifts module has five major functions:

**setup_dbShifts** creates an empty dbShifts table in the database.

**insert_dbShifts** inserts a new shift (row) into the dbShifts table.

**select_dbShifts** selects a unique shift (row) from the dbShifts table.

**delete_dbShifts** removes a shift (row) from the dbShifts table.

**update_dbShifts** replaces an existing shift in the dbShifts table by another with the same key.

This is a straightforward module, so its unit testing can contain one call for each of these five functions. A unit test for dbShifts is shown in Figure 5.11. It has calls to four of the five distinct functions in the module.

The reason we cannot test the dbShifts module until after the Shift class has been fully tested is because dbShifts uses some of the Shift class’s functions (generate_new_shift in particular) to enable the testing of the dbShifts functions. Knowing a priori that the former are reliable helps us pin down problems in the testing of the dbShifts functions themselves.

The reason we do not test the **setup_dbShifts** function is that this function clears the dbShifts table completely. A non-empty dbShifts table is needed for the testing of other modules, as we shall see below.

5.3.2.3 Example 3: Testing a User Interface Module

Having tested both the Shift and dbShifts modules, we can now design a unit test for the user interface module editShift, which supports the user options on the Shift form shown in Figure 5.12.

We can see that this form has several buttons: “Add Slot,” “Clear Entire Shift,” “Generate Sub Call List,” “Remove Person,” “Assign Volunteer,” “Ignore
class test_dbShifts extends UnitTestCase {
    function test_dbShiftModule() {
        $s1 = generate_new_shift("02-25-08-night", 3);
        $this->assertTrue(insert_dbShifts($s1));
        $s2 = generate_new_shift("02-25-08-3-6", 3);
        $this->assertTrue(insert_dbShifts($s2));
        $this->assertTrue(select_dbShifts("02-25-08-3-6") !== null);
        print_r(select_dbShifts("02-25-08-3-6"));

        $this->assertTrue(delete_dbShifts($s1));
        $s2 = generate_new_shift("02-25-08-3-6", 2);
        $this->assertTrue(update_dbShifts($s2));
        $this->assertTrue(delete_dbShifts($s2));
        echo ("test_dbShifts complete");
    }
}

FIGURE 5.11: A unit test for the dbShifts module.

FIGURE 5.12: The Shift form in the user interface.
Working with Code

Vacancy,” and “Back to Calendar.” Our unit tests must eventually cover each of these possible user actions separately. A unit test for this form is shown in Figure 5.13.

This unit test relies on the existence of a calendar week that includes the date October 30, 2009, from which the 3–6 shift is being retrieved. Without that assumption, we would need to create a new shift equivalent to the one that is retrieved by the line:

```php
$myshift = select_dbShifts("10-30-09-3-6");
```

This unit test has five parts, corresponding to five of the seven distinct buttons on the Shift form. It does not include a test for the “Generate Sub Call List” button or the “Back to Calendar” button because these are included in other unit tests.

Notice, after each of the first four parts of this unit test is completed, that the line

```php
$myshift = select_dbShifts("10-30-09-3-6");
```

is repeated. This is needed because the variable `$myshift` must be updated by the new value stored in the database for this shift by the previous assertion. Otherwise, this variable would not be up-to-date with its corresponding database entry.

5.3.2.4 Example 4: Testing a Use Case

As a final unit testing example, consider the testing of the entire use case ChangeACalendar, described in Figure A.11. This is a very different process from the individual unit tests described above. Here, we are examining the integrity of the entire system, not a single module.

To do this, we start by assuming that all the individual forms in the use case have been individually tested. In the case of ChangeACalendar, this includes the calendar form as well as the Shift form and the Sub Call List form.

Then we follow each individual step in the use case by exercising the user interface elements that support that step. In the case of ChangeACalendar, we start by editing a week’s calendar such as the one shown in Figure 5.14.

This form can initiate any one of the following user activities:

- Editing a shift
- Adding or changing notes for an individual shift
- Adding or changing manager notes for an individual day
- Adding or changing the guest chef for an individual day
class testeditShift extends UnitTestCase {
    function testeditShiftModule() {
        // get a shift from the database
        $myshift = select_dbShifts("10-30-09-3-6");
        print_r($myshift);

        // test generate sub call list/view sub call list button
        // test clear shift button
        $this->assertTrue(process_clear_shift(
            array('submit_clear_shift'=>true), $myshift, "") );
        // test assign volunteer button
        $myshift = select_dbShifts("10-30-09-3-6");
        $this->assertFalse(process_add_volunteer(
            array('submit_add_volunteer'=>true, 'all_vol'=>"0",
                'scheduled_vol'=>"rob2077291234+rob+jones"),$myshift,"" ));
        // test add slot button
        $myshift = select_dbShifts("10-30-09-3-6");
        $this->assertTrue(process_add_slot(
            array('submit_add_slot'=>true), $myshift, "" ));
        // test ignore vacancy button
        $myshift = select_dbShifts("10-30-09-3-6");
        $this->assertTrue(process_ignore_slot(
            array('submit_ignore_vacancy'=>true, 'all_vol'=>"0",
                'scheduled_vol'=>"jon2077291234+jon+jones"),$myshift,"" ));
        // test assign volunteer button
        $myshift = select_dbShifts("10-30-09-3-6");
        $this->assertFalse(process_add_volunteer(
            array('submit_add_volunteer'=>true, 'all_vol'=>"0",
                'scheduled_vol'=>"rob2077291234+rob+jones"),$myshift,"" ));
        // test remove person/create vacancy button
        $myshift = select_dbShifts("10-30-09-3-6");
        $this->assertTrue(process_unfill_shift(
            array('submit_filled_slot_0'=>true), $myshift, "" ));
        $myshift = select_dbShifts("10-30-09-3-6");
        print_r($myshift);
        echo("testeditShift complete");
    }
}

FIGURE 5.13: A unit test for the editShift module.
At this point, we assume that each of these activities has already been tested individually. Now we are interested in verifying that all the steps of the use case can be effectively carried out by a sequence of actions that may invoke several other forms in the process. In that sense, we are basically testing the integrity of the links that connect the individual forms together, as summarized below.

- Editing a shift—user selects the shift to be edited. The user should be transferred to the Shift form (see Figure 5.12) for that shift.

- Adding or changing notes for an individual shift—user edits the shift’s notes and then selects the “Save changes” button at the bottom of the form. The user should see the calendar refreshed with an acknowledgment that the changes have been made.

- Adding or changing manager notes for an individual day—user edits the manager notes line and then selects “Save changes.” The user should see the calendar refreshed with an acknowledgment that the changes have been made.

- Adding or changing the guest chef for an individual day—user edits the guest chef line and then selects “Save changes.” The user should see the calendar refreshed with an acknowledgment that the changes have been made.
The guide in this exercise is the use case itself. A “Success” is achieved when all the steps of the use case can be completed and the desired result (making a change on the calendar) can be achieved by the user. Otherwise, either a “Failure” will occur (when a different result from the one expected is observed) or an “Error” will occur (when the code base fails to execute for some particular step).

5.3.3 A Strategy for Sequencing Unit Tests

As the above examples suggest, there is a certain order in which the individual unit tests can be conducted. This order is determined by the functional dependencies that occur among the modules at the different layers in the architecture.

For instance, to find a sub, the user must begin by editing the Sub Call List form, which is managed by the subCallList.php module. Looking back at Figure 4.3, we see that this module requires the services of both the dbShifts.php module and the dbSCL.php module. These in turn require the services of the Shift.php and SCL.php classes, and so on.

So before we can begin to test the Sub Call List form, we must first unit-test all the modules from which it requires services. Then, if that unit test fails, we can assume that the error is most likely confined to the subCallList.php module, rather than a module upon which it is functionally dependent.

An overall strategy for designing and conducting a complete suite of unit tests for a Multi-Tier architecture can be developed in the following way:

1. Begin by testing all the classes/modules that do not receive services from any other classes or modules.

2. Next, test each class/module that only receives services from classes or modules that have already been tested.

3. Repeat step 2 until no more classes/modules remain to be tested.

4. Finally, test each use case.

Using this strategy for testing RMH Homebase, we should test the classes and modules in Figure 4.3 in a sequence that respects the dependencies among them:

1. Begin by testing the classes Person, Shift, and SCL.

2. Test the class RMHDate and the modules dbSCL, dbPersons, login_form, and index.

Notice that this strategy requires that there be no circular dependencies in the architecture. If there were, repetition of step 2 would not necessarily result in the testing of all classes/modules in the system.
3. Test the class Week and the module dbShifts.

   Test the modules dbDates and subCallList.

   Test the modules dbWeeks and editShift.

   Test the module calendar.

4. Test each of the nine use cases.

   By the time we test the use cases, this testing sequence gives us reasonable confidence that all the underlying classes and modules are reliable, since they themselves have already been unit tested.

5.4 Debugging

We can expect that, throughout the development and useful life of a software product, errors in the code will occur. Importantly, the users provide the first line of defense in locating software errors. Perhaps this is because the users exercise the code far more rigorously than any suite of tests. Users thus provide the most reliable and thorough source of feedback to developers, especially for the existence of errors, or bugs, in the software itself.

Technically, the term bug refers to a defect or flaw in the code base that contributes to the occurrence of a failure.

The term failure refers to any unacceptable behavior in the software that can be observed by the user.

In this book we use the term bug rather informally, in the sense that it will simultaneously refer to both a failure in the software and its related defect in the underlying code base.

A most severe kind of bug is one that causes the software to “crash,” or go into a permanent dysfunctional state from which the user cannot recover. One example of such a crash is the occurrence of a so-called “white screen of death” (WSOD, for short), where the user suddenly ends up staring at a blank screen. Little evidence of the source of the bug is apparent when this happens. So one debugging strategy for this case is to gather all the information that was available just before the WSOD happened (by restarting, repeating the steps, and gathering information along the way).

In any case, when finding and correcting a bug, the developer should add a new test to the test suite that addresses that particular error. Severe defects may even cause developers to refactor parts of the code base, perhaps even modifying the architecture itself. Thus, the interplay among software architecture, refactoring, test case design, and debugging is intimate.
5.4.1 Tool Use vs Developer Skill

Recall from Chapter 3 that some important tools are available for reporting, tracking, finding, and correcting software bugs. For example, the IDE contains debugging support for developers. The bug tracker provides support for reporting and updating the status of a bug in a community-oriented software project from its inception to its resolution. Yet the tools alone are often insufficient for effectively locating and removing a bug from a code base.

Earlier chapters illustrate that a healthy open source software project relies on the wide participation of users and developers to help keep the code base up to date and reasonably free of errors. FOSS developers thus rely on the active participation of users to help them test new releases and verify bug corrections in the code base.

Beyond tool use and collaboration with users, developers need two additional skills for effectively diagnosing and removing a bug from the code base: a healthy understanding of the software architecture and an ability to traverse and analyze the code base to find and correct the bug.

In the absence of a coherent software architecture, finding a bug in a large code base can be equivalent to finding a needle in a haystack. How can the architecture help find and correct a software error?

The major challenge in finding a bug in a large system is to use the architecture to narrow the search to a particular module or level where the bug is most likely to occur. For instance, if the bug seems to be “cosmetic” in nature, with no apparent impact on the permanent data maintained by the system, it may be traceable to a single user interface module.

On the other hand, if the bug seems to have a more permanent impact on the system’s data, the developer may need to follow the bug all the way down from the user interface level to a related module at the database level. In either case, the developer should understand the code base at different levels, including the interdependencies among different modules.

The examples in the following sections illustrate the effective interplay between tools, users, software architecture, and developer skills in identifying and removing bugs from an active code base.

---

9Projects like Linux and Apache, and language projects like Perl, PHP, and MySQL, maintain an especially close connection between developers and users, since the developers are the users. Mozilla, which maintains the open source Firefox browser, is a good example where a user community of non-developers participates directly and closely with developers. Moreover, there are many examples of proprietary software projects that have wide participation of their user base through alpha and beta testing processes.

10These examples represent actual errors that were discovered by users and corrected by developers during the first year after the RMH Homebase project was installed and put into productive use.
if ($edit==true &&
  (!$days[6]->get_year()<$year ||
   ($days[6]->get_year()=>$year &&
    $days[6]->get_day_of_year()<$doy)
  ) &&
  $_SESSION['access_level']>=2)

FIGURE 5.15: Locating a bug in the calendar.php module.

5.4.2 Example 1: A User Interface Bug

The simplest kind of bug to detect and correct is one that can be isolated in a single module in the user interface part of the code base.

Here is a bug recently reported by one of the users of RMH Homebase:

Hi Alex,

On the calendar, beginning in January the button to click to save notes is missing. The last week it appears on is December 21-27.

Thank you!

Gina

To recreate this error, Alex tried to edit the calendar page for the same week reported by the user. He used a sandbox version of RMH Homebase, so as not to interfere with the “live” version.

At the bottom of each calendar week, the button “Save changes to all notes” should appear, as it does in Figure 5.14. For some reason, this button does not appear on any calendar week after the week of December 21-27.

So the first step in debugging is to locate the module that manages the calendar form, which is called calendar.php. We locate this by referring back to the RMH system architecture, which has the structure shown in Figure 4.3.

The next step is to think about what causes this button to appear in the first place. The particular date on which the calendar is being edited must play a role in this determination, since we know it is not possible to edit calendar shifts for weeks that are fully gone by (try it!).

So the appearance of this button must be determined by a comparison of the date when editing took place with the dates of the calendar week being edited. If the week is fully in the past, then this button should not appear.

We can begin examining the calendar.php module for a code snippet that determines whether or not to display this button. Such a snippet occurs near the bottom of the calendar module in the code shown in Figure 5.15.

This is pretty ugly, so let’s parse it a bit more. We quickly learn that $edit==true is the test that determines whether the calendar is in “edit” mode or “view mode,” respectively. We also learn that $edit==true is the test to be sure that the user has edit access to the calendar notes ... more about this in Chapter 8.
Now for the gritty middle part of this code. A week is an array of 7 days, indexed from 0 to 6, and so the reference $days[6] is talking about the last day of the week. The references $year and $doy are talking about the year and the day of the year (0 to 365) in which the calendar editing is currently taking place.

So literally this code is displaying the “Save changes to all notes” button when the following three conditions are met:

1. the user is editing a calendar week,
2. that calendar week being edited does not fully precede the current week, and
3. the user is authorized to edit the notes.

Clearly, the fault must lie with item 2, since otherwise the error would not appear only in selected weeks. Looking at the logic of item 2, we can break down the code a little more finely to read:

\[
\text{the calendar week being edited is:}
\text{not (last year or}
\text{(this year or later and its day of the year precedes today))}
\]

Now the error becomes more apparent, since the clause “or later” allows the earlier part of a future year to “precede” today, which is incorrect. This would explain why the “Save changes to all notes” button disappears at the beginning of next year but not for the week of December 21–27.

To correct that error, we need to eliminate that clause so that the text reads:

\[
\text{the calendar week being edited is:}
\text{not (last year or}
\text{(this year and its day of the year precedes today))}
\]

To fix the code, we need only to change the $ operator in the third line of Figure 5.15 to ==, and then retest the calendar editing form to be sure we have corrected the error properly.

We note also that this sort of bug can occur whenever a calendar is being displayed—not only in the calendar.php module but also in the calendar-Fam.php module (which displays a week on the Family Room Calendar). A careful checking of the code in the latter module reveals that the same bug reoccurs there as well.

An important lesson comes out of this discovery. That is, when the same bug occurs identically in two or more places (because the code is identically wrong in both places), we may have an opportunity for refactoring. In our
example, we can extract a new function and replace all occurrences of the duplicated code by appropriate calls to that new function. This activity is left as an exercise.

### 5.4.3 Example 2: A Multi-Level Bug

Many bugs in complex software systems require the developer to navigate through the architecture and utilize information that the IDE provides about coupling between modules.

Here is a bug reported by one of the users of *RMH Homebase*:

Hello!

I think we may have discovered another bug: a volunteer created a sub call list and began making calls, she saved the information in the sub call list. I went in and added notes under the shifts which then deleted her sub call lists and the information. If it would be helpful, I can try to recreate this with you over the phone.

Thanks and no hurry

Gina

The first response from the developer was to confirm and understand the error more precisely by trying to recreate it on the sandbox version of *RMH Homebase*, so as not to interfere with the “live” version. This exercise invoked services from the following modules (the date reflects the week for which the exercise was being conducted, and is arbitrary):

- calendar.php?id=10-29-09&edit=true
- editShift.php?shift=10-29-09-12-3
- subCallList.php

Now when we return to the calendar.php module and edit and save the notes for some other shift, the sub call list associated with the shift 10-29-09-12-3 suddenly vanishes.

Let’s look at the code that is responsible for saving the shift, day, and guest chef notes. That code is in the function process_edit_notes inside the calendar.inc module. Within that code, there are only two ways in which the shift in question ($shift), and hence its sub call list, can be permanently changed. One would be to execute the update_dbShifts($shift) call midway through that function. The other would be to execute the update_dbDates($days[$i]) call near the end of that function.

Looking at the update_dbShifts function inside the dbShifts module, we see that it is accomplished through a delete_dbShifts call followed by an insert_dbShifts call. No other side-effects seem to be taking place here.

Looking at the update_dbDates function inside the dbDates module, we see a similar pattern—a delete followed by an insert. However, there is something else going on here, as shown in Figure 5.16.
function delete_dbDates($d) {
    ...
    $shifts = $d->get_shifts();
    foreach ($shifts as $key => $value) {
        $s = $d->get_shift($key);
        delete_dbShifts($s);
        delete_dbSCL(new SCL($s->get_id(),null,null,null,null));
    }
}

function update_dbDates($d) {
    if (! $d instanceof RMHdate)
        die ("Invalid argument for dbDates->update_dbDates call");
    delete_dbDates($d);
    insert_dbDates($d);
}

FIGURE 5.16: Locating a bug in the dbDates module.

That is, while an update is accomplished by deleting and adding the same shift to dbShifts, each shift’s sub call list is also deleted at the same time. Once gone from the database, that sub call list cannot be re-added, since sub call lists are stored in a separate table.

So the proper correction for this bug is to remove the delete_dbSCL call from the delete_dbDates function, leaving each sub call list unchanged in the database even though its corresponding shift has been deleted. Once the shift is re-inserted into the database to complete the update, it is reunited with its old sub call list (if it had one).

5.5 Extending the Software for a New Project

So far in this chapter, we have focused on techniques for understanding the elements of a software system by analyzing, refactoring, testing, and debugging an existing code base. Our focus on architecture, testing, and debugging shows how classes and modules relate to an existing system’s functionality at the user level.

At this point, we are prepared to consider the addition of new features to a code base, which will be guided by a new set of requirements. The ultimate
goal, of course, is to create an enhanced software system that will provide new functionality for the user.

The first step in reaching that goal is to listen to users, learn the new system requirements, and then adapt and create new classes and modules to implement the enhanced functionality suggested by the users and the new requirements and use cases.

New requirements for an open source software artifact usually originate from the collected views of current system users. Often these requirements appear in a post on a discussion thread. In the case of RMH Homebase, the user sent an e-mail to the developer outlining a “wish list” of nine new features that would be useful and bugs that should be removed. The text of that e-mail, along with a complete list of the new features and bugs that it identifies, appears in Appendix B.

The entire e-mail from the user suggests a collection of development activities that will be revisited incrementally throughout the next four chapters. Most of these involve additions to different parts of the code base (Chapters 6, 7, and 8). Many of these new features require additions to the user documentation (Chapter 9).

In this section, we concentrate on taking the first step toward implementing these new features: reading and understanding the new use case that comes directly from four of these nine new requirements.

5.5.1 A New Use Case

Suppose we want to add new functionality to RMH Homebase that will improve its searching and reporting capabilities. In particular, the House Manager has asked for modifications so that the following new kinds of information can be retrieved:

1. A summary of the total and average number of vacancies for each day and shift, for any series of weeks in the archive, accessible only to the House Manager

2. A summary of the total and average number of shifts worked by any volunteer, for any series of weeks in the archive, accessible only to the House Manager

3. A list of all volunteers who have not been scheduled for any series of dates in the past

4. A list of all inactive volunteers

5. The ability to export the data for any volunteer so that it can be imported by another application

The House Manager should be able to change the status (active/inactive) of any volunteer(s) in the list, and the entire database entry for any volunteer(s)
in the list should also be exportable as a comma-separated list suitable for import to a spreadsheet or other application.

Figure 5.17 describes a new use case that encapsulates this requirement. The use case is called “Housecleaning,” and it is an extension of the original use case “ViewAList” shown in Figure A.14.

This use case suggests several questions for the developers to answer as they consider extending the existing code base to satisfy these new requirements.

1. What user interface modifications, including navigation changes, are needed for these functions to be effectively added to \textit{RMH Homebase}?

2. What new or modified domain classes are required for accomplishing this task?

3. What new or modified database tables are needed?

4. What are the security implications of this new functionality?

5. What new user documentation is needed to support these new capabilities?

5.5.2 Impact on the Code Base

Several modules in the code base will be affected when this use case is implemented. Our task in the following sections is to identify those modules and sketch what will be needed to implement the new requirements. The implementation itself, along with the associated refactoring and testing, will be fully treated in later chapters.

5.5.2.1 User Interface

A review of the current system suggests that the items in this use case can be accommodated by developing a new \textit{Calendar Housecleaning} form, like the one sketched in Figure 5.18.

In making this addition, we can also streamline the user interface by adding a new \textit{calendar: search} tab to the main menu which will lead the user to the new form. This will allow the House Manager to view vacancy summaries and manage inactive volunteer lists directly.

With this change, the new main menu that the House Manager sees could look like this:

\begin{verbatim}
home | about | calendar: house, family room, search |
people: view, search, add
master schedule: house | family room | log | help | logout
\end{verbatim}
Housecleaning

**Description:** Occasionally, the House Manager needs to view the total and average number of vacancies for each shift and each day, over a period of weeks in the recent past. Various other information about volunteer schedules should also be retrievable, such as the total and average number of shifts worked by any particular volunteer for any such period, a list of active volunteers who have not been scheduled during that period, and a list of all the inactive volunteers. The active/inactive status of any person on such a list should be reversible. The database entries for any group of volunteers should be exportable as a comma-separated list so that they can be imported into a spreadsheet or another application.

**Actor:** the House Manager

**Goals:** Different selections should be possible, in particular:

- “View the total and average number of vacancies for each day and shift, over any period of time in the past.”
- “View the total and average number of shifts worked by any volunteer, over any period of time in the past.”
- “View all active volunteers who have not been scheduled for a shift over any period of time (change their status to inactive).”
- “View all inactive volunteers (change anyone’s status to active).”
- “Export the database entries for any group of volunteers in the list.”

**Preconditions:** 1. A beginning and ending date for the period of time has been specified.
2. One of the above selections has been made.

**Postconditions:** 1. The desired list is displayed, and some volunteers’ status is changed or their data is exported.

**Related Use Cases:** ViewAList

**Steps:**

<table>
<thead>
<tr>
<th>Actor</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Log on to Web site.</td>
<td>2. Ask for id and password.</td>
</tr>
<tr>
<td>3. Enter id and password.</td>
<td>4. Verify and display outcome.</td>
</tr>
<tr>
<td>5. Identify a period of time.</td>
<td>6. Retrieve shifts for that period.</td>
</tr>
<tr>
<td>7. Select an option.</td>
<td>8. Display/export those shifts.</td>
</tr>
</tbody>
</table>

**FIGURE 5.17:** Use case Housecleaning
5.5.2 Impact on Classes and Modules

To determine which classes and modules are affected for implementing this new use case, we must take a systematic look at the code base. This process is discussed in the next chapter, which provides a broader treatment of the influence of software architecture on development.

But informally, we can easily observe that implementing the main menu change will affect the header.php module. We can also predict that a new user interface module will be required, one that manages the user’s entry and retrieval of information while performing the new use case.

All underlying classes and modules that this new module imports will also need to be suitably modified. In particular, it will need to call upon existing and new functionality provided by the Shift, RMHdate, Week, and Person classes, as well as their associated database modules dbShift, dbDates, dbWeeks, and dbPersons.

5.5.2.3 Impact on the Database

When considering the impact of this new use case on the database tables, we ask the question: “Does an implementation of the use case require additional information to be stored permanently in the database tables, beyond what is already stored there?”

If such additional information needs to be added to existing tables, then all functions in the database module that work with that table must also be revised accordingly. If an entirely new table is needed, then we must define
a new module to manage that table, including initialization, adding rows, deleting rows, and updating rows.

For the new use case described above, we do not need a new database table or new table columns, since all of the new reports can be fully generated from information already in the tables. Moreover, none of the reports need to be kept permanently after the manager retrieves and views them.

5.5.2.4 System Security

All of the current system’s security constraints must be upheld when the new functionality is implemented. In addition, only the House Manager should have access to the functionality described in the new use case.

For this to happen, we must ensure that all code added to the user interface modules occurs “under the umbrella” of \$_SESSION[‘access_level’]==2, which distinguishes a manager’s login privileges from everyone else’s.

5.5.2.5 User Help

The development of effective user support is so important that it is covered separately in Chapter 9. For this discussion, we note that the current user help pages for RMH Homebase, as summarized in Figure 9.7, will need only minor modifications to document these new features for users.

For the first requirement, the help page “Generating and publishing calendar weeks” should be modified and a new help page “Summarizing Vacant Shifts (Managers Only)” should be added. For the second requirement, the help page “Searching for People” should be changed and a new help page “Summarizing People’s Hours (Managers Only)” should be added.

5.5.3 Team Discussions

At this time, the team should be actively discussing the initial development of this new project, including the assignment of tasks, the scheduling of milestones, and a review of design decisions already made (such as those outlined above).

Since the user initiated the development of these new requirements by stating needs that aren’t fulfilled by the current software, it is imperative that the user remain in the loop as new classes, modules, security constraints, and user help menus are identified for fulfilling those needs.

These initial discussions should include a review of the user interface (forms) that will be changed or added, the strategy that will be used to ensure security, and the new help menus that will be needed. Discussions about technical details involving new and existing classes, modules, and database tables may require less user participation.
5.6 Summary

This chapter has presented fundamental ideas underlying working with a code base—refactoring, testing, and debugging. Strategies for developing new classes and modules that implement new functionality were introduced as a by-product of reading a new requirements statement and relying on knowledge of the existing code base.

While refactoring and testing are highly individualized activities, debugging and adding new features are highly collaborative activities. Some of the exercises below will help individuals develop basic refactoring and testing skills, and others will help teams to begin working together in reading requirements, reading code, and defining classes and modules for new projects that build upon an existing code base.

Exercises

5.1 Examine the RMH Homebase release 1.5 code base and its accompanying documentation in Appendix A. Identify at least one instance of each of the following “bad smells” in the code base.

a. Long Method
b. Too Few Comments
c. Data Clumps
d. Speculative Generality

5.2 For each of the bad smells identified in the previous exercise, complete a refactoring that will remove it from the code base. Which of these refactorings reduces the size of the code base? Which one(s) improves its readability?

5.3 After completing the previous exercise, rerun the unit tests to be sure that your refactorings have not compromised any of the project’s functionality.

5.4 Design a new unit test for the dbSchedules module that tests each function except the setup_dbschedules function. Add a call to this unit test from within the AllTests module shown in Figure 5.8; be sure that your call occurs in the correct sequence.

5.5 Design a new unit test for the editMasterSchedule module that tests each of its functions. Add a call to this unit test from within the AllTests
5.6 Consider the strategy for sequencing unit tests described in Section 5.3.3, along with the extended layering chart that you developed in Exercise 4.5. Extend that strategy by adding to the sequence unit tests for all the remaining modules in *RMH Homebase*. Be sure that you locate each unit test in its proper place in the sequence.

5.7 The debugging example in Section 5.4.2 suggests an opportunity for refactoring. That is, the determination of whether or not a given shift in the calendar display should show its “notes” field for editing is governed by the question of whether or not today’s date follows that shift’s date chronologically. If it does, the shift’s “notes” field should be displayed for editing; otherwise, it should not.

a. Locate the module in *RMH Homebase* that displays a shift’s “notes” field for editing.

b. Locate instances of the code in that module which are similar to the “ugly” code in Figure 5.15. Confirm that those instances perform the same computation as the ugly code.

c. Define a new function called “predates(a, b)” that performs that same computation and returns true or false if a predates b, respectively. Insert that function as a new feature of the Shift class.

d. Replace each instance of the “ugly” code shown in Figure 5.15 by a call to your new function.

e. Test your refactoring by adding a new test case to each of the test-Calendar and testeditShift unit tests and rerunning AllTests.

5.8 Examine the *RMH Homebase* requirements in Appendix A and the release 1.5 code base to answer the following questions:

a. Why do some shifts have a “Generate Sub Call List” button and others have a “View Sub Call List” button instead? What does that tell you about the relationship between shifts and sub call lists—does every shift have a sub call list?

b. What is an “archived week” and how does a week become archived? Who does the archiving? How can an archived week be removed from the database? Suggest a different approach for handling this, including its advantages and disadvantages vs the current approach.

5.9 Suppose we want to generalize the *RMH Homebase* software so that it can be more easily adapted for use in other organizations with similar scheduling needs. Typically, another organization’s scheduling needs will vary in the following fundamental ways:
• The number of venues for scheduling can vary. For *RMH Homebase*, the number of venues is two, the House and the Family Room.

• The layout of shifts for a typical week on the calendar can vary for each venue.

• The master schedule's shifts and number of slots in each shift will vary for each venue. For *RMH Homebase*, the House schedules four shifts on each weekday (9-12, 12-3, 3-6, and 6-9) and a different layout on the weekends. The Family Room schedules three shifts on each weekday (10-1, 1-4, and 4-7) and no shifts on the weekends.

• The frequency for repeating the master schedule will vary: some organizations will schedule two groups of volunteers, each on a bi-weekly basis, others will rotate four groups on a monthly cycle, and others will schedule the same group every week.

• The individual fields in a volunteer's database entry will vary. In most cases, a person's first name, last name, home address, phone number, and e-mail address are standard fields. For *RMH Homebase*, additional fields include an alternate phone number, a type (volunteer, family room volunteer, etc.), shift availability for each day in the week, and special notes.

What new GUI pages, if any, would be needed for these functions to be effectively added to *RMH Homebase*? What would need to be added to the existing pages and menu items? What new classes would be required for accomplishing this task? What new database tables? What are the security implications of this new feature? Which classes and modules would be affected by this change, and which ones would not?

5.10 Suppose we want to integrate *RMH Homebase* with iCalendar (see http://en.wikipedia.org/wiki/iCalendar), which is a standard file format that allows people to keep personal calendars and exchange meeting times via e-mail. We would like the weekly schedule to be stored in this standard format and thus more easily downloaded, printed, and shared by Volunteers and the House Manager.

What new GUI forms, if any, would be needed for iCalendar to be effectively integrated with *RMH Homebase*? What would need to be added to the existing pages and menu items? What new classes would be required for accomplishing this task? What new database tables? What are the security implications of this new feature? Which classes and modules would be affected by this change, and which ones would not?