Design Patterns for Customer Testing

Misha Rybalov, Quality Centered Developer
autotest@gomisha.com

Abstract

This paper presents design patterns for automated customer tests. As applications continue to evolve, these patterns help lower maintenance costs and facilitate the creation of new customer tests. Each design pattern includes a definition, a discussion of the pattern details highlighting its suitability for customer test automation, and a code sample illustrating its use. These design patterns can be used regardless of programming language and test tool. As such, specific test tool code is omitted in this paper and for consistency examples are given in Java, using the JUnit testing framework. This paper aims to illustrate that test code should be treated with the same importance as application code. This assertion has the following consequences:

- tests should have a clean design that facilitates code reuse without duplication
- tests should be easily adaptable to modifications in the application
- tests should be easy to create
- tests should have a low maintenance cost

Using application development design principles, automated test development can become a well factored art form that is adaptable to application changes. This adaptability results in lowered maintenance costs and easier test creation.

About the Author

Misha Rybalov is a quality centered developer that works with Development and Quality Assurance departments to create automated testing frameworks and build quality into the product from the start of the development cycle. He has automated the testing of a wide array of projects including web, .NET, and Java customer tests, as well as standard unit tests, and a multi-threaded security system. Misha's passion is designing and building automated testing frameworks using design patterns. He can be reached at autotest@gomisha.com.

Introduction

Automated testing offers numerous benefits to any software organization, including finding defects cost-effectively early in the development cycle, providing rapid feedback, and giving developers the courage to refactor their code. With the recent popularization of Test Driven Development,[1] automated testing has also become a design tool for application development.

Ironically, with all the attention paid to the benefits of automated testing, the actual art of designing and writing automated tests is often assumed to be taken care of by the test tool or the test writer. The principles of simple design, refactoring, and design patterns that are so common in software development are under utilized when it comes to test development.

While unit testing involves very simple method calls and checking for expected results, customer tests are significantly more complicated. It is quite common to have a single customer test scenario involving dozens of navigations and checks which span multiple screens and modules. The consequences of a haphazard approach to automated customer testing can result in massive code duplication, tests that are cumbersome to create, costly to maintain, and buggy.

This paper presents a catalog of design patterns that facilitate the creation and maintenance of automated customer tests. Each pattern consists of 1) a definition, 2) a discussion of the pattern details highlighting its suitability for customer test automation, and 3) a code sample illustrating its use.

This paper uses a fictional web-based shopping cart and catalog application to demonstrate how each pattern can be implemented. We will test the application's capabilities of searching for, adding, and removing items from the cart, as well as checking out and processing coupons.

Test Tools and Code

The design patterns introduced in this paper are not test tool specific in any way. One of the goals of applying these patterns is to decouple test code from the test tool code. For our purposes, we will use the JUnit testing framework[2] to demonstrate the application of these patterns. These tests could just as easily have been written with another object-oriented testing framework (see Extreme Programming Software for a list of object-oriented testing frameworks[3]).

Customer Tests

This paper uses the term "customer tests" to refer to tests that exercise the application from an end-user's perspective. These types of tests are also referred to as functional or acceptance tests. Customer testing can be performed through a web browser (e.g. Internet Explorer, Netscape), an operating system's graphical user interface (e.g. Windows, Unix, MacOS), or any other user interface (e.g. Java, .NET). The patterns presented in this paper need not be limited to customer tests - they can be successfully applied to unit tests as well. This paper, however, focuses on customer tests because they are inherently more complex than unit tests and thus will derive the greatest benefit from design patterns.

Why Use Design Patterns to Automate Customer Tests?

Design patterns are used "to build flexible, reusable software."[4] An automated test is, in essence, a program that checks another program. Consequently, it is vulnerable to the same design problems and entropy as application code. These vulnerabilities, often referred to as "code smells,"[5] include test code duplication, tight coupling between the application and tests, and long test methods.

Traditionally, design patterns were not used for test code because test code was considered more simplistic and subordinate to application code. Also, most test tools generate customer test code through a record-and-playback mechanism (i.e. they record all user actions as the tester navigates through the application, create the corresponding code, and then play it back as the test), which encourages test developers to use the tool generated code without modifications. The challenge with both of these approaches is that tests become more difficult to maintain as the application changes over time. Design patterns help us minimize maintenance costs by making tests easier to update and more adaptable to application changes.

The patterns presented in this paper have been successfully used on real projects. These patterns can be applied in any object-oriented language such as Java, C++ or C#. They can also be used to refactor auto generated code from a commercial testing tool, but the mostly procedural nature of tool languages yields less benefit than an object-oriented language.

Patterns Catalog

I. Basic Patterns

This group of patterns applies object-oriented programming principles to reduce test code maintenance costs and facilitate new test creation. The patterns included are Template, Object Genie, Domain Test Object, and Transporter.

1. Template

Definition

Group a common sequence of steps for testing a module into a class and have subclasses specify unique test case data without changing the main algorithm's structure.

Discussion

Customer tests involve navigating to a particular part of the application and trying different scenarios. Even though each scenario is unique, there are still many commonalities between test cases. For instance, the way that you navigate to the module and the assertions that are made along the way can be grouped into one place - the Template class, which is a test-specific application of the Template Method design pattern.[6]

When the application inevitably changes, the majority of updates will occur in the Template class instead of within the test cases. This is because the Template class encapsulates navigation and assertion logic, which is more vulnerable to application changes than the data contained in test cases.

After each update to the Template class, the modifications will cascade automatically to all subclasses. This has the added benefit of making each test case (subclass) simpler since it is only required to specify the unique aspects of the test instead of the entire sequence of test steps and assertions. This reduces test code duplication and standardizes common navigations and assertions of related test cases.

If not using this design pattern, then every time the application changed or a new test case was added, the test writer would have to copy or re-record all the navigations and assertions that are part of each test. In that case, the automation effort would incur serious maintenance costs.

Since the Template pattern is adaptable to application changes, it can be used early in the development cycle without having to wait for application completion. Templates can be created and then continually updated as features are developed. This lends itself to agile software development practices such as Test Driven Development[7] where tests are written early in the software lifecycle and are an integral part of the development effort.

Example

Figure 1 presents two traditional tests which are then contrasted with Figures 2 and 3 which use the Template pattern. These tests check the functionality of adding and removing items from a shopping cart.

Figure 1: Non-template test cases.

```
import junit.framework.*;

public class ShoppingCartTest extends TestCase {
        public ShoppingCartTest(String pName) { super(pName); }
        public static Test suite() { return new TestSuite(ShoppingCartTest.class); }
        public void testAddItem() {  //1st test case
                login();
                searchForItem1();
                addItem1ToCart();
                searchForItem2();
                addItem2ToCart();
                logout();
        }
```

```
    public void testRemoveItem() {  //2nd test case
        login();
        searchForItem1();
        addItem1ToCart();
        addItem1ToCart();
        removeItem1FromCart();
        searchForItem2();
        addItem2ToCart();
        removeItem2FromCart();
        logout();
    }
    //test tool specific implementations of methods
    //(e.g. login(), searchForItem1(), etc.)
}
```

The first test logs in, searches for item one, adds it to the cart, searches for item two, adds it to the cart, and logs out. The second test logs in, searches for item one, adds it twice to the cart, removes one quantity of item one from the cart, searches for item two, adds it to the cart, removes item two from the cart, and logs out. These two traditional tests have the following common elements: logging in, searching for item one, searching for item two, and logging out. These common steps can be moved into a new Template class.

Figure 2: Template class.

```
import junit.framework.*;

public abstract class ShoppingCartTemplate extends TestCase {
    public ShoppingCartTemplate(String pName) { super(pName); }

    protected void setUp() {
        login();
        searchForItem1();
        cartAction1();  // <------ test hook 1
        searchForItem2();
        cartAction2(); //<-------- test hook 2
        logout();
    }

    //Subclasses are supposed to implement these test hooks anyway they like.
    abstract protected void cartAction1();
    abstract protected void cartAction2();

    //test tool specific implementations of methods
    //(e.g. login(), searchForItem1(), etc.)
}
```

The Template defines the main sequence of steps as 1) login, 2) search for item one, 3) do something (to be defined by subclasses), 4) search for item two, 5) do something else (to be defined by subclasses), and 6) log out. Note that JUnit's setUp() method is used to define the sequence of steps, which enables this sequence of steps to occur for every test case subclass.

Each test case can now be moved to a separate subclass and implement the abstract methods of the Template.

Figure 3: Two test cases using the Template.

```
public class ShoppingCartTest_AddItem extends ShoppingCartTemplate {
        public ShoppingCartTest_AddItem(String pName) { super(pName); }
        protected void cartAction1() {
                addItem1ToCart();
        }
        protected void cartAction2() {
                addItem2ToCart();
        }
}

public class ShoppingCartTest_RemoveItem extends ShoppingCartTemplate {
        public ShoppingCartTest_RemoveItem(String pName) { super(pName); }
        protected void cartAction1() {
                addItem1ToCart();
                addItem1ToCart();
                removeItem1FromCart();
        }
        protected void cartAction2() {
                addItem2ToCart();
                removeItem2FromCart();
        }
}
```

The Add test case (ShoppingCartTest_AddItem) implements the two abstract methods by adding items one and two to the cart, respectively. The Remove test case (ShoppingCartTest_RemoveItem) implements the first abstract method by adding item one twice and then removing it. It implements the second abstract method by adding item two to the cart and then removing it.

Even though it appears that there is not a great reduction in code size in this example, code consolidation becomes more evident in real life scenarios which contain dozens of test steps and hundreds of test cases.

Adding more tests is relatively easy with the Template pattern since one only needs to implement the two abstract methods of the Template (i.e. cartAction1() and cartAction2()) rather than having to write all the setup and navigation code. This consolidation of redundant test steps into one place can drastically reduce maintenance costs as the application continues to evolve. For instance, if the application were to change so that customers now had to go to a specials screen before being able to add any items to the their cart, our Template can be updated to account for this and have the change cascade to every test without having to modify each test individually.

Figure 4: Modification of Template.

```
import junit.framework.*;

public abstract class ShoppingCartTemplate extends TestCase {
        public ShoppingCartTemplate(String pName) { super(pName); }

        protected void setUp() {
                login();
                gotoSpecials();    // <------ new test step added
                searchForItem1();
                cartAction1();     // <------ test hook 1
                searchForItem2();
                cartAction2();     // <------ test hook 2
                logout();
        }
        abstract protected void cartAction1();
        abstract protected void cartAction2();
}
```

2. Object Genie

Definition

Relieve tests from having to instantiate and initialize common non-trivial objects by having them ask for those objects from a central authority that grants objects in a pre-defined state. This pattern is also known as ObjectMother.[8]

Discussion

There are many occasions during testing when multiple tests will require the same object and then each proceed to interact with it in different ways. If it is a simple object, each test can just instantiate it normally (SomeObject object = new SomeObject();) However, there are many occasions where the object that is needed by tests is cumbersome to instantiate or initialize.

The Object Genie consolidates the construction and initialization of that object to just one place, thereby reducing future maintenance costs and facilitating the creation of new tests. If the way an object is instantiated or initialized ever changes, the update would only be done in the Object Genie and the test cases that use that Object Genie would be unaffected.

Example

In our shopping cart application, multiple tests (e.g. test deleting an item from a cart, test the checkout process, test processing a coupon) require a shopping cart with a few items in it. Without the Object Genie pattern, each test would have to re-create and re-populate the shopping cart, increasing code duplication and maintenance costs. Using an Object Genie, a test can request a non-empty shopping cart and then proceed with testing. The Object Genie handles the instantiation of the shopping cart and populates it. In Figure 5, the Object Genie populates the shopping cart with two items (orange juice and cereal).

Figure 5: Defining a shopping cart Object Genie.

```
public class CartGenie {
        public static ShoppingCart getNonEmptyCart() {
                ShoppingCart shoppingCart = new ShoppingCart();
                Item ojItem = searchForItem("orange juice");
                shoppingCart.add(ojItem);
                Item cerealItem = searchForItem("cereal");
                shoppingCart.add(cerealItem);
                return shoppingCart;
        }
        private static Item searchForItem(String pItemId) {
                //search for the item based on the item id
        }
}
```

Each test would call CartGenie.getNonEmptyCart() to receive an initialized shopping cart and then proceed with the details of the test. The tests would be decoupled from having to know how a shopping cart is instantiated and populated, thereby reducing future maintenance costs (see Dumb Test in Section II: General Design Principles).

3. Domain Test Object (DTO)

Definition

Encapsulates an application's visual components into objects that can be reused by many tests.

Discussion

We use object-oriented programming (OOP) to model real world objects inside our applications. We can also use OOP to model visual representations of the application (e.g. a shopping cart, a coupon, and a catalog) in our tests. In other words, the application uses objects to model the real world, while the tests use objects to model the application's world.

The DTO pattern groups visual representations of the application into objects that can be used by tests. Each test communicates to the DTO what data is expected but not how the data should appear (i.e. order and name of input fields to which the data applies or placement of the data on the page). The DTO then checks the test provided data against the actual data in the application. Thus, the DTO allows us to decouple the expected data from its visual representation. When the visual representation of the application changes (e.g. new table columns, new fields, and new labels), the update is done to the DTO instead of to each test. All the tests that use that DTO will be insulated from the change and will not need updating, thereby reducing maintenance costs. In essence, the DTO is a test-specific application of the Model/View/Controller (MVC) design pattern,[9] where the tests are the Model and the DTO is the View.

Example

Customer tests need to check the contents of the shopping cart after certain actions are performed (e.g. adding, removing, updating quantity of cart items). A simple table can represent the shopping cart by displaying the list of items, quantity, unit price and total price.

Table 1: Visual representation of shopping cart contents.

Item	Quantity	Unit Price	Price
Milk	1	$2.99	$2.99
Bread	2	$1.50	$3.00
Total			$5.99

This table is the application's representation of a shopping cart. Every time a test wants to check the cart's contents it must check this table. Thus, any changes to this table (e.g. a new column being added) could have severe test maintenance consequences. To minimize this maintenance cost, we can model the shopping cart table as a DTO and have tests interact with the DTO instead of with the table directly. The following code sample demonstrates checking the shopping cart page without using a DTO. This is then contrasted with doing the same check using a DTO.

Figure 6: Checking shopping cart contents without DTO.

```
public void checkCart_noDto() {
        String [][] expectedCartTableCells = {
                {"Item",  "Quantity", "Unit Price", "Price"},
                {"Milk",  "1",        "$2.99",      "$2.99"},
                {"Break", "2",        "$1.50",      "$3.00"},
                {"",      "",         "",           "",    },
                {"Total", "",         "",           "$5.99"}
        };
        //tool-specific way to retrieve table cell contents
        String [][] actualCartTableCells = getCartTableCells();

        //check every web table cell of the cart page
        assertEquals(actualCartTableCells.length, expectedCartTableCells.length);
        for(int i=0; i<actualCartTableCells.length; i++) {
            assertEquals(expectedCartTableCells[i].length, actualCartTableCells[i].length);
            for(int j=0; j<actualCartTableCells[i].length; j++) {
                assertEquals(expectedCartTableCells[i][j], actualCartTableCells[i][j]);
            }
        }
}
```

Without using the DTO pattern, we are forced to specify the expected content in a rigid order that is vulnerable to application changes. Every time we need to check the contents of the shopping cart page, we would have to repeat the same checking code, but with different values. For example, we would need to replicate the same checking code after adding to the cart, removing from a cart, updating the quantity of items in the cart, etc. The maintenance problem occurs when the layout of the cart changes and we would have to update all the checking code for each cart action. For instance, the application might change so that the shopping cart page has a new column called "Discount" which contains percentage values of the discount for an item. The application might also change by having the quantity and unit price columns switched. In both of these cases, all our test code would break because now the web table wouldn't match our expected table values. By using a DTO, we can avoid this problem and be more adaptable to application changes. The following code demonstrates how a DTO object can be used:

Figure 7: Checking shopping cart contents with DTO.

```
public void checkCart_withDto() {
        ShoppingCart shoppingCart = new ShoppingCart();

        //...add/remove/edit cart using other patterns

        ShoppingCartDto shoppingCartDto = new ShoppingCartDto();

        //setup expected values
        shoppingCartDto.setItem("1", "2.99", "2.99", "Milk");
        shoppingCartDto.setItem("2", "1.50", "3.00", "Bread");
        shoppingCartDto.setTotal("5.99");
        shoppingCartDto.check(shoppingCart);
}

public class ShoppingCartDto {
        public void setItem(String pQuantity, String pUnitPrice, String pTotalPrice,
        String pName) {
                //code to store expected item internally would follow
        }

        public void setTotal(String pTotal) {
                //code to store expected total internally would follow
        }
```

```
	public void check(ShoppingCart pShoppingCart) {
		//code to check expected versus actual shopping cart contents would follow
	}
}
```

From Figure 7, you can see that the DTO is a separate class that is instantiated and configured by test cases. Note that the DTO knows how to check itself so tests never have to explicitly specify the column order or any other details about the shopping cart table. The tests simply tell the DTO the items that are expected to be in the cart and then ask the DTO to check the cart contents. Now even if the entire shopping cart page changed so that the cart is no longer represented by a table, the update would be done to the DTO while leaving the tests insulated from such a drastic change.

4. Transporter

Definition

Create a central authority that navigates through the application on behalf of tests.

Discussion

Similar to the Object Genie is the concept of a Transporter. Whereas the Object Genie grants tests whatever objects they need, a Transporter encapsulates navigation code so that tests don't have to know the details of how to move around the application. Instead, tests use the Transporter to navigate throughout the application on their behalf.

If the application changes so that users must now alter the way they navigate to a specific module, the modifications would be localized to the Transporter and changes would automatically cascade to all tests that use that Transporter. This reduces maintenance costs since none of the tests would be affected by this change and facilitates creating new tests since there is less test code to write.

Example

Each test needs a way to navigate to pages and then perform specific actions (e.g. log in, enter a coupon, buy an item on special, etc.). To use a coupon, one must first do the following navigations: login and navigate to the checkout page. There must also be items in the cart to which the coupon applies - this is taken care of by the Object Genie. The following code samples demonstrate the creation and use of a Transporter that navigates to the login and checkout pages.

Figure 8: Defining a Transporter.

```
public class Transporter {
	public static void login() {
		//Tool specific code to navigate to login screen and login to the application
	}
```

```
          public static void gotoCheckout() {
                    //Tool specific code to navigate to the checkout page
          }
}
```

Figure 9: Using a Transporter.

```
import junit.framework.*;

public class CouponTest extends TestCase {
          public CouponTest(String pName) { super(pName); }

          public static Test suite() { return new TestSuite(CouponTest.class); }

          protected void setUp() {
                    Transporter.login();                // <---- loose coupling
                    ShoppingCart shoppingCart = CartGenie.getNonEmptyCart();
                    Transporter.gotoCheckout();   //<----- loose coupling
          }
          public void testCheckoutWithValidCoupon() {    //tests for a valid coupon
                    applyValidCoupon();
                    checkCouponAccepted();
          }
          public void testCheckoutWithExpiredCoupon() { //test for an expired coupon
                    applyExpiredCoupon();
                    checkCouponRejected();
          }
          public void testCheckoutWithInvalidCoupon() {  //test for an invalid coupon
                    applyInvalidCoupon();
                    checkCouponRejected();
          }
          //… specific implementations of test methods (e.g. applyValidCoupon())
}
```

By moving navigation to the login and checkout pages to a central location, we allow other tests to reuse that functionality. The CouponTest class is now only responsible for testing coupon functionality without worrying about the details of how to get to the coupon page. This loose coupling makes the test more maintainable.

II. General Design Principles

These patterns introduce general test design principles that can be used in conjunction with the basic patterns introduced previously. This group includes Dumb Test, Independent Test, Don't Repeat Yourself, and Multiple Failures.

1. Dumb Test

Definition

Minimize a test's knowledge about navigation, checking logic and user interface details.

Discussion

This principle is commonly used in application programming where decoupling objects from each other allows each object to change without affecting the other. Applying this principle to test development, the Dumb Test pattern focuses on making an individual test know as little about its surroundings as possible. This makes Dumb Tests less vulnerable to changes in the application, which in turn lead to lower test maintenance costs. Each Dumb Test doesn't know the sequence of steps that were taken to get to the component being tested nor about common checking that has been done by other test modules. Ideally, the only things a Dumb Test is aware of are its inputs and its expected outputs.

Dumb tests work well with the Template, Object Genie, DTO, and Transporter patterns since common test steps can be consolidated into these classes. When the application changes, the bulk of the updates will need to be made in those classes, while individual Dumb Tests will be mostly insulated from the changes. This creates very compact tests that are quick to create and cost less to maintain than tests that provide all the details about navigation and assertions. In a nutshell, a Dumb Test doesn't see the big picture.

Example

Both test cases in the Template pattern example are Dumb Tests (see Figure 3). They have no knowledge of how to login, logout, go to the specials page or search for items. If these parts of the application were to ever change, the Dumb Tests would be unaffected by it. The tests only know that they need to add and remove items from the cart.

2. Independent Test

Definition

Each test leaves the application in the same pre-defined state, irrespective of whether the test passed or failed.

Discussion

In order to ensure test integrity, it is important that tests be unaffected by the outcome of other tests. Each test expects to start from a constant application state. This requires each test to de-initialize to this constant state upon completion or failure. If this does not occur then subsequent tests will fail to execute. Unit testing frameworks such as JUnit recognize the importance of this principle and have mechanisms to initialize and de-initialize each test in order to keep the application in a consistent state (i.e. using setUp() and tearDown() methods).

Example

For our sample application, it is important that each test start on the login page, with an empty shopping cart. This necessitates a mechanism for emptying the shopping cart and logging out at the end of each test or upon test failure. If we use the Template pattern, we can insert these common steps of initialization and de-initialization, right into the Template, thereby relieving test writers from having to remember to include these steps in every test and ensuring tests stay independent.

Figure 10: Common initialization and de-initialization steps for each test case.

```
    protected void setUp() {
        login();          // <------ common setup code for all test cases
        searchForItem1();
        cartAction1();
        searchForItem2();
        cartAction2();
        emptyCart();  // <------- common tear down code for all test cases
        logout();
    }
```

3. Don't Repeat Yourself (DRY)

Definition

Minimize test code duplication to lower test maintenance costs.

Discussion

The evils of code duplication have been documented extensively in software development literature.[10] The same principles that apply to application development also apply to test development. Namely, just as it becomes a maintenance problem to modify application code in multiple places, the same problem occurs when tests need modification in multiple places. Many of the patterns mentioned in this paper (e.g. Template, Object Genie, DTO, Transporter) minimize code duplication by consolidating common test code into one place instead of duplicating it. The advantage of this approach is that it becomes easier to update tests as the application changes, which in turn lowers the maintenance costs of tests.

Example

To test the coupon feature of the application, we need tests for submitting a valid coupon, an expired coupon, and an invalid coupon. In order to test these scenarios, we need to load our cart with items to which the coupon would apply. This would be the setup part of the test (see Independent Test pattern), since before we could test the checkout process, we'd have to have a cart with items in it already. If we don't adhere to DRY, we would simply copy and paste previous test code that searched for an item(s) and inserted it into the cart. Figures 11 and 12 illustrate non-DRY and DRY approaches, respectively.

Figure 11: Repeating test code for coupon test cases.

```
import junit.framework.*;

public class CouponTest extends TestCase {
	public CouponTest(String pName) { super(pName); }

	public static Test suite() { return new TestSuite(CouponTest.class); }

	public void testValidCoupon() {
		login();
		searchForItem1();
		addItem1ToCart();
		searchForItem2();
		addItem2ToCart();
		gotoCheckout();
		applyValidCoupon();
		checkCouponAccepted();
	}

	public void testExpiredCoupon() {
		login();
		searchForItem1();
		addItem1ToCart();
		searchForItem2();
		addItem2ToCart();
		gotoCheckout();
		applyExpiredCoupon();
		checkCouponRejected();
	}

	public void testInvalidCoupon() {
		login();
		searchForItem1();
		addItem1ToCart();
		searchForItem2();
		addItem2ToCart();
		gotoCheckout();
		applyInvalidCoupon();
		checkCouponRejected();
	}
	//tool-specific implementations of methods (e.g. login(), addItem2ToCart(), etc.)
}
```

This non-DRY approach is recommended by some due to its inherent simplicity and explicitness.[11] The argument is that if a test developer tries to get too fancy with their test code, they will have to start writing tests for the test code, which can quickly get out of hand. However, writing more maintainable code does not mean that the code has to get more complex.

Using DRY, we have several options: 1) use the Template pattern to extract the act of filling up your cart to a super class, 2) use an Object Genie to grant us a cart with items already in it, 3) extract the setup code into a set up method so that each test case uses the same setup code, and 4) combine the above methods.

Template Pattern

Using a Template pattern, we can extract all the setup code into a super class and have each subclass implement a few small methods. All the common code is encapsulated in the super class and the subclasses don't have to concern themselves of dealing with it. The disadvantage, though, is that a separate subclass has to be created for each test case.

Object Genie

An Object Genie can grant us a cart with items in it anytime we want it. The advantage of this technique is that we are not forced to use subclasses. We just request the cart with items in it at the appropriate time.

Setup Method

JUnit supports Setup Methods that allow a test writer to place all common setup code into one place. The setup code gets executed before each test case is run. This is a good start to avoiding duplication, but often the code that gets factored out into the Setup Method could be duplicated in other Setup Methods for other tests. In our example, the setup code of filling a cart with items is needed by other tests, as well. For instance, to test cart manipulation functions (adding, removing, modifying items in the cart) we would need to have a cart with items in it as setup code. This implies that the setup code should be located in a more central location, as outlined in the previous two examples.

Combining Patterns

Filling up a cart is a common test step that is useful to many customer tests. The three pattern options to address this step can be used in combination to yield a greater benefit. Figure 12 combines the use of an Object Genie and a Setup Method. Other combinations are possible as well, including using all three patterns. We use the Setup Method to login, call the Object Genie, and go to the checkout page. The Object Genie returns a shopping cart with items in it. The result is that each test case only has to worry about entering a specific coupon and checking whether the coupon was accepted or rejected.

Figure 12: Consolidating test code using Object Genie and Setup Method.

```
import junit.framework.*;

public class CouponTest extends TestCase {
        public CouponTest(String pName) { super(pName); }
        public static Test suite() { return new TestSuite(CouponTest.class); }
        protected void setUp() {
                login();
```

```
            ShoppingCart shoppingCart = CartGenie.getNonEmptyCart();
            gotoCheckout();
    }
    public void testCheckoutWithValidCoupon() {
            applyValidCoupon();
            checkCouponAccepted();
    }
    public void testCheckoutWithExpiredCoupon() {
            applyExpiredCoupon();
            checkCouponRejected();
    }
    public void testCheckoutWithInvalidCoupon() {
            applyInvalidCoupon();
            checkCouponRejected();
    }
    //implementations of methods (e.g. login(), addItem2ToCart(), etc.)
}
```

We have now significantly reduced the size of each test, eliminated test code duplication and reduced future test code maintenance costs.

4. Multiple Failures

Definition

Create a mechanism to allow a customer test to continue executing after a non-critical failure.

Discussion

Traditionally, unit testing tools such as JUnit fail a test automatically after encountering the first failure. This does not impede unit testing, since most unit tests execute only a few steps where each is dependent on the previous one. Customer tests, however, often execute dozens of steps and can have steps that are independent of each other.

When testing a module that is buried deep within an application, there are many steps involved to reach the module of interest. If one of those initial steps fails, all subsequent steps would not be executed. This includes all test steps related to the module of interest. Bugs beyond this first failure can be masked within the application. This also introduces inefficiency since each bug must be found and fixed before the next one can be found. Finding multiple bugs at a time increases efficiency and allows bugs to be addressed in a priority sequence.

If the initial failure was due to incorrect information but that information is not required by the module of interest, then the test could potentially continue validly executing and performing additional steps. Thus, when it comes to customer tests, an approach is needed to allow multiple failures.

The Multiple Failures pattern, however, is more difficult to implement than the other patterns, since JUnit does not support multiple failures. In collaboration, the author has modified the JUnit testing framework to add this functionality.

Example

In this example, there is a problem with the specials page such that incorrect items are displayed. This page is mandatory in the navigation path of searching for and adding an item to your cart. Therefore, without a multiple failures mechanism, the adding function would not be tested because the test would fail and stop executing on the specials page. However, the specials page is not integral to adding an item to the cart. By allowing multiple failures, the specials page failure can be logged, but the test can proceed and test the function of adding an item to the cart. Figure 13 illustrates how the multiple failure mechanism is applied.

Figure 13: Setting up multiple failures mechanism.

```
import junit.framework.*;

public abstract class ShoppingCartTemplate extends TestCase {
        public ShoppingCartTemplate(String pName) { super(pName); }

        protected void setUp() {
                login();
                gotoSpecialsPage();    //navigate to the specials page
                setStopOnFail(false);  //JUnit extension - test will continue if check fails
                checkSpecialsPage();   //if this check fails, the test will not stop
                setStopOnFail(true);   //JUnit extension - test failure will now stop the test
                searchForItem1();
                cartAction1();         // <------ test hook 1
                searchForItem2();
                cartAction2();        //<-------- test hook 2
                logout();
        }
        abstract protected void cartAction1();
        abstract protected void cartAction2();
}
```

The setStopOnFail() method demarcates when to start or stop the multiple failure mechanism. Notice that we demarcated only the checking of the specials pages and not the navigation. This is because a problem with the application's navigation cannot be overcome by our test. On the other hand, a checking failure does not prevent us from navigating to other parts of the application and performing our add to cart test. By demarcating which parts of a test allow multiple failures, we can exercise more parts of the application.

III. Architectural Patterns

Architectural patterns dictate the organization of test code into a framework. This group consists of the Three-Tier Testing Architecture.

1. Three-Tier Testing Architecture

Definition

Create three layers of test code: Tool Layer, Application Testing Interface Layer and Test Case Layer.

Discussion

It is a well established idea in distributed object architecture to divide an application into separate tiers. The first tier encapsulates the presentation logic, the second tier the business logic, and the third tier the data storage logic.[12] Using this paradigm, application maintenance costs are reduced since components inside each tier can change without impacting other tiers. The same principle can be applied to customer testing architecture.

Test code can be separated into three layers: the Tool Layer, the Application Testing Interface (ATI) Layer and the Test Case Layer. Each layer has a specific responsibility, with the overall goal of reducing the maintenance costs of tests and facilitating new test creation.

Tool Layer

The Tool Layer's responsibility is to provide an extensive Application Programming Interface (API) to facilitate test writing. For instance, the tool can allow tests to find and interact with objects such as buttons, forms, and tables. Each tool is generic such that it can be used to write tests for various applications of a specific type (e.g. web, .NET or Java). Some tools have capabilities to test applications of several types.

Test tools usually fall into two types: record-playback (e.g. QuickTest[13]) and programmer-oriented. Using a record-playback tool, test developers can quickly start recording tests and playing them back. However, the code that is generated by the tool is usually not object-oriented and uses a proprietary language. Tools that are programmer-oriented are meant to be used by someone with programming experience. There is usually no record and playback mechanism because test developers are expected to write the tests themselves. The advantage of these tools is that they use a standard object-oriented language, which makes them very suitable to using design patterns.

Application Testing Interface (ATI) Layer

The ATI Layer[14] is responsible for consolidating all functions common to multiple tests. Every application will have a specific ATI Layer. For instance, each application is sufficiently unique as to require different navigations and assertions. Thus, the common ones contained in this layer would vary between applications. The ATI Layer is in effect a utility service provider for individual tests. It offloads the majority of the work that each test would otherwise be required to do.

The Template, Domain Test Object, Object Genie, and Transporter patterns are all examples of ATI layer code. The ATI Layer uses the tool (from the Tool Layer) to implement each pattern's details. If the tool or the application changes, then the corresponding ATI code would need to be updated to accommodate the changes. The ATI layer is the primary factor in lowering maintenance costs, since a well designed ATI will encapsulate many common navigations and checks that are used by test cases in the Test Case Layer.

Test Layer

The purpose of the Test Layer is to outline specific inputs and expected outputs for each test. The Test Layer relies heavily on the ATI Layer and as a result, is quite brief. Ideally, the Test Layer contains only data unique to each test. For this reason, the tests are insulated from changes in the application. Examples of Test Layer code are the subclasses of a Template, and the tests that use an Object Genie, Transporter, or DTO.

Conclusion

Applications are constantly evolving, which poses serious maintenance problems for automated customer tests. Maintenance problems can be addressed by using design patterns and thus, treating test code with the same importance as application code. This paper presented a catalog of design patterns that demonstrate how to facilitate: test code reuse without duplication, test adaptability to application changes, creation of new tests, and maintenance of existing tests. To this end, the following nine design patterns were presented:

- Group 1: Basic Patterns
  - Template - consolidate sequence of steps to perform a test
  - Object Genie - consolidate object creation and initialization logic
  - Domain Test Object - consolidate knowledge of application's UI components
  - Transporter - consolidate navigation logic
- Group 2: General Design Principles
  - Dumb Test - minimize tests' knowledge of the application
  - Independent Test - maximize test independence from other tests
  - Don't Repeat Yourself - minimize duplication of code between tests
  - Multiple Failures - maximize the information obtained from each test
- Group 3: Architectural Patterns
  - Three-Tier Testing Architecture - separate test code into layers

The design patterns were categorized into three groups: Basic Patterns, General Design Principles, and Architectural Patterns. The basic patterns form the building blocks that the general design principles use to establish best practices. The architectural pattern dictates the organization of all test code into a reusable framework.

This paper has demonstrated how design patterns can be applied to increase the effectiveness of automated customer testing. This results in tests that consistently provide valuable and reliable feedback about an application. It is my hope that this information will be of benefit to others in their automated testing endeavors.

Acknowledgments

I would like to thank Kit Bradley and Susan Wolfman for their valuable feedback and guidance throughout this process. I would also like to thank Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, the authors of *Design Patterns* and Martin Fowler, the author of *Refactoring*, for showing me that software development can be elevated to an art form.

References

[1] Beck, Kent *Test Driven Development: By Example.* Addison-Wesley, 2002.
[2] JUnit Website, online at http://www.junit.org/
[3] Extreme Programming Software, online at http://www.xprogramming.com/software.htm
[4] Gamma, Erich, Richard Helm, Ralph Johnson and John Vlissides *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
[5] Fowler, Martin *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.
[6] Gamma, Erich, Richard Helm, Ralph Johnson and John Vlissides *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.
[7] Beck, Kent *Test Driven Development: By Example.* Addison-Wesley, 2002.
[8] Schuh, Peter, Stephanie Punke *ObjectMother: Easing Test Object Creation in XP.* XP Universe 2001, online at http://www.xpuniverse.com/2001/pdfs/Testing03.pdf
[9] Krasner, Glenn E., Stephen T. Pope *A cookbook for using the model-view controller user interface paradigm in Smalltalk-80.* Journal of Object-Oriented Programming, 1(3):26-49, August/September 1988.
[10] Fowler, Martin *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, 1999.
[11] Kaner, Cem, James Bach and Bret Pettichord *Lessons Learned in Software Testing: A Context Driven Approach.* Wiley, 2001.
[12] Monson-Haefel, Richard *Enterprise JavaBeans, Second Edition.* O'Reilly, 2000.
[13] Mercury Interactive QuickTest Professional, online at http://www.mercury.com/us/products/quality-center/functional-testing/quicktest-professional/
[14] Crispin, Lisa, Tip House *Testing Extreme Programming.* Addison-Wesley, 2002.