Cape Penisula University of Technology

TP 2 Notes and Exercise Manual for the Course

Technical Programming II Course Manual

Put Together by:
Boniface Kabaso

Reviewed by:
Kruben Naidoo

September 2012

Acknowledgements

The acknowledgements and the people to thank go here, ...

For/Dedicated to/To my…

Contents

List of Figures

List of Tables

Abbreviations

LAH	**L**ist **A**bbreviations **H**ere

Chapter 1

Development Infrastructure

1.1 Chapter Objectives

1. Demostrate the ability to set up a development infratructure
2. Understand the Source code version Systems available to software Development process
3. Understand IDEs and build systems used in the development of Software.

1.2 Development Infrastructure

The productivity of the software industry is defined by an infrastructure that helps the developers to produce quality software from the word go. The infrastructure has a lot of faces to it ranging from a simple IDE to complex platform like deployment cloud environment. This section introduces you to things you need to put in place if you want to develop quality software in a collaborative way or if you need to open source your personal project for the rest of the world community to help you code it much faster and provide the feedback and the quality required to be built into any software product.

1.2.1 Integragated Development Platform (IDEs)

Integrated Development Environments (IDE) provide benefits to programmers that plain text editors cannot match. IDEs can parse source code as it is typed, giving it a syntactic understanding of the code, help you with establishing coding standards and many more feature. This allows advanced features like code generators, auto-completion, refactoring, and debuggers. This course is going to use Netbeans for development, but note that there are other IDEs that can be used for executing the same task

1.2.2 Netbeans IDE

Netbeans is a free IDE backed by Oracle Corporation. Netbeans is built on a plugin architecture, and it has respectable third-party vendor support . The main advantage of Netbeans its excellent GUI designer. It includes syntax highlighting and language support for Java, JSP, XML/XHTML, visual design tools, code generators, ant, Maven2 and CVS, Git, Mercurila and Subversion support.

1.2.3 Eclipse

Eclipse is a free IDE that has taken the Java industry by storm. Built on a plugin architecture, Eclipse is highly extensible and customizable. Third-party vendors have embraced Eclipse and are increasingly providing Eclipse integration. Eclipse is built on its own SWT GUI library. Eclipse excels at refactoring, J2EE support, and plugin support. The only current weakness of Eclipse is its lack of a Swing or SWT GUI designer. The eclipse platform provides tool developers with ultimate flexibility and control over their software technology.

1.2.4 IntelliJ IDEA

IntelliJ IDEA is a commercial IDE with a loyal following that swear by it. It has excellent J2EE and GUI support. It is extensible via plugins. Its standout feature is the outstanding refactoring support. It provides a robust combination of enhanced development tools, including: refactoring, J2EE support, Ant, JUnit, and CVS integration. Packaged with an intelligent Java editor, coding assistance and advanced code automation tools, IDEA enables Java programmers to boost their productivity while reducing routine time consuming tasks.

Recently starting with version 9, they have launched an open source free IDE.

1.3 Source Code Management and Version Control

1.3.0.1 Subversion-Basic Concepts

This chapter is a short introduction to Subversion. We begin with a discussion of general version control concepts, work our way into the specific ideas behind Subversion, and show some simple examples of Subversion in use.

Even though the examples here show people sharing collections of program source code, keep in mind that Subversion can manage any sort of file collection. It is not limited to helping computer programmers.

1.3.0.2 The Repository

Subversion is a centralized system for sharing information. At its core is a repository, which is a central store of data. The repository stores information in the form of a filesystem, treea typical hierarchy of files and directories. Any number of clients connect to the repository, and then read or write to these files. By writing data, a client makes the information available to others; by reading data, the client receives information from others.

So why is this interesting? So far, this sounds like the definition of a typical file server. What makes the Subversion repository special is that it remembers every change ever written to it: every change to every file, and even changes to the directory tree itself, such as the addition, deletion, and rearrangement of files and directories.

When a client reads data from the repository, it normally sees only the latest version of the filesystem tree. But the client also has the ability to view previous states of the filesystem. For example, a client can ask historical questions like,

1. What did this directory contain last Wednesday?
2. Who was the last person to change this file, and what changes did they make?

These are the sorts of questions that are at the heart of any version control system: systems that are designed to record and track changes to data over time.

1.3.0.3 Versioning Models

The core mission of a version control system is to enable collaborative editing and sharing of data. But different systems use different strategies to achieve this.

1.3.0.4 The Problem of File-Sharing

All version control systems have to solve the same fundamental problem: how will the system allow users to share information, but prevent them from accidentally stepping on each other's feet? It's all too easy for users to accidentally overwrite each other's changes in the repository.

Consider this scenario. Suppose we have two co-workers, Harry and Sally. They each decide to edit the same repository file at the same time. If Harry saves his changes to the repository first, then it's possible that (a few moments later) Sally could accidentally overwrite them with her own new version of the file. While Harry's version of the file won't be lost forever (because the system remembers every change), any changes Harry made won't be present in Sally's newer version of the file, because she never saw Harry's changes to begin with. Harry's work is still effectively lost or at least missing from the latest version of the file and probably by accident. This is definitely a situation we want to avoid!

1.3.0.5 The Lock-Modify-Unlock Solution

Many version control systems use a lock-modify-unlock model to address this problem. In such a system, the repository allows only one person to change a file at a time. First Harry must lock the file before he can begin making changes to it. Locking a file is a lot like borrowing a book from the library; if Harry has locked a file, then Sally cannot make any changes to it. If she tries to lock the file, the repository will deny the request. All she can do is read the file, and wait for Harry to finish his changes and release his lock. After Harry unlocks the file, his turn is over, and now Sally can take her turn by locking and editing.

The problem with the lock-modify-unlock model is that it's a bit restrictive, and often becomes a roadblock for users:

- Locking may cause administrative problems. Sometimes Harry will lock a file and then forget about it. Meanwhile, because Sally is still waiting to edit the file, her hands are tied. And then Harry goes on vacation. Now Sally has to get an administrator to release Harry's lock. The situation ends up causing a lot of unnecessary delay and wasted time.
- Locking may cause unnecessary serialization. What if Harry is editing the beginning of a text file, and Sally simply wants to edit the end of the same file? These changes don't overlap at all. They could easily edit the file simultaneously, and no great harm would come, assuming the changes were properly merged together. There's no need for them to take turns in this situation.
- Locking may create a false sense of security. Pretend that Harry locks and edits file A, while Sally simultaneously locks and edits file B. But suppose that A and B depend on one another, and the changes made to each are semantically incompatible. Suddenly A and B don't work together anymore. The locking system was powerless to prevent the problem yet it somehow provided a false sense of security. It's easy for Harry and Sally to imagine that by locking files, each is beginning a safe, insulated task, and thus not bother discussing their incompatible changes early on.

1.3.0.6 The Copy-Modify-Merge Solution

Subversion, CVS, and other version control systems use a copy-modify-merge model as an alternative to locking. In this model, each user's client contacts the project repository and creates a personal working copya local reflection of the repository's files and directories. Users then work in parallel, modifying their private copies. Finally, the private copies are merged together into a new, final version. The version control system often assists with the merging, but ultimately a human being is responsible for making it happen correctly.

Here's an example. Say that Harry and Sally each create working copies of the same project, copied from the repository. They work concurrently, and make changes to the same file A within their copies. Sally saves her changes to the repository first. When Harry attempts to save his changes later, the repository informs him that his file A is out-of-date. In other words, that file A in the repository has somehow changed since he last copied it. So Harry asks his client to merge any new changes from the repository into his working copy of file A. Chances are that Sally's changes don't overlap with his own; so once he has both sets of changes integrated, he saves his working copy back to the repository.

But what if Sally's changes do overlap with Harry's changes? What then? This situation is called a conflict, and it's usually not much of a problem. When Harry asks his client to merge the latest repository changes into his working copy, his copy of file A is somehow flagged as being in a state of conflict: he'll be able to see both sets of conflicting changes, and manually choose between them. Note that software can't automatically resolve conflicts; only humans are capable of understanding and making the necessary intelligent choices. Once Harry has manually resolved the overlapping changes, perhaps after a discussion with Sally, he can safely save the merged file back to the repository.

The copy-modify-merge model may sound a bit chaotic, but in practice, it runs extremely smoothly. Users can work in parallel, never waiting for one another. When they work on the same files, it turns out that most of their concurrent changes don't overlap at all; conflicts are infrequent. And the amount of time it takes to resolve conflicts is far less than the time lost by a locking system.

In the end, it all comes down to one critical factor: user communication. When users communicate poorly, both syntactic and semantic conflicts increase. No system can force users to communicate perfectly, and no system can detect semantic conflicts. So there's no point in being lulled into a false promise that a locking system will somehow prevent conflicts; in practice, locking seems to inhibit productivity more than anything else. Subversion in Action

1.3.0.7 Working Copies

You've already read about working copies; now we'll demonstrate how the Subversion client creates and uses them.

A Subversion working copy is an ordinary directory tree on your local system, containing a collection of files. You can edit these files however you wish, and if they're source code files, you can compile your program from them in the usual way. Your working copy is your own private work area: Subversion will never incorporate other people's changes, nor make your own changes available to others, until you explicitly tell it to do so.

After you've made some changes to the files in your working copy and verified that they work properly, Subversion provides you with commands to **publish** your changes to the other people working with you on your project (by writing to the repository). If other people publish their own changes, Subversion provides you with commands to merge those changes into your working directory (by reading from the repository).

A working copy also contains some extra files, created and maintained by Subversion, to help it carry out these commands. In particular, each directory in your working copy contains a subdirectory named **.svn**, also known as the working copy administrative directory. The files in each administrative directory help Subversion recognize which files contain unpublished changes, and which files are out-of-date with respect to others' work.

A typical Subversion repository often holds the files (or source code) for several projects; usually, each project is a subdirectory in the repository's filesystem tree. In this arrangement, a user's working copy will usually correspond to a particular subtree of the repository.

1.4 Netbeans Support for Subversion

Check this link for Subversion support in Netbeans

http://netbeans.org/kb/docs/ide/subversion.html

1.5 Software Debugging

A practicing programmer inevitably spends a lot of time tracking down and fixing bugs. Debugging, particularly debugging of other people's code, is a skill separate from the ability to write programs in the first place. Unfortunately, while debugging is often practiced, it is rarely taught. A typical course in debugging techniques consists merely of reading the manual for a debugger.

Debugging is a methodical process of finding and reducing the number of bugs, or defects, in a program or a piece of code

NetBeans IDE provide a debugging tool that will allow to trace the execution of your programs you can debug by setting breakpoints, watches in your code and run your application in the debugger. Using the debugger tool you can execute your program line by line and examine the value within your application to locate logical errors in your java programs.

A breakpoint is a mark in the source code that indicates the debugger to stop when the execution reach it. When your program stops on a breakpoint, you can inspect the current values of the variables in your program or continue the execution of your program, one line at a time.

You can monitor the values of variables or expressions during the execution of your program. An easy way to inspect the value of a variable during step-by-step execution of your program is to hover your mouse over the variable, the debugger will display the value of the variable close to where the cursor is placed. Another way to examine the value of a variable is by adding a watch on the variable. To add a watch to a variable or expression, select the variable or expression you want to monitor, then you can do one of the following:

1. Select New Watch in the Run menu.
2. The New Watch window pop up, click OK to add the variable to the Watch list.
3. Right Click and click on New Watch. Click on the OK button in the New Watch Window to add the variable to the watch list.
4. The Watches window displays the list of current watches, their type and values.

If the Watch window is not visible, choose Debugging¿Watches in the Windows menu . To remove a watch from the list, right click on the watch you want to remove and select Delete. There are several ways to start the execution of your java program in debug mode.

1. Choose Debug Main Project in the Run menu to execute the program to the the first breakpoint. If you did not set a breakpoint, the program will run until it terminates.
2. Choose Step Into in the Run menu to run your program to the first line of the main method of your program. At this point you can examine the values of variables and continue the execution of your program.
3. Choose Run to Cursor in the Run menu to execute your program until the line in the source code where the cursor is currently located.

Once the execution of your program has stopped, you can trace the execution of your program using the following options:

- Step Over:Executes one line of code. If the line is a call to a method, executes the method without stepping into the method's code.
- Step Into:Executes one line of code. If the line is a call to a method, step into the method and stop on the first line of the method.
- Step Out:Executes one line of code. However, if the source code line is in a method, it will execute the remaining source code in the method and returns to the caller of the method.
- Run to Cursor:Executes the program to the line where the cursor is located.
- Continue:Continue the execution of program until the next breakpoint or until the program terminates.

The local variables displays the name, data type and values of all the values in the current scope as well as static member variables among others. To activate the Local Variables window, select **Debugging , Local Variables** in the Windows menu. The debugger allows you to change the value of a variable in the Local Variable window and then continue the execution of your program using that new variable's value.

To stop the debugger, select **Finish** Debugger Session in the Run menu ending the execution of your program in debug mode.

1.6 Coding Standards

Code Convention is important to make you code readable and understandable. Fortunately Netbeans will give you the hints required to use for your work. More will be shown later as we progress through the course for now we shall follow this convention

Class names start with Capital Letters and the rest of the name is small

All Methods and variables will use Camel Casing

Package names will all be in be in small letters, except a few speacial packages .eg Impl

variable names have to me meaningful. Avoid variables like d, t, c as they do not carry any meaning.

1.7 Continuous Integration

Continuous integration (CI) is a set of practices intended to ease and stabilize the process of creating software builds. CI assists development teams with the following challenges:

- Software build automation: With CI, you can launch the build process of a software artifact at the push of a button, on a predefined schedule, or in response to a specified event. If you want to build a software artifact from source, your build process is not bound to a specific IDE, computer, or person.
- Continuous automated build verification: A CI system can be configured to constantly execute builds as new or modified source code is checked in. This means that while a team of software developers periodically checks in new or modified code, the CI system continuously verifies that the build is not being broken by the new code. This reduces the need for developers to check with each other on changes to interdependent components.
- Continuous automated build testing: An extension of build verification, this process ensures that new or modified code does not cause a suite of predefined tests on the built artifacts to fail. In both build verification and testing, failures can trigger notifications to interested parties, indicating that a build or some tests have failed.
- Post-build procedure automation: The build lifecycle of a software artifact may also require additional tasks that can be automated once build verification and testing are complete, such as generating documentation, packaging the software, and deploying the artifacts to a running environment or to a software repository. In this way artifacts can quickly be made available to users.

To implement a CI server you need, at minimum, an accessible source code repository (and the source code in it), a set of build scripts and procedures, and a suite of tests to execute against the built artifacts.

Chapter 2

Testing, Test Driven Development and Source Code Management

2.1 Chapter Objectives

1. Explain and demonstrate Unit and integration testing
2. Use the important technique called Test-Driven Development, which enables developers to write the tests that prove their code is correct before they write their code.
3. Learn tools, and patterns for using testing to deliver good, defect free code as fast as possible.
4. Explain and use version control and code management in application development

2.2 Introduction To Testing

There are lots of different kinds of testing that can and should be performed on a software project. Some of this testing requires extensive involvement from the end users; other forms may require teams of dedicated Quality Assurance personnel or other expensive resources.

But that's not what we're going to talk about in this module. Instead, we're talking about unit testing: an essential, if often misunderstood, part of project and personal success.

Unit testing is a relatively inexpensive, easy way to produce better code, faster. A unit test is a piece of code written by a developer that exercises a very small, specific functionality of the code being tested. Usually a unit test exercises some particular method in a particular context. Unit testing will make your life easier. It will make your designs better and drastically reduce the amount of time you spend debugging.

When writing test code, there are some naming conventions you need to follow. If you have a method named **createAccount** that you want to test, then your first test method might be named **testCreateAccount**. The method **testCreateAccount** will call **createAccount** with the necessary parameters and verify that **createAccount** works as advertised.You can, of course, have many test methods that exercise **createAccount**. If you have a class called **Calculator** that you need to unit test, you would call the class **CalculatorTest**

The test code must be written to do a few things:

1. Setup all conditions needed for testing (create any required objects, allocate any needed resources, etc.)
2. Call the method to be tested
3. Verify that the method to be tested functioned as expected
4. Clean up after itself

You write test code and compile it in the normal fashion, as you would any other bit of source code in your project.

As we've seen, there are some helper methods that assist you in determining whether a method under test is performing

There are some helper methods that assist you in determining whether a method under test is performing correctly or not. Generically, we call all these methods *asserts*. They let you assert that some condition is true; that two bits of data are equal, or not, and so on. We'll take a look at each one of the assert methods that **JUnit** provides next.

2.2.1 Important Unit Testing Methods

In this course we shall be using JUnit 4 as a Testing Framework. JUnit 4 uses Java 5 annotations to completely eliminate both of these conventions. The class hierarchy is no longer required and methods intended to function as tests need only be decorated with a newly defined @Test annotation. Java 5 annotations make JUnit 4 a notably different framework from previous versions. Declaring a test in JUnit 4 is a matter of decorating a test method with the @Test annotation.

2.2.1.1 Fixtures

Sometimes the setup can be more complex than you want to put in a constructor or a field initializer Sometimes you want to reinitialize static data Sometimes you just want to share setup code between different methods.

Fixtures foster reuse through a contract that ensures that particular logic is run either before or after a test. In older versions of JUnit, this contract was implicit regardless of whether you implemented a fixture or not. JUnit 4, however, has made fixtures explicit through annotations, which means the contract is only enforced if you actually decide to use a fixture.

Through a contract that ensures fixtures can be run either before or after a test, you can code reusable logic.

This logic, for example, could be initializing a class that you will test in multiple test cases or even logic to populate a database before you run a data-dependent test. Either way, using fixtures ensures a more manageable test case: one that relies on common logic.

Fixtures come in especially handy when you are running many tests that use the same logic and some or all of them fail. Rather than sifting through each test's set-up logic, you can look in one place to deduce the cause of failure. In addition, if some tests pass and others fail, you might be able to avoid examining the fixture logic as a source of the failures altogether. JUnit 4 uses annotations to cut out a lot of the overhead of fixtures, allowing you to run a fixture for every test or just once for an entire class or not at all. There are four fixture annotations: two for class-level fixtures and two for method-level ones. At the class level, you have **@BeforeClass** and **@AfterClass**, and at the method (or test) level, you have **@Before** and **@After.**

```
package za.co.kijali.test.repository;

/**
*
* @author boniface
*/
public class NewEmptyJUnitTest {
public NewEmptyJUnitTest() {
}
@BeforeClass
public static void setUpClass() throws Exception {
}
@AfterClass
public static void tearDownClass() throws Exception {
}
@Before
public void setUp() {
}
@After
public void tearDown() {
}
// TODO add test methods here.
// d with annotation @Test. For example:
@Test
public void hello() {
}
}
```

LISTING 2.1: Testing Code

2.2.1.2 Freeing resources after each test

user the tearDown methods, rarely used.

```

@AfterClass
public static void tearDownClass() throws Exception {
}
@After
public void tearDown() {
}
```

LISTING 2.2: Testing Code

2.2.1.3 Assertion Messages

First argument to each assertFoo method can be a string that's printed if the assertion fails:

```
public void testAdd() {
Complex z1 = new Complex(1, 1);
Complex z2 = new Complex(1, 1);
Complex z3 = z1.add(z2);
assertEquals("Addition failed in real part", 2, z3.getRealPart());
assertEquals("Addition failed in imaginary part", 2, z3.getImaginaryPart());
}
```

Listing 2.3: Testing Code

2.2.1.4 Floating point assertions

As always with floating point arithmetic, you want to avoid direct equality comparisons.

```
public static void assertEquals(double expected, double actual, double tolerance)
public static void assertEquals(float expected, float actual, float tolerance)
public static void assertEquals(String message, double expected, double actual, double
tolerance)
public static void assertEquals(String message, float expected, float actual, float ↩
     tolerance)
```

Listing 2.4: Testing Code

2.2.1.5 Integer assertions

Straight-forward because integer comparisons are straight-forward:

```
public static void assertEquals(boolean expected, boolean actual)
public static void assertEquals(byte expected, byte actual)
public static void assertEquals(char expected, char actual)
public static void assertEquals(short expected, short actual)
public static void assertEquals(int expected, int actual)
public static void assertEquals(long expected, long actual)
public static void assertEquals(String message, boolean expected, boolean actual)
public static void assertEquals(String message, byte expected, byte actual)
public static void assertEquals(String message, char expected, char actual)

public static void assertEquals(String message, short expected, short actual)
public static void assertEquals(String message, int expected, int actual)
public static void assertEquals(String message, long expected, long actual)
```

Listing 2.5: Testing Code

2.2.1.6 Object comparisons: asserting object equality

uses equals methods

```
assertEquals()
```

Listing 2.6: Testing Code

2.2.1.7 Object comparisons: asserting object identity

use the == operator:

```
assertSame ()
assertNotSame()
```

Listing 2.7: Testing Code

2.2.1.8 Asserting Truth

For general boolean tests, one can assert that some condition is true

```
public static void assertTrue(boolean condition)
public static void assertTrue(String message, boolean condition)
```

Listing 2.8: Testing Code

2.2.1.9 Asserting Falsity

Can also assert that some condition is false

```
public static void assertFalse(boolean condition)
public static void assertFalse(String message, boolean condition)
```

Listing 2.9: Testing Code

2.2.1.10 Asserting Nullness

Testing for the presence of Null Values

```
public static void assertNull(Object o)
public static void assertNull(String message, Object o)
```

Listing 2.10: Testing Code

2.2.1.11 Asserting Non-Nullness

Testing for the presence of Null Values

```
public static void assertNotNull(Object o)
public static void assertNotNull(String message, Object o)
```

Listing 2.11: Testing Code

2.2.1.12 Deliberately failing a test

Make a Test fail on purpose to just eliminate false pass

```
public static void fail()
public static void fail(String message)
```

Listing 2.12: Testing Code

2.2.1.13 Testing for exceptions

It's usually a good idea to specify that your test throws Exception. The only time you want to ignore this rule is if you're trying to test for a particular exception. If a test throws an exception, the framework reports a failure. If you'd actually like to test for a particular exception, JUnit 4's @Test annotation supports an expected parameter, which is intended to represent the exception type the test should throw upon execution. A simple comparison demonstrates what a difference the new parameter makes.

```
@Test(expected=IndexOutOfBoundsException.class)
public void verifyZipCodeGroupException() throws Exception{
Matcher mtcher = this.pattern.matcher("22101-5051");
boolean isValid = mtcher.matches();
mtcher.group(2);
}
```

Listing 2.13: Testing Code

2.2.1.14 Testing with timeouts

In JUnit 4, a test case can take a timeout value as a parameter. The timeout value represents the maximum amount of time the test can take to run: if the time is exceeded, the test fails. Testing with timeouts is easy: Simply decorate a method with @Test followed by a timeout value and you've got yourself an automated timeout test!

```
@Test(timeout=1)
public void verifyFastZipCodeMatch() throws Exception{
Pattern pattern = pattern.compile("^\\d{5}([\\-]\\d{4})?$");
Matcher mtcher = pattern.matcher("22011");
boolean isValid = mtcher.matches();
assertTrue("Pattern did not validate zip code", isValid);
}
```

Listing 2.14: Testing Code

2.2.1.15 Ignoring tests

JUnit 4 has introduced an annotation called @Ignore, which forces the framework to ignore a particular test method. You can also pass in a message documenting your decision for unsuspecting developers who happen upon the ignored test

```
@Ignore("this regular expression isn't working yet")
@Test
public void verifyZipCodeMatch() throws Exception{
Pattern pattern = Pattern.compile("^\\d{5}([\\-]\\d{4})");
Matcher mtcher = pattern.matcher("22011");
boolean isValid = mtcher.matches();
assertTrue("Pattern did not validate zip code", isValid);
}
```

Listing 2.15: Testing Code

2.2.1.16 Assert Arrays Content

You can compare the contents of arrays in with JUnit 4

```
@Test
public void verifyArrayContents() throws Exception{
String[] actual = new String[] {"JUnit 3.8.x", "JUnit 4", "TestNG"};
String[] var = new String[] {"JUnit 3.8.x", "JUnit 4.1", "TestNG 5.5"};
assertEquals("the two arrays should not be equal", actual, var);
}
```

Listing 2.16: Testing Code

2.2.2 Creating Unit Test with Netbeans

In a Big project a programmer might need to test each component of our program independently from the rest of our program. In Java this is supported by JUnit Test cases. In the design phase of the project every method is supposed to do some particular job and the developer is also supposed to come up with a set of tests to ensure that the method is functioning properly. JUnit provides a framework to take each one of the methods and perform individual tests so that we can ensure that we are getting proper values.

NetBeans IDE provides an easy way to support JUnit testing .Here is a tutorial from netbeans.org on how to create JUnit tests in netbeans

2.2.2.1 Creating the Project

1. Choose File then New Project from the main menu.
2. Select Java Class Library from the Java category and click Next.
3. Type JUnit-Sample for the project and set the project location.
4. Deselect the Use Dedicated Folder option, if selected and click finish

After you create the project, if you look in the Test Libraries node in the Projects window you can see that the project contains JUnit 4 libraries. The IDE adds both libraries to new projects by default. The first time that you create a JUnit test the IDE prompts you to select a version and then removes the library that is not needed.

2.2.2.2 Downloading the Solution Project

You can download the sample project **JUnitSampleSol** used in this tutorial by going to WEBCT and downloading the file **JUnitSampleSol.zip.** in the sample code folder

2.2.2.3 Creating the Java Classes

In this exercise you copy the files **Utils.java** and **Vectors.java** from the sample project **JUnitSampleSol** into the class library project that you created.

1. In the Projects window, right-click the Source Packages node of your project JUnit-Sample and choose New ¿ Java Package from the popup menu.
2. Type sample as the package name. Click Finish.
3. Open the JUnitSampleSol project in the IDE and expand the project node in the Projects window.
4. Copy **Utils.java** and **Vectors.java** in the Source Packages folder of the **JUnit-SampleSol** project into the sample source package in JUnit-Sample.

If you look at the source code for the classes, you can see that **Utils.java** has three methods (computeFactorial, concatWords, and normalizeWord) and that **Vectors.java** has two methods (equals and scalarMultiplication). The next step is to create test classes for each class and write some test cases for the methods.

2.2.3 Writing JUnit 4 Tests

In this exercise you create JUnit 4 unit tests for the classes **Vectors.java** and **Utils.java**. The JUnit 4 test cases are the same as the JUnit 3 test cases, but you will see that the syntax for writing the tests is simpler.

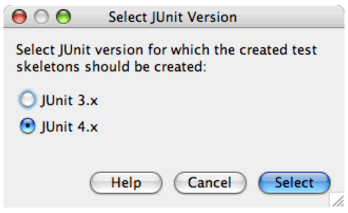
You will use the IDE's create test skeletons based on the classes in your project. The first time that you use the IDE to create some test skeletons for you, the IDE prompts you to choose the JUnit version.

2.2.3.1 Creating a Test Class for Vectors.java

In this exercise you will create the JUnit test skeletons for Vectors.java.

1. Right-click Vectors.java and choose Tools ¿ Create JUnit Tests.

2. Select JUnit 4.x in the Select JUnit Version dialog box.



Select JUnit Version

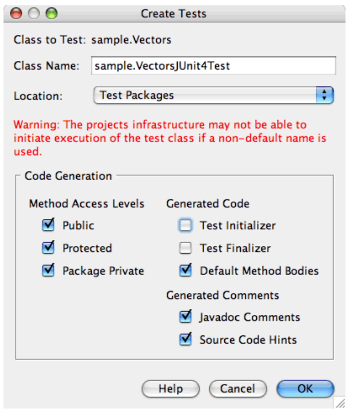
Select JUnit version for which the created test skeletons should be created:

JUnit 3.x [✓ JUnit 4.x]

3. Modify the name of the test class to VectorsJUnit4Test in the Create Tests dialog.

   When you change the name of the test class, you will see a warning about changing the name. The default name is based on the name of the class you are testing, with the word Test appended to the name. For example, for the class MyClass.java, the default name of the test class is MyClassTest.java.

4. Deselect Test Initializer and Test Finalizer. Click OK



Create Tests

Class to Test: sample.Vectors

Class Name: sample.VectorsJUnit4Test

Location: Test Packages

Warning: The projects infrastructure may not be able to initiate execution of the test class if a non-default name is used.

Code Generation

Method Access Levels: [✓ Public] [✓ Protected] [✓ Package Private]

Generated Code: Test Initializer, Test Finalizer, [✓ Default Method Bodies]

Generated Comments: [✓ Javadoc Comments] [✓ Source Code Hints]

When you click OK, the IDE creates a JUnit test skeleton in the sample test package directory.

A project requires a directory for test packages to create tests. The default location for the test packages directory is at the root level of the project, but you can specify a different location for the directory in the project's Properties dialog.

If you look at VectorsJUnit3Test.java in the editor, you can see that the IDE generated the test methods testEqual and testScalarMultiplication. In JUnit 4, each test method is annotated with @Test. The IDE generated the names for the test methods based on the names of the method in Vectors.java but the name of the test method is not required to have test prepended. The default body of each generated test method is provided solely as a guide and needs to be modified to be actual test cases.

You can deselect Default Method Bodies in the Create Tests dialog if you do not want the bodies of the method generated for you.

The IDE also generated the following test class initializer and finalizer methods:

```
@BeforeClass
public static void setUpClass() throws Exception {
}

@AfterClass
public static void tearDownClass() throws Exception {
}
```

LISTING 2.17: Sample code

The IDE generates the class initializer and finalizer methods by default when creating JUnit 4 test classes. The annotations **@BeforeClass** and **@AfterClass** are used to mark methods that should be run before and after running the test class. You can delete the methods because you will not need them to test Vectors.java.

You can configure the methods that are generated by default by configuring the JUnit options in the Options window.

2.2.3.2 Writing Test Methods for Vectors.java

In this exercise you modify each of the generated test methods to test the methods using the JUnit assert method and to change the names of the test methods. In JUnit 4 you have greater flexibility when naming test methods because test methods are indicated by the @Test annotation and do not require the word test prepended to test method names.

1. Open VectorsJUnit4Test.java in the editor.
2. Modify the test method for testScalarMultiplication by changing the name of the method, the value of the println and removing the generated variables. The test method should now look like the following (changes displayed in bold):

```
    @Test
public void ScalarMultiplicationCheck() {
    System.out.println("* VectorsJUnit4Test: ScalarMultiplicationCheck()");
    assertEquals(expResult, result);
}
```

LISTING 2.18: Sample code

When writing tests it is not necessary to change the printed output. You do this in this exercise so that it is easier to identify the test results in the output window.

3. Now add some assertions to test the method.

```
    @Test
public void ScalarMultiplicationCheck() {
    System.out.println("* VectorsJUnit4Test: ScalarMultiplicationCheck()");
    assertEquals(  0, Vectors.scalarMultiplication(new int[] { 0, 0}, new int[] { 0, ↩
    0}));
    assertEquals( 39, Vectors.scalarMultiplication(new int[] { 3, 4}, new int[] { 5, ↩
    6}));
    assertEquals(-39, Vectors.scalarMultiplication(new int[] {-3, 4}, new int[] { ↩
    5,-6}));
    assertEquals(  0, Vectors.scalarMultiplication(new int[] { 5, 9}, new int[] {-9, ↩
    5}));
    assertEquals(100, Vectors.scalarMultiplication(new int[] { 6, 8}, new int[] { 6, ↩
    8}));
}
```

LISTING 2.19: Sample code

In this test method you use the JUnit assertEquals method. To use the assertion, you supply the input variables and the expected result. To pass the test, the test method must successfully return all the expected results based on the supplied variables when running the tested method. You should add a sufficient number of assertions to cover the various possible permutations.

4. Change the name of the testEqual test method to equalsCheck.

5. Modify the equalsCheck test method by deleting the generated method body and adding the following println.

```
System.out.println("* VectorsJUnit4Test: equalsCheck()");
```

The test method should now look like the following:

```
@Test
public void equalsCheck() {
    System.out.println("* VectorsJUnit4Test: equalsCheck()");
}
```

6. Modify the equalsCheck method by adding the following assertions (displayed in bold).

```
   @Test
public void equalsCheck() {
    System.out.println("* VectorsJUnit4Test: equalsCheck()");
    assertTrue(Vectors.equal(new int[] {}, new int[] {}));
    assertTrue(Vectors.equal(new int[] {0}, new int[] {0}));
    assertTrue(Vectors.equal(new int[] {0, 0}, new int[] {0, 0}));
    assertTrue(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 0, 0}));
    assertTrue(Vectors.equal(new int[] {5, 6, 7}, new int[] {5, 6, 7}));

    assertFalse(Vectors.equal(new int[] {}, new int[] {0}));
    assertFalse(Vectors.equal(new int[] {0}, new int[] {0, 0}));
    assertFalse(Vectors.equal(new int[] {0, 0}, new int[] {0, 0, 0}));
    assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 0}));
    assertFalse(Vectors.equal(new int[] {0, 0}, new int[] {0}));
    assertFalse(Vectors.equal(new int[] {0}, new int[] {}));

    assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 0, 1}));
    assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {0, 1, 0}));
    assertFalse(Vectors.equal(new int[] {0, 0, 0}, new int[] {1, 0, 0}));
    assertFalse(Vectors.equal(new int[] {0, 0, 1}, new int[] {0, 0, 3}));
}
```

This test uses the JUnit assertTrue and assertFalse methods to test a variety of possible results. For the test of this method to pass, the assertTrue must all be true and assertFalse must all be false.

2.2.3.3 Creating a Test Class for Utils.java

You will now create the JUnit test methods for Utils.java. When you created the test class in the previous exercise, the IDE prompted you for the version of JUnit. You are not prompted to select a version this time because you already selected the JUnit version and all subsequent JUnit tests are created in that version.

1.

2. Right-click Utils.java and choose Tools ¿ Create JUnit Tests.

3. Select Test Initializer and Test Finalizer in the dialog box, if unselected.
4. Modify the name of the test class to UtilsJUnit4Test in the Create Tests dialog box. Click OK.

When you click OK, the IDE creates the test file UtilsJUnit4Test.java in the Test Packages ¿ samples directory. You can see that the IDE generated the test methods testComputeFactorial, testConcatWords, and testNormalizeWord for the methods in **Utils.java**. The IDE also generated initializer and finalizer methods for the test and the test class.

2.2.3.4 Writing Test Methods for Utils.java

In this exercise you will add test cases that illustrate some common JUnit test elements. You will also add a println to the methods because some methods do not print any output to the JUnit Test Results window to indicate that they were run, or to indicate that the method passed the test. By adding a println to the methods you can see if the methods were run and the order in which they were run.

Test Initializers and Finalizers

When you created the test class for **Utils.java** the IDE generated annotated initializer and finalizer methods. You can choose any name for the name of the method because there is no required naming convention.

In JUnit 4 you can use annotations to mark the following types of initializer and finalizer methods.

-
- Test Class Initializer. The **@BeforeClass** annotation marks a method as a test class initialization method. A test class initialization method is run only once, and before any of the other methods in the test class. For example, instead of creating a database connection in a test initializer and creating a new connection before each test method, you may want to use a test class initializer to open a connection before running the tests. You could then close the connection with the test class finalizer.
- Test Class Finalizer. The **@AfterClass** annotation marks a method as a test class finalizer method. A test class finalizer method is run only once, and after all of the other methods in the test class are finished.
- Test Initializer. The **@Before** annotation marks a method as a test initialization method. A test initialization method is run before each test case in the test class. A test initialization method is not required to run tests, but if you need to initialize some variables before you run a test, you use a test initializer method.
- Test Finalizer. The **@After** annotation marks a method as a test finalizer method. A test finalizer method is run after each test case in the test class. A test finalizer method is not required to run tests, but you may need a finalizer to clean up any data that was required when running the test cases.

Make the following changes (displayed in bold) to add a println to the initializer and finalizer methods.

```
@BeforeClass
public static void setUpClass() throws Exception {
    System.out.println("* UtilsJUnit4Test: @BeforeClass method");
}

@AfterClass
public static void tearDownClass() throws Exception {
    System.out.println("* UtilsJUnit4Test: @AfterClass method");
}

@Before
public void setUp() {
    System.out.println("* UtilsJUnit4Test: @Before method");
}

@After
public void tearDown() {
    System.out.println("* UtilsJUnit4Test: @After method");
}
```

When you run the test class the println text you added is displayed in the output pane of the JUnit Test Results window. If you do not add the println, there is no output to indicate that the initializer and finalizer methods were run.

Testing Using a Simple Assertion

This simple test case tests the concatWords method. Instead of using the generated test method testConcatWords, you will add a new test method called helloWorldCheck that uses a single simple assertion to test if the method concatenates the strings correctly. The assertEquals in the test case uses the syntax **assertEquals**$EXPECTED_{R}ESULT, ACTUAL_{R}ESULT$ to test if the expected result is equal to the actual result. In this case, if the input to the method concatWords is *"Hello", ",", "world" and "!"*, the expected result should equal **"Hello, world!".**

1. Delete the generated test method testConcatWords.

2. Add the following helloWorldCheck method to test Utils.concatWords.

```
@Test
public void helloWorldCheck() {
    assertEquals("Hello, world!", Utils.concatWords("Hello", ", ", "world", "!"));
}
```

3. Add a println statement to display text about the test in the JUnit Test Results window.

```
@Test
public void helloWorldCheck() {
    System.out.println("* UtilsJUnit4Test: test method 1 - helloWorldCheck()");
    assertEquals("Hello, world!", Utils.concatWords("Hello", ", ", "world", "!"));
```

Testing Using a Timeout

This test demonstrates how to check if a method is taking too long to complete. If the method is taking too long, the test thread is interrupted and the test fails. You can specify the time limit in the test.

The test method invokes the computeFactorial method in Utils.java. You can assume that the computeFactorial method is correct, but in this case you want to test if the computation is completed within 1000 milliseconds. You do this by interrupting the test thread after 1000 milliseconds. If the thread is interrupted the test method throws a TimeoutException.

1. Delete the generated test method testComputeFactorial.
2. Add the testWithTimeout method that calculates the factorial of a randomly generated number.

```
    @Test
    public void testWithTimeout() {
        final int factorialOf = 1 + (int) (30000 * Math.random());
        System.out.println("computing " + factorialOf + '!');
        System.out.println(factorialOf + "! = " + Utils.computeFactorial(factorialOf↩
  ));
    }
```

3. Add the following code (displayed in bold) to set the timeout and to interrupt the thread if the method takes too long to execute.

```
    @Test(timeout=1000)
    public void testWithTimeout() {
        final int factorialOf = 1 + (int) (30000 * Math.random());
```

You can see that the timeout is set to 1000 milliseconds.

4. Add the following println (displayed in bold) to print the text about the test in the JUnit Test Results window.

```
    @Test(timeout=1000)
    public void testWithTimeout() {
        System.out.println("* UtilsJUnit4Test: test method 2 - testWithTimeout()");
        final int factorialOf = 1 + (int) (30000 * Math.random());
        System.out.println("computing " + factorialOf + '!');
```

Testing for an Expected Exception

This test demonstrates how to test for an expected exception. The method fails if it does not throw the specified expected exception. In this case you are testing that the computeFactorial method throws an IllegalArgumentException if the input variable is a negative number (-5).

1. Add the following testExpectedException method that invokes the computeFactorial method with an input of -5.

```
    @Test
    public void checkExpectedException() {
        final int factorialOf = -5;
        System.out.println(factorialOf + "! = " + Utils.computeFactorial(factorialOf↩
  ));
    }
```

2. Add the following property (displayed in bold) to the @Test annotation to specify that the test is expected to throw IllegalArgumentException.

```
    @Test(expected=IllegalArgumentException.class)
    public void checkExpectedException() {
        final int factorialOf = -5;
        System.out.println(factorialOf + "! = " + Utils.computeFactorial(factorialOf↩
  ));
    }
```

3. Add the following println (displayed in bold) to print the text about the test in the JUnit Test Results window.

```
    @Test (expected=IllegalArgumentException.class)
    public void checkExpectedException() {
        System.out.println("* UtilsJUnit4Test: test method 3 - ↩
  checkExpectedException()");
        final int factorialOf = -5;
        System.out.println(factorialOf + "! = " + Utils.computeFactorial(factorialOf↩
  ));
    }
```

Disabling a Test

This test demonstrates how to temporarily disable a test method. In JUnit 4 you simply add the @Ignore annotation to disable the test.

1. Delete the generated test method testNormalizeWord.
2. Add the following test method to the test class.

```
    @Test
    public void temporarilyDisabledTest() throws Exception {
        System.out.println("* UtilsJUnit4Test: test method 4 - ↩
  checkExpectedException()");
        assertEquals("Malm\u00f6", Utils.normalizeWord("Malmo\u0308"));
    }
```

   The test method temporarilyDisabledTest will run if you run the test class.

3. Add the @Ignore annotation (displayed in bold) above @Test to disable the test.

```
    @Ignore
    @Test
    public void temporarilyDisabledTest() throws Exception {
        System.out.println("* UtilsJUnit4Test: test method 4 - ↩
  checkExpectedException()");
        assertEquals("Malm\u00f6", Utils.normalizeWord("Malmo\u0308"));
    }
```

4. Fix your imports to import org.junit.Ignore.

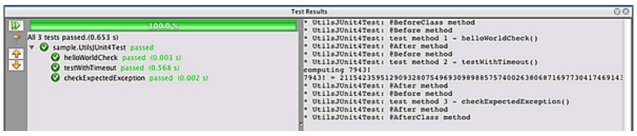
Now that you have written the tests you can run the test and see the test output in the JUnit Test Results window.

2.2.3.5 Running the Tests

You can run JUnit tests on the entire application or on individual files and see the results in the IDE. The easiest way to run all the unit tests for the project is to choose **Run**, then **Test** PROJECTNAME from the main menu. If you choose this method, the IDE runs all the test classes in the Test Packages. To run an individual test class, right-click the test class under the Test Packages node and choose Run File.

1. Right-click UtilsJUnit4Test.java in the Projects window.
2. Choose Test File.

When you run UtilsJUnit4Test.java the IDE only runs the tests in the test class. If the class passes all the tests you will see something similar to the following image in the JUnit Test Results window.



```
All 3 tests passed.(0.653 s)
▼ sample.UtilsJUnit4Test passed
    helloWorldCheck passed (0.003 s)
    testWithTimeout passed (0.568 s)
    checkExpectedException passed (0.002 s)
```

```
• UtilsJUnit4Test: @BeforeClass method
• UtilsJUnit4Test: @Before method
• UtilsJUnit4Test: test method 1 - helloWorldCheck()
• UtilsJUnit4Test: @After method
• UtilsJUnit4Test: @Before method
• UtilsJUnit4Test: test method 2 - testWithTimeout()
computing 7943!
7943! = 211542359512909328075496930989885757400263806871697730417469143
• UtilsJUnit4Test: @After method
• UtilsJUnit4Test: @Before method
• UtilsJUnit4Test: test method 3 - checkExpectedException()
• UtilsJUnit4Test: @After method
• UtilsJUnit4Test: @AfterClass method
```

In this image (click the image to see a larger image) you can see that the IDE ran the JUnit test on Utils.java and that the class passed all the tests. The left pane displays the results of the individual test methods and the right pane displays the test output. If you look at the output you can see the order that the tests were run. The println that you added to each of the test methods printed out the name of the test to Test Results window and the Output window.

You can see that in UtilsJUnit4Test the test class initializer method annotated with @BeforeClass was run before any of the other methods and it was run only once. The test class finalizer method annotated with @AfterClass was run last, after all the other methods in the class. The test initializer method annotated with @Before was run before each test method.

The controls in the left side of the Test Results window enable you to easily run the test again. You can use the filter to toggle between displaying all test results or only the failed tests. The arrows enable you to skip to the next failure or the previous failure.

When you right-click a test result in the Test Results window, the popup menu enables you to choose to go to the test's source, run the test again or debug the test.

2.2.4 Creating Test Suites

When creating tests for a project you will generally end up with many test classes. While you can run test classes individually or run all the tests in a project, in many cases you will want to run a subset of the tests or run tests in a specific order. You can do this by creating one or more test suites. For example, you can create test suites that test specific aspects of your code or specific conditions.

A test suite is basically a class with a method that invokes the specified test cases, such as specific test classes, test methods in test classes and other test suites. A test suite can be included as part of a test class but best practices recommends creating individual test suite classes.

You can create JUnit 3 and JUnit 4 test suites for your project manually or the IDE can generate the suites for you. When you use the IDE to generate a test suite, by default the IDE generates code to invoke all the test classes in the same package as the test suite. After the test suite is created you can modify the class to specify the tests you want to run as part of that suite.

2.2.4.1 Creating JUnit 4 Test Suites

If you selected JUnit 4 for the version of your tests, the IDE can generate JUnit 4 test suites. JUnit 4 is back-compatible so you can run JUnit 4 test suites that contain JUnit 4 and JUnit 3 tests. In JUnit 4 test suites you specify the test classes to include as values of the @Suite annotation.

1. Right-click the project node in the Projects window and choose New ¿ Other to open the New File wizard.
2. Select the JUnit category and Test Suite. Click Next.
3. Type JUnit4TestSuite for the file name.
4. Select the sample package to create the test suite in the sample folder in the test packages folder.
5. Deselect Test Initializer and Test Finalizer. Click Finish.

When you click Finish, the IDE creates the test suite class in the sample package and opens the class in the editor. The test suite contains the following code.

```
@RunWith(Suite.class)
@Suite.SuiteClasses(value={UtilsJUnit4Test.class, VectorsJUnit4Test.class})
public class JUnit4TestSuite {
}
```

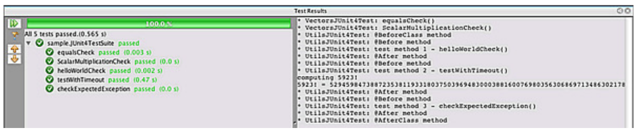
When you run the test suite the IDE will run the test classes UtilsJUnit4Test and VectorsJUnit4Test in the order they are listed.

2.2.4.2 Running Test Suites

You run a test suite the same way you run any individual test class.

1. Expand the Test Packages node in the Projects window.
2. Right-click the test suite class and choose Test File.

When you run the test suite the IDE runs the tests included in the suite in the order they are listed. The results are displayed in the JUnit Test Results window.



```
All 5 tests passed.(0.565 s)
▼ sample.JUnit4TestSuite passed
    equalsCheck passed (0.003 s)
    ScalarMultiplicationCheck passed (0.0 s)
    helloWorldCheck passed (0.002 s)
    testWithTimeout passed (0.47 s)
    checkExpectedException passed (0.0 s)
```

```
* VectorsJUnit4Test: equalsCheck()
* VectorsJUnit4Test: ScalarMultiplicationCheck()
* UtilsJUnit4Test: @BeforeClass method
* UtilsJUnit4Test: @Before method
* UtilsJUnit4Test: test method 1 - helloWorldCheck()
* UtilsJUnit4Test: @After method
* UtilsJUnit4Test: @Before method
* UtilsJUnit4Test: test method 2 - testWithTimeout()
computing 5923!
5923! = 529459847388723538119331803750396948300038816007698035630686971348630217
* UtilsJUnit4Test: @After method
* UtilsJUnit4Test: @Before method
* UtilsJUnit4Test: test method 3 - checkExpectedException()
* UtilsJUnit4Test: @After method
* UtilsJUnit4Test: @AfterClass method
```

2.3 Test Driven Development

2.3.1 Introduction

Test-driven development (TDD) is an advanced technique of using automated unit tests to drive the design of software and force decoupling of dependencies. The result of using this practice is a comprehensive suite of unit tests that can be run at any time to provide feedback that the software is still working.

The motto of Test-Driven Development is RED, GREEN, REFACTOR.

- RED : Create a test and make it fail.
- GREEN: Make the test pass by any means necessary.
- REFACTOR: Change the code to remove duplication in your project and to improve the design while ensuring that all tests still pass.

The /Red/Green/Refactor cycle is repeated very quickly for each new unit of code. Process Example

Follow these steps :

1. Understand the requirements of the story, work item, or feature that you are working on.
2. RED: Create a test and make it fail.
    - Imagine how the new code should be called and write the test as if the code already existed. You will not get IntelliSense because the new method does not yet exist.
    - Create the new production code stub. Write just enough code so that it compiles.
    - Run the test. It should fail. This is a calibration measure to ensure that your test is calling the correct code and that the code is not working by accident. This is a meaningful failure, and you expect it to fail.
3. GREEN: Make the test pass by any means necessary.
    - Write the production code to make the test pass. Keep it simple.

   - Some advocate the hard-coding of the expected return value first to verify that the test correctly detects success. This varies from practitioner to practitioner.
   - If you've written the code so that the test passes as intended, you are finished. You do not have to write more code speculatively. The test is the objective definition of "done." The phrase "You Ain't Gonna Need It" (YAGNI) is often used to veto unnecessary work. If new functionality is still needed, then another test is needed. Make this one test pass and continue.
   - When the test passes, you might want to run all tests up to this point to build confidence that everything else is still working.

4. REFACTOR: Change the code to remove duplication in your project and to improve the design while ensuring that all tests still pass.
   - Remove duplication caused by the addition of the new functionality.
   - Make design changes to improve the overall solution.
   - After each refactoring, rerun all the tests to ensure that they all still pass.
5. Repeat the cycle. Each cycle should be very short, and a typical hour should contain many RED/GREEN/REFACTOR cycles.

2.3.2 Benefits of Test-Driven Development

1. The suite of unit tests provides constant feedback that each component is still working.
2. The unit tests act as documentation that cannot go out-of-date, unlike separate documentation, which can and frequently does.
3. When the test passes and the production code is refactored to remove duplication, it is clear that the code is finished, and the developer can move on to a new test.
4. Test-driven development forces critical analysis and design because the developer cannot create the production code without truly understanding what the desired result should be and how to test it.
5. The software tends to be better designed, that is, loosely coupled and easily maintainable, because the developer is free to make design decisions and refactor at any time with confidence that the software is still working. This confidence is gained by running the tests. The need for a design pattern may emerge, and the code can be changed at that time.
6. The test suite acts as a regression safety net on bugs: If a bug is found, the developer should create a test to reveal the bug and then modify the production code so that the bug goes away and all other tests still pass. On each successive test run, all previous bug fixes are verified.
7. Reduced debugging time!

2.3.3 Characteristics of a Good Unit Test

A good unit test has the following characteristics.

1. Runs fast, runs fast, runs fast. If the tests are slow, they will not be run often.
2. Separates or simulates environmental dependencies such as databases, file systems, networks, queues, and so on. Tests that exercise these will not run fast, and a failure does not give meaningful feedback about what the problem actually is.
3. Is very limited in scope. If the test fails, it's obvious where to look for the problem. Use few Assert calls so that the offending code is obvious. It's important to only test one thing in a single test.
4. Runs and passes in isolation. If the tests require special environmental setup or fail unexpectedly, then they are not good unit tests. Change them for simplicity and reliability. Tests should run and pass on any machine. The "works on my box" excuse doesn't work.
5. Often uses stubs and mock objects. If the code being tested typically calls out to a database or file system, these dependencies must be simulated, or mocked. These dependencies will ordinarily be abstracted away by using interfaces.
6. Clearly reveals its intention. Another developer can look at the test and understand what is expected of the production code.

2.3.4 Summary of TDD and A Practical Example

The idea is simple is that No production code is written except to make a failing test pass. The Implications are that You have to write test cases before you write code This means that when you first write a test case, you may be testing code that does not exist And since that means the test case will not compile, obviously the test case fails After you write the skeleton code for the objects referenced in the test case, it will now compile, but also may not pass So, then you write the simplest code that will then make the test case pass

2.3.4.1 Practical Example

Consider writing a program to score the game of bowling You might start with the following test

```
public class TestGame extends TestCase {
public void testOneThrow() {
Game g = new Game();
g.addThrow(5);
assertEquals(5, g.getScore());
}
}
```

When you compile this program, it fails because the Game class does not yet exist

You would now write the Game class

```
public class Game {
public void addThrow(int pins) {
}
public int getScore() {
return 0;
}
}
```

The test code now compiles but the test will still fail. In Test-Driven Design, Beck, the proponent of the TDD philosophy, recommends taking small, simple steps So, we get the test case to compile before we get it to pass

Once we confirm that the test still fails, we would then write the simplest code to make the test case pass; that would be

```
public class Game {
public void addThrow(int pins) {
}
public int getScore() {
return 5;
}
}
```

The test case now passes!

2.3.4.2 TDD Life Cycle

The life cycle of test-driven development is Quickly add a test

1. Run all tests and see the new one fail
2. Make a simple change
3. Run all tests and see them all pass
4. Refactor to remove duplication

This cycle is followed until you have met your goal; note that this cycle simply adds testing to the add functionality; refactor loop of refactoring covered in the next lecture

perform TDD within a Testing Framework, such as JUnit, within such frameworks failing tests are indicated with a red bar passing tests are shown with a green bar. As such, the TDD life cycle is sometimes described as red bar/green bar/refactor

2.3.4.3 Principles of TDD

Testing List

keep a record of where you want to go; Experts keep two lists, one for current coding session and one for later; You wont necessarily finish everything in one go!

Test First

Write tests before code, because you probably wont do it after. Writing test cases gets you thinking about the design of your implementation; does this code structure make sense? what should the signature of this method be?

Assert First

How do you write a test case? By writing its assertions first! Suppose you are writing a client/server system and you want to test an interaction between the server and the client. Suppose that for each transaction, some string has to have been read from the server and that the socket used to talk to the server should be closed after the transaction

Lets write the test case

```
public void testCompleteTransaction {

assertTrue(reader.isClosed());
assertEquals(   abc    , reply.contents());
}
```

Now write the code that will make these asserts possible

2.3.5 Conclusion

Test-driven development is an advanced technique that uses unit tests to drive the design of software. Test-Driven Design is a mini software development life cycle that helps to organize coding sessions and make them more productive

1. Write a failing test case
2. Make the simplest change to make it pass
3. Refactor to remove duplication
4. Repeat!

2.4 Chapter Questions

1. Write 10 small programmes using the TDD method you have learnt. Please do not cheat yourself by
2. Write programmes first before writing tests. This exercise is meant to reconfigure your brain and wipe out all the bad habits you accumulated since the time you wrote your first program.

MAKE SURE THAT YOU PROGRAMMES TEST EACH OF THE ASPECTS MENTIONED IN THE SECTION 2.2.1 WITH TEST METHODS

MAKE SURE ALSO THAT YOU CODE TO AN INTERFACE AND NOT A CONCRETE CLASS

Chapter 3

Application Design Concepts and Principles

3.1 Chapter Objectives

1. Explain and demonstrate the main advantages of an object ori- ented approach to system design including the effect of encap- sulation, inheritance, delegation, and the use of interfaces, on architectural issues.
2. Describe how the principle of separation of concerns has been applied to the main system design
3. Describe and apply software design Concepts to system designs.

3.2 Fundamentals of Object Oriented Programming Concepts

There are three major features in object-oriented programming: **encapsulation**, **inheritance** and **polymorphism**.

3.2.1 Encapsulation

Encapsulation enforces modularity.

Encapsulation refers to the creation of self-contained modules that bind processing functions to the data. These user-defined data types are called classes, and one instance of a class is an object. In other words encapsulation means that the **attributes (data) and the behaviors (code) are encapsulated in to a single object.**

In other models, namely a structured model, code is in files that are separate from the data. An object, conceptually, combines the code and data into a single entity.

In programing at the class level, Encapsulation is the technique of making the fields in a class private and providing access to the fields via public methods. If a field is declared private, it cannot be accessed by anyone outside the class, thereby hiding the fields within the class.

For this reason, encapsulation is also referred to as data hiding. Encapsulation can be described as a protective barrier that prevents the code and data being randomly accessed by other code defined outside the class.

Access to the data and code is tightly controlled by an interface. The main benefit of encapsulation is the ability to modify our implemented code without breaking the code of others who use our code. With this feature Encapsulation gives maintainability, flexibility and extensibility to our code.

The code below shows a class which is encapsulated.

```java
public class Name implements Serializable {
    private static final long serialVersionUID = 1L;
    private String firstname;
    private String lastname;
    private String title;
    private String otherName;

    /**
     * @return the firstname
     */
    public String getFirstname() {
        return firstname;
    }

    /**
     * @param firstname the firstname to set
     */
    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    /**
     * @return the lastname
     */
    public String getLastname() {
        return lastname;
    }

    /**
     * @param lastname the lastname to set
     */
    public void setLastname(String lastname) {
        this.lastname = lastname;
    }

    /**
     * @return the title
     */
    public String getTitle() {
        return title;
    }

    /**
     * @param title the title to set
     */
    public void setTitle(String title) {
        this.title = title;
    }

    /**
     * @return the otherName
     */
    public String getOtherName() {
        return otherName;
    }

    /**
     * @param otherName the otherName to set
     */
    public void setOtherName(String otherName) {
        this.otherName = otherName;
    }
}
```

Benefits of Encapsulation include:

1. The fields of a class can be made read-only or write-only.

2. A class can have total control over what is stored in its fields.
3. The users of a class do not know how the class stores its data. A class can change the data type of a field, and users of the class do not need to change any of their code.

3.2.2 Inheritance

Inheritance passes knowledge down.

Inheritance can be defined as the process where one object acquires the properties of another. With the use of inheritance the information is made manageable in a hierarchical order. When we talk about inheritance the most commonly used keyword would be extends and implements. These words would determine whether one object **IS-A** type of another.

By using these keywords we can make one object acquire the properties of another object.

Usually classes are created in hierarchies, and inheritance allows the structure and methods in one class to be passed down the hierarchy. That means less programming is required when adding functions to complex systems.

If a step is added at the bottom of a hierarchy, then only the processing and data associated with that unique step needs to be added. Everything else about that step is inherited. The ability to reuse existing objects is considered a major advantage of object technology.

One of the major design issues in O-O programming is to factor out commonality of the various classes.

For example, say you have a Dog class and a Cat class, and each will have an attribute for eye color. In a procedural model, the code for Dog and Cat would each contain this attribute. In an O-O design, the color attribute can be abstracted up to a class called Mammal along with any other common attributes and methods.

Lets revisit Encapsulation: One of the primary advantages of using objects is that the object need not reveal all of its attributes and behaviors.

In good O-O design , an object should only reveal the interfaces needed to interact with it. Details not important to the use of the object should be hidden from other objects. This is called encapsulation. The interface is the fundamental means of communication between objects. Each class design specifies the interfaces for the proper instantiation and operation of objects. Any behavior that the object provides must be invoked by a message sent using one of the provided interfaces. The interface should completely describe how users of the class interact with the class.

In Java, the methods that are part of the interface are designated as public; everything else is part of the private implementation.

This is The Encapsulation Rule: Whenever the interface/implementation paradigm is covered, you are really talking about encapsulation. The basic question is what in a class should be exposed and what should not be exposed. This encapsulation pertains equally to data and behavior.

When talking about a class, the primary design decision revolves around encapsulating both the data and the behavior into a well-written class. In other words the process of packaging your program, dividing each of its classes into two distinct parts: the interface and the implementation is Encapsulation in the big picture.

Encapsulation is so crucial to O-O development that it is one of the generally accepted object-oriented designs cardinal rules.

Yet, Inheritance is also considered one of the four primary O-O concepts. However, in one way, inheritance actually breaks encapsulation!

How can this be? Is it possible that two of the three primary concepts of O-O are incompatible with each other?

There are three criteria that determine whether or not a language is object-oriented: encapsulation, inheritance, polymorphism.

To be considered a true object-oriented language, the language must support all three of these concepts.

Because encapsulation is the primary object-oriented mandate, you must make all attributes private. Making the attributes public is not an option.

Thus, it appears that the use of inheritance may be severely limited. If a subclass does not have access to the attributes of its parent, this situation presents a sticky design problem. To allow subclasses to access the attributes of the parent, language designers have included the access modifier protected. There are actually two types of protected access.

Rule of thumb is **AVOID CONCRETE INHERITANCE!!!** Only use it when it has been imposed on your and you cannot change the class you are reusing.

What is the Inheritance alternative? **Favour Composition over inheritance.**

3.2.3 Polymorphism

Polymorphism takes any shape.

Polymorphism is the ability of an object to take on many forms. The most common use of polymorphism in OOP occurs when a parent class reference is used to refer to a child class object. Any java object that can pass more than one **IS-A** test is considered to be polymorphic. In Java, all java objects are polymorphic since any object will pass the IS-A test for their own type and for the class Object. It is important to know that the only possible way to access an object is through a reference variable.

A reference variable can be of only one type. Once declared the type of a reference variable cannot be changed. The reference variable can be reassigned to other objects provided that it is not declared final.

The type of the reference variable would determine the methods that it can invoke on the object. A reference variable can refer to any object of its declared type or any subtype of its declared type. A reference variable can be declared as a class or interface type.

Let us look at an example.

Now the Cow class is considered to be polymorphic since this has multiple inheritance. Following are true for the above example:

- A Cow IS-A aAnimal
- A Cow IS-A Vegetarian
- A Cow IS-A Cow
- A Cow IS-A Object

When we apply the reference variable facts to a Deer object reference, the following declarations are legal:

```
public interface Vegetarian{}
public class Animal{}
public class Cow extends Animal implements Vegetarian{}
```

Cow c = new Cow(); Animal a = c; Vegetarian v = c; Object o = c;

```
Cow c = new Cow();
Animal a = c;
Vegetarian v = c;
Object o = c;
```

All the reference variables **c,a,v,o** refer to the same Cow object in the heap.

3.3 Software Design Principals

Great software is built in an agile way. Agility is about building software in tiny increments, it is important to take the time to ensure that the software has a good structure that is flexible, maintainable, and reusable.

In an agile team, the big picture evolves along with the software. With each iteration, the team improves the design of the system so that it is as good as it can be for the system as it is now.

The team does not spend very much time looking ahead to future requirements and needs. Nor does it try to build in today the infrastructure to support the features that may be needed tomorrow.

Rather, the team focuses on the current structure of the system, making it as good as it can be. This is a way to incrementally evolve the most appropriate architecture and design for the system. It is also a way to keep that design and architecture appropriate as the system grows and evolves over time.

Agile development makes the process of design and architecture continuous. How do we know how whether the design of a software system is good?

Below are the tips to help us detect bad design

The symptoms are:

1. **Rigidity:** The design is difficult to change.Rigidity is the tendency for software to be difficult to change. A design is rigid if a single change causes a cascade of subsequent changes in dependent modules.
2. **Fragility** is the tendency of a program to break in many places when a single change is made.
3. **Immobility:** A design is immobile when it contains parts that could be useful in other systems, but the effort and risk involved with separating those parts from the original system are too great.
4. **Viscosity.** It is difficult to do the right thing. Viscosity comes in two forms: viscosity of the software and viscosity of the environment. Goal is to design our software such that the changes that preserve the design are easy to make. Viscosity of environment comes about when the development environment is slow and inefficient In both cases, a viscous project is one in which the design of the software is difficult to preserve. We want to create systems and project environments that make it easy to preserve and improve the design.
5. **Needless complexity.** Overdesign. A design smells of needless complexity when it contains elements that aren't currently useful. This frequently happens when developers anticipate changes to the requirements and put facilities in the software to deal with those potential changes.
6. **Needless repetition.** Mouse abuse.Cut and paste may be useful text-editing operations, but they can be disastrous code-editing operations. Bad code can easily propagate to all modules and make it hard to change.
7. **Opacity.** Disorganized expression. Opacity is the tendency of a module to be difficult to understand. Code can be written in a clear and expressive manner, or it can be written in an opaque and convoluted manner. Code that evolves over time tends to become more and more opaque with age. A constant effort to keep the code clear and expressive is required in order to keep opacity to a minimum.

3.3.1 The Single-Responsibility Principle (SRP)

The Single Responsibility Principle (SRP) states that a class or module should have one, and only one, reason to change.

In the context of the SRP, we define a responsibility to be a reason for change. If you can think of more than one motive for changing a class, that class has more than one responsibility

If a class assumes more than one responsibility, that class will have more than one reason to change. Why SRP matters

1. We want it to be easy to reuse code
2. Big classes are more difficult to change
3. Big classes are harder to read
4. Dont code for situations that you wont ever need
5. Dont create unneeded complexity
6. However, **more class files != more complicated** Offshoot of SRP - Small Methods
7. A method should have one purpose (reason to change)
8. Easier to read and write, which means you are less likely to write bugs
9. Write out the steps of a method using plain English method names

3.3.2 The Open/Closed Principle (OCP)

Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

When a single change to a program results in a cascade of changes to dependent modules, the design smells of rigidity. OCP advises us to refactor the system so that further changes of that kind will not cause more modifications.

If OCP is applied well, further changes of that kind are achieved by adding new code, not by changing old code that already works.

Anytime you change code, you have the potential to break it

Modules that conform to OCP have two primary attributes.

1. They are open for extension. This means that the behavior of the module can be extended. As the requirements of the application change, we can extend the module with new behaviors that satisfy those changes. We are able to change what the module does.
2. They are closed for modification. Extending the behavior of a module does not result in changes to the source, or binary, code of the module.

The normal way to extend the behavior of a module is to make changes to the source code of that module. Modules behaviours can be changed without modifying the code by means of abstractions. In object-oriented programming language (OOPL), it is possible to create abstractions that are fixed and yet represent an unbounded group of possible behaviors.

The abstractions are abstract base classes, and the unbounded group of possible behaviors are represented by all the possible derivative classes. It is possible for a module to manipulate an abstraction.

Such a module can be closed for modification, since it depends on an abstraction that is fixed. Yet the behavior of that module can be extended by creating new derivatives of the abstraction It is possible for a module to manipulate an abstraction. Such a module can be closed for modification, since it depends on an abstraction that is fixed. Yet the behavior of that module can be extended by creating new derivatives of the abstraction.

Two Design Patterns used to enforce OCP are Template Method and Strategy . These two patterns are the most common ways of satisfying OCP. They represent a clear separation of generic functionality from the detailed implementation of that functionality. Will cover more of these in the next chapter on Design Patterns and Refactoring.

The Open/Closed Principle is at the heart of object-oriented design. Conformance to this principle is what yields the greatest benefits claimed for object-oriented technology: flexibility, reusability, and maintainability.

3.3.3 The Liskov Substitution Principle (LSP)

Subclasses should be substitutable for their base classes. The Super class and the sub class should be substitutable and make sense when used.

We mentioned that OCP is the most important of the class category principles. We can think of the Liskov Substitution Principle (LSP) as an extension to OCP. In order to take advantage of LSP, we must adhere to OCP because violations of LSP also are violations of OCP, but not vice versa.

In its simplest form, LSP is difficult to differentiate from OCP, but a subtle difference does exist. OCP is centered around abstract coupling. LSP, while also heavily dependent on abstract coupling, is in addition heavily dependent on preconditions and postconditions, which is LSPs relation to Design by Contract, where the concept of preconditions and postconditions was formalized.

A precondition is a contract that must be satisfied before a method can be invoked. A postcondition, on the other hand, must be true upon method completion. If the precondition is not met, the method shouldnt be invoked, and if the postcondition is not met, the method shouldnt return.

The Liskovs Substitution Principle provides a guideline to sub-typing any existing type. Stated formally it reads:

If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behaviour of P is unchanged when o1 is substituted for o2 then S is a subtype of T.

In a simpler term, if a program module is using the reference of a Base class, then it should be able to replace the Base class with a Derived class without affecting the functioning of the program module.

3.3.4 The Dependency-Inversion Principle (DIP)

Depend upon abstractions. Do not depend upon concretions.

The Dependency Inversion Principle (DIP) formalizes the concept of abstract coupling and clearly states that we should couple at the abstract level, not at the concrete level.

Two classes are tightly coupled if they are linked together and are dependent on each other Tightly coupled classes can not work independent of each other

It Makes changing one class difficult because it could launch a wave of changes through tightly coupled classes

In our own designs, attempting to couple at the abstract level can seem like overkill at times. Pragmatically, we should apply this principle in any situation where were unsure whether the implementation of a class may change in the future.

But in reality, we encounter situations during development where we know exactly what needs to be done. Requirements state this very clearly, and the probability of change or extension is quite low. In these situations, adherence to DIP may be more work than the benefit realized.

At this point, there exists a striking similarity between DIP and OCP. In fact, these two principles are closely related. Fundamentally, DIP tells us how we can adhere to OCP.

Or, stated differently, if OCP is the desired end, DIP is the means through which we achieve that end. While this statement may seem obvious, we commonly violate DIP in a certain situation and dont even realize it.

Abstract coupling is the notion that a class is not coupled to another concrete class or class that can be instantiated. Instead, the class is coupled to other base, or abstract, classes. This abstract class can be either a class with the abstract modifier or a Java interface data type.

Regardless, this concept actually is the means through which LSP achieves its flexibility, the mechanism required for DIP, and the heart of OCP. The OCP is the guiding principle for good Object- Oriented design.

The design typically has two aspects with it. One, you design an application module to make it work and second, you need to take care whether your design, and thereby your application, module is reusable, flexible and robust. The OCP tells you that the software module should be open for extension but closed for modification.

As you might have already started thinking, this is a very high level statement and the real problem is how to achieve this. Well, it comes through practice, experience and constant inspection of any piece of design, understanding that how it performs and works tackles with the expanding re- quirements of the application. But even though you are a new designer as student, you can follow certain principles to make sure that your design is a good one.

The Dependency Inversion Principle helps you make your design OCP compliant. Formally stated, the principle makes two points: High level modules should not depend upon low level modules. Both should depend upon abstractions Abstractions should not depend upon details. Details should depend upon abstractions.

3.3.5 The Interface Segregation Principle (ISP)

Many specific interfaces are better than a single, general interface.

This principle deals with the disadvantages of ”fat” interfaces. Classes whose interfaces are not cohesive have ”fat” interfaces. In other words, the interfaces of the class can be broken up into groups of methods. Each group serves a different set of clients. Thus, some clients use one group of methods, and other clients use the other groups.

Any interface we define should be highly cohesive. In Java, we know that an interface is a reference data type that can have method declarations, but no implementation. In essence, an interface is an abstract class with all abstract methods.

As we define our interfaces, it becomes important that we clearly understand the role the interface plays within the context of our application. In fact, interfaces provide flexibility: They allow objects to assume the data type of the interface.

Consequently, an interface is simply a role that an object plays at some point throughout its lifetime. It follows, rather logically, that when defining the operation on an interface, we should do so in a manner that doesnt accommodate multiple roles.

Therefore, an interface should be responsible for allowing an object to assume a SINGLE ROLE, assuming the class of which that object is an instance implements that interface. 3.3.6. Composite Reuse Principle (CRP)

Favor polymorphic composition of objects over inheritance.

The Composite Reuse Principle (CRP) prevents us from making one of the most catastrophic mistakes that contribute to the demise of an object-oriented system: using inheritance as the primary reuse mechanism.

Inheritance can be thought of as a generalization over a specialization relationship That is, a class higher in the inheritance hierarchy is a more general version of those inherited from it. In other words, any ancestor class is a partial descriptor that should define some default characteristics that are applicable to any class inherited from it.

Any time we have to override default behavior defined in an ancestor class, we are saying that the ancestor class is not a more general version of all of its descendants but actually contains descriptor characteristics that make it too specialized to serve as the ancestor of the class in question.

Therefore, if we choose to define default behavior on an ancestor, it should be general enough to apply to all of its descendents. In practice, its not uncommon to define a default behavior in an ancestor class. However, we should still accommodate CRP in our relationships.

3.3.6 Principle of Least Knowledge (PLK)

For an operation O on a class C, only operations on the following objects should be called:itself, its parameters, objects it creates, or its contained instance objects.

The Principle of Least Knowledge (PLK) is also known as the Law of Demeter. The basic idea is to avoid calling any methods on an object where the reference to that object is obtained by calling a method on another object.

Instead, this principle recommends we call methods on the containing object, not to obtain a reference to some other object, but instead to allow the containing object to forward the request to the object we would have formerly obtained a reference to.

The primary benefit is that the calling method doesnt need to understand the structural makeup of the object its invoking methods upon. The obvious disadvantage associated with PLK is that we must create many methods that only forward method calls to the containing classes internal components. This can contribute to a large and cumbersome public interface.

An alternative to PLK, or a variation on its implementation, is to obtain a reference to an object via a method call, with the restriction that any time this is done, the type of the reference obtained is always an interface data type.

This is more flexible because we arent binding ourselves directly to the concrete implementation of a complex object, but instead are dependent only on the abstractions of which the complex object is composed.

This is how many classes in Java typically resolve this situation.

3.4 Software Packing Principles

In this section, we explore the principles of design that help us split a large software system into packages.

As software applications grow in size and complexity, they require some kind of highlevel organization. Classes are convenient unit for organizing small applications but are too finely grained to be used as the sole organizational unit for large applications. Something ”larger” than a class is needed to help organize large applications. That something is called a package, or a component.

This section outlines six principles for managing the contents and relationships between components. The first three, principles of package cohesion, help us allocate classes to packages. The last three principles govern package coupling and help us determine how packages should be interrelated. The last two principles also describe a set of dependency management metrics that allow developers to measure and characterize the dependency structure of their designs.

3.4.1 Principles of Component Cohesion: Granularity

The principles of component cohesion help developers decide how to partition classes into components. These principles depend on the fact that at least some of the classes and their interrelationships have been discovered. Thus, these principles take a bottom-up view of partitioning.

3.4.1.1 The Reuse/Release Equivalence Principle (REP)

The granule of reuse is the granule of release.

REP states that the granule of reuse, a component, can be no smaller than the granule of release. Anything that we reuse must also be released and tracked. It is not realistic for a developer to simply write a class and then claim that it is reusable.

Reusability comes only after a tracking system is in place and offers the guarantees of notification, safety, and support that the potential reusers will need.

Whenever a client class wishes to use the services of another class, we must reference the class offering the desired services. If the class offering the service is in the same package as the client, we can reference that class using the simple name. If, however, the service class is in a different package, then any references to that class must be done using the class fully qualified name, which includes the name of the package. Any Java class may reside in only a single package.

Therefore, if a client wishes to utilize the services of a class, not only must we reference the class, but we must also explicitly make reference to the containing package. Failure to do so results in compile- time errors. Therefore, to deploy any class, we must be sure the containing package is deployed. Because the package is deployed, we can utilize the services offered by any public class within the package.

While we may presently need the services of only a single class in the containing package, the services of all classes are available to us.

Consequently, our unit of release is our unit of reuse, resulting in the Release Reuse Equivalency Principle (REP). This leads us to the basis for this principle, and it should now be apparent that the packages into which classes are placed have a tremendous impact on reuse. Careful consideration must be given to the allocation of classes to packages.

3.4.1.2 The Common Reuse Principle (CReP)

Classes that arent reused together should not be grouped together.

If we need the services offered by a class, we must import the package containing the necessary classes.

As we stated previously in our discussion of REP (Release Reuse Equivalency Principle), when we import a package, we also may utilize the services offered by any public class within the package. In addition, changing the behavior of any class within the service package has the potential to break the client.

Even if the client doesnt directly reference the modified class in the service package, other classes in the service package being used by clients may reference the modified class. This creates indirect dependencies between the client and the modified class that can be the cause of mysterious behavior. We can state the following: If a class is dependent on another class in a different package, then it is dependent on all classes in that package, albeit indirectly. This principle has a negative connotation.

It doesnt hold true that classes that are reused together should reside together, depending on CCP. Even though classes may always be reused together, they may not always change together. In striving to adhere to CCP, separating a set of classes based on their likelihood to change together should be given careful consideration.

Of course, this impacts REP because now multiple packages must be deployed to use this functionality. Experience tells us that adhering to one of these principles may impact the ability to adhere to another. Whereas REP and Common Reuse Principle (CReP) emphasize reuse, CCP emphasizes maintenance.

3.4.1.3 The Common Closure Principle (CCP)

Classes that change together, belong together.

The basis for the Common Closure Principle (CCP) is rather simple. Adhering to fundamental programming best practices should take place throughout the entire system.

Functional cohesion emphasizes well-written methods that are more easily maintained. Class cohesion stresses the importance of creating classes that are functionally sound and dont cross responsibility boundaries.

And package cohesion focuses on the classes within each package, emphasizing the overall services offered by entire packages.

During development, when a change to one class may dictate changes to another class, its preferred that these two classes be placed in the same package.

Conceptually, CCP may be easy to understand; however, applying it can be difficult because the only way that we can group classes together in this manner is when we can predictably determine the changes that might occur and the effect that those changes might have on any dependent classes.

Predictions often are incorrect or arent ever realized.

Regardless, placement of classes into respective packages should be a conscious decision that is driven not only by the relationships between classes, but also by the cohesive nature of a set of classes working together.

3.4.2 Principles of Component Coupling: Stability

The next three principles deal with the relationships between components. Here again, we will run into the tension between developability and logical design. The forces that impinge on the architecture of a component structure are technical, political, and volatile.

3.4.2.1 The Acyclic Dependencies Principle (ADP)

The dependencies between packages must form no cycles.

Cycles among dependencies of the packages composing an application should almost always be avoided.

Packages should form a Directed Acyclic Graph (DAG). Acyclic Dependencies Principle (ADP) is important from a deployment perspective.

Along with packages being reusable and maintanable, the should also be deployable as well. Just as in the class design, the package design should have defined dependencies so that it is deployment- friendly.

3.4.2.2 The Stable-Dependencies Principle (SDP)

Depend in the direction of stability.

Stability implies that an item is fixed, permanent, and unvarying. Attempting to change an item that is stable is more difficult than inflicting change on an item in a less stable state. Aside from poorly written code, the degree of coupling to other packages has a dramatic impact on the ease of change.

Those packages with many incoming dependencies have many other components in our application dependent on them.

These more stable packages are difficult to change because of the far-reaching consequences the change may have throughout all other dependent packages. On the other hand, packages with few incoming dependencies are easier to change.

Those packages with few incoming dependencies most likely will have more outgoing dependencies. A package with no incoming or outgoing dependencies is useless and isnt part of an application because it has no relationships.

Therefore, packages with fewer incoming, and more outgoing dependencies, are less stable. Stability metrics Stability is calculated by counting the number of dependencies that enter and leave that component.

These counts allow us to calculate the positional stability of the component:

Ca (afferent couplings): The number of classes outside this component that depend on classes within this component

Ce (efferent couplings): The number of classes inside this component that depend on classes outside this component

Formula: I(instability) = Ce/(Ce +Ca)

This metric has the range [0,1]. I = 0 indicates a maximally stable component. I = 1 indicates a maximally unstable component.

The Ca and Ce metrics are calculated by counting the number of classes outside the component in question that have dependencies on the classes inside the component in question.

3.4.2.3 The Stable-Abstractions Principle (SAP)

Stable packages should be abstract packages.

One of the greatest benefits of object orientation is the ability to easily maintain our systems. The high degree of resiliency and maintainability is achieved through abstract coupling. By coupling concrete classes to abstract classes, we can extend these abstract classes and provide new system functions without having to modify existing system structure.

Consequently, the means through which we can depend in the direction of stability, and help ensure that these more depended-upon packages exhibit a higher degree of stability, is to place abstract classes, or interfaces, in the more stable packages.

We can state the following: More stable packages, containing a higher number of abstract classes, or interfaces, should be heavily depended upon. Less stable packages, containing a higher number of concrete classes, should not be heavily depended upon. Any packages containing all abstract classes with no incoming dependencies are utterly useless.

On the other hand, packages containing all concrete classes with many incoming dependencies are extremely difficult to maintain.

Measuring abstraction

The A metric is a measure of the abstractness of a component. Its value is simply the ratio of abstract classes in a component to the total number of classes in the component, where Nc is the number of classes in the component. Na is the number of abstract classes in the component. Remember, an abstract class is a class with at least one abstract method and cannot be instantiated:

3.5 Chapter Questions

1. Write small program using TDD method to demonstrate the three core principals of Object Oriented Programming
2. In question 1, you wrote code demonstrating inheritance. Modify your code to use an alternative solution to inheritance.
3. There are 7 core software principles and for each write code that violet the principle and then write code that obeys the principle
4. Write code that violets ADP and also write code that corrects the violation.

Chapter 4

Domain Driven Design

4.1 Chapter Objectives

1. Understand what Domain Driven Design is and when and why it is valuable to software intensive organizations.
2. Describe how the principle of " separation of concerns " has been applied to the main system design
3. Know the the basic principles and processes needed to develop the useful sort of models, tie them into implementation and business analysis, and place them within a viable, realistic strategy.
4. Context Mapping: A pragmatic approach to dealing with the diversity models and processes on real large projects with multi-team/multi-subsystem development.
5. Combining the Core Domain and Context Map to illuminate Strategic Design options for a project.

4.2 Introduction

Domain Driven Design (DDD) is an approach of how to model the core logic of an application. The term itself was coined by Eric Evans in his book "Domain Driven Design". The basic idea is that the design of your software should directly reflect the Domain and the Domain-Logic of the (business-) problem you want to solve with your application. That helps understanding the problem as well as the implementation and increases maintainability of the software.

The Domain Driven Design approach introduces common principles and patterns that should be used when modelling your Domain. There are the "building blocks" that should be used to build your domain model and principles that helps to have a nice "supple design" in your implementation.

Domain-driven design flows from the premise that the heart of software development is knowledge of the subject matter and finding useful ways of understanding that subject matter. The complexity that we should be tackling is the complexity of the domain itself

– not the technical architecture, not the user interface, not even specific features. This means designing everything around our understanding and conception of the most essential concepts of the business and justifying any other development by how it supports that core.

4.3 Basics

The Developer never knows enough about the problem. But the domain experts know their domain and the developer has to understand it. The domain experts know rules of their business process but often they are not aware using them because it is natural for them. Therefore "knowledge crunching" is required to identify a proper domain model (DM).

A common language between users (domain experts) and developers is required that helps to understand the domain and the problem.

The domain model offers a simplified, abstract view of the problem. It can have several illustrations: speech / UML / code. Defining the domain model is a cyclic process of refining the domain model. The DM grows and changes because knowledge grows during implementation and analysis!

Of course the domain model must be useable for implementation. In practice there are so many analytic Models which are not implemented ever. You have to find one which can be implemented in design!

4.4 Ubiquitous Language

In order to understand the problem area, ie the domain, of your application, a common language should be used to describe the problem. The goal is that this language (and the common vocabulary in it) can be understood by the developers and the domain experts (e.g. the client).

1. The DM is the backbone and it uses terms of the "ubiquitous language"
2. Therefore no translation between developers and domain experts - as well between developers code and models.
3. During the analysis the ubiquitous language should be used to describe and discuss problems and requirements. If there are new requirements it means new words enter the common language

Speak in ubiquitous language (like a foreign language)

1. Try to explain scenarios loud with the use of the model and the ubiquitous language
2. Try to explain scenarios more simple/better (find easier ways to say what you need to say). That helps refining the model.

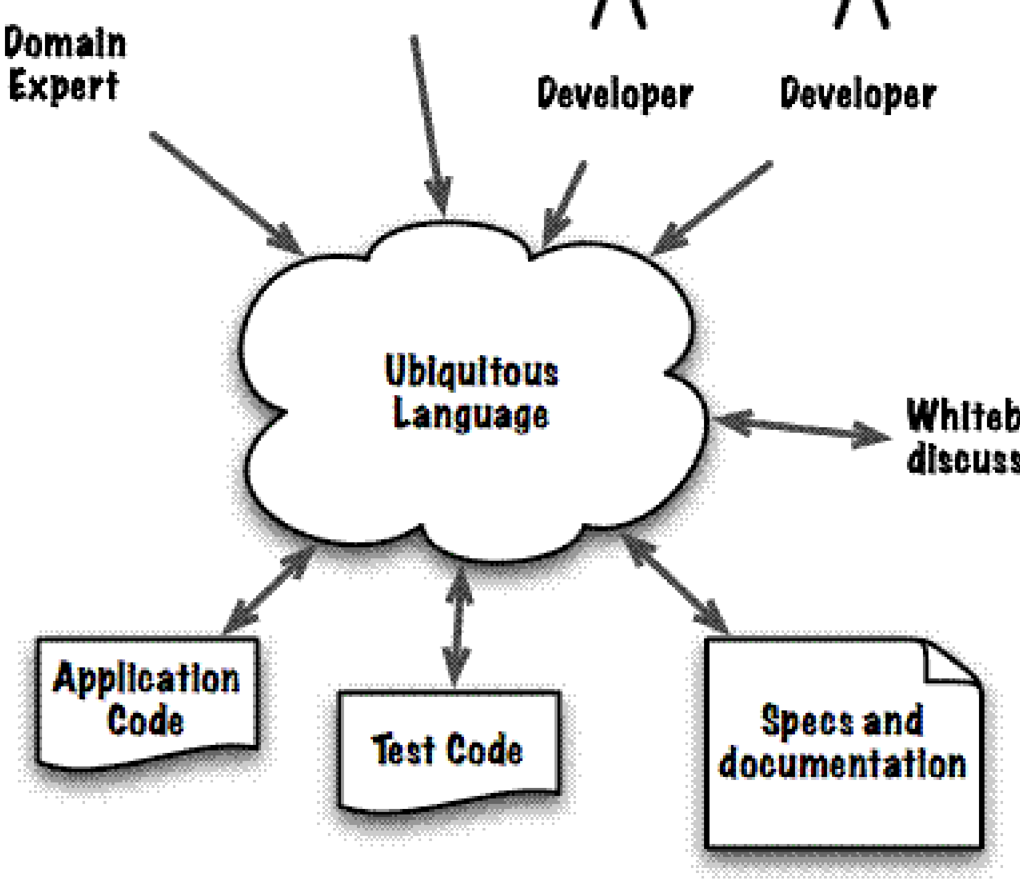


Figure 4.1: The Ubiquitous Language should be the only language used to express a model. Everybody in the team should be able to agree on every specific term without ambiguities and no translation should be needed

4.4.1 Context Mapping

In Domain Driven Design, a Context is defined as: "The setting in which a word or a statement appears that determines its meaning"

4.4.1.1 Example of Context Mapping

Let's start with an example where the ambiguity might happen at the terminology level. Some words have different meaning depending on the context they're used in.

Let's suppose we're working on a web-based Personal Finance Management Application (PFM) . We'll probably use this application to manage the banking accounts, stocks, and savings, to track the budget and expenses, and so on.

In our application, the domain term Account might refer to different concepts. Talking about banking, an account is some kind of a logical "container for money"; we'll then expect the corresponding class to have attributes such as balance, account number, and so on. But, in the context of web applications, the term account has a very different meaning, related to authentication and user credentials. The corresponding model, as shown in Figure 2.2 , will then be something completely different.

Please see the Cheat Sheet for DDD on WEBCT for this in detail.

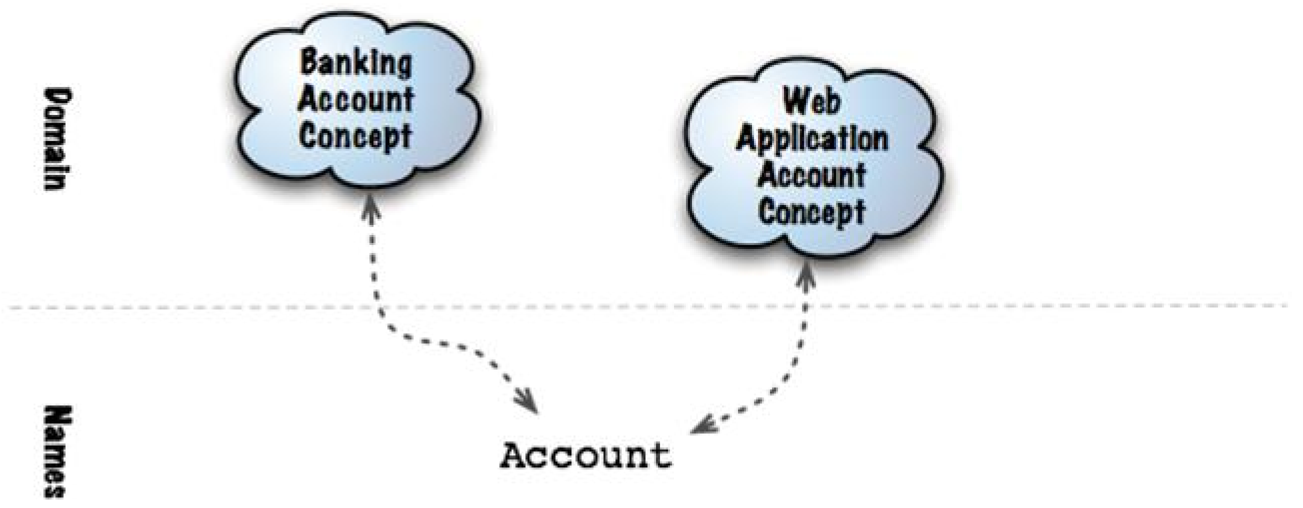


FIGURE 4.2: A somewhat trivial case of ambiguity: the term Account might mean something very different according to the context it's used in

4.5 UML and Domain Driven Design

Common is the use of class-diagrams to describe the domain model.

1. Use of UML class diagrams to sketch the main classes.
2. Do not go to draw every class you going to code. Keep overview of the main details!
3. Use structure (packages and subpackages). Use freetext to explain constrains and relations.
4. Simplify the class diagram when you use it to talk with domain experts (better a diagramm is useful than standard compliant)

Please note that a comprehensive UML of entire object model fails to communicate or explain. It tends to overwhelm reader with details. Therefore use simplified diagrams of conceptual important parts that are essential to understand.

4.6 Main Domain Model elements-the building blocks

The main elements of a domain model are Layered Architecture, Elements, Associations between elements, Aggregates, Factories, and Repositories. Common patterns like Repositories and Factories helps completing the model.

4.6.1 Layered Architecture

The basic principle for software that is build with domain driven design is to use a layered architecture. Where the heart of the software is the domain model. Basic principles of

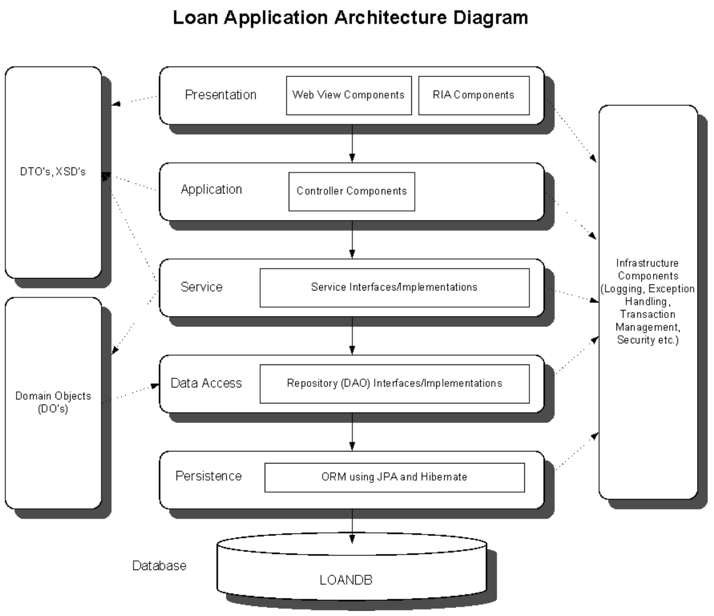

Loan Application Architecture Diagram
DTO's, XSD's
Domain Objects (DO's)
Presentation
Web View Components
RIA Components
Application
Controller Components
Service
Service Interfaces/Implementations
Data Access
Repository (DAO) Interfaces/Implementations
Persistence
ORM using JPA and Hibernate
Database
LOANDB
Infrastructure Components (Logging, Exception Handling, Transaction Management, Security etc.)

FIGURE 4.3: The Sample Layers in Domain Driven Design

layered architecture is that a layer never knows something about the layers above. That also mean the domain layer, as the heart of the application, is responsible for all the business logic but should never be "dirtied" by view requirements.

Modern applications are often implemented using a layered architecture where different layers support separation of concerns. A typical breakdown consists of the following layers

- User Interface
- Application
- Domain
- Infrastructure

The key is to isolate domain concepts from system concepts. To apply our best thinking, we need to be able to look at the elements of our model and see them as a system. We must not be forced to pick [domain concepts] out of a much larger mix of objects, like trying to identify constellations in the night sky

Dependencies between layers should exist in only one direction. As such, within a layer, an object can depend on other objects in its layer and objects in layers below it. If, and a big IF here, an object in a lower layer needs to communicate with an object in a layer above it needs to use indirect mechanisms, such as callbacks an upper object passes itself as a parameter to a lower object after implementing a predefined callback interface. The lower object uses this reference to communicate back up using something like the Observer pattern See in Chapter on Design Patterns

4.6.1.1 Domain Layer

Objects within the domain layer are elements of the model. They should be isolated from the UI, Application, and Infrastructure layers as much as possible. The domain objects, free of the responsibility of displaying themselves, storing themselves, managing application tasks, and so forth, can be focused on expressing the domain model. This allows a model to evolve and capture essential business knowledge and put it to work.

The domain layer is where all of the concepts, behaviors, and rules specified for the model are implemented; the other layers should be devoid of domain logic as much as possible. Rather than implementing a domain rule in the application layer, have it call the domain layer and respond appropriately e.g., a violation of a business rule might raise an exception in the domain layer, that is caught by the application layer, and displayed by the UI layer.

4.6.1.2 Entities

Some objects are not defined primarily by their attributes. They represent a thread of identity that runs through time and often across distinct representations. Consider the notion of customer in a typical business system. Customer may have a payment history. If its good, status will accrue; if its bad, the customers information may be transferred to a bill-collection agency.

The same customer may be in the contact management software used by your companys sales force. The customer may be squashed flat for storage in a database. If business stops, the customer may be placed in an archive. Each aspect of the customer may be implemented in multiple ways, using different representations and/or programming languages They all represent the SAME customer however, and some means must exist to match them even though their attributes may be different

An object defined primarily by its identity is called an Entity. They have life cycles that can radically change their form and content. Their identities must be defined so that they can be effectively tracked. This notion of identity is DIFFERENT from the identity mechanisms of programming languages; i.e., it is different from a == b and a.equals(b) that OO languages provide

For example, two deposits of the same amount made to the same bank account on the same day are NOT identical; they are two separate entity objects in the banking domain the objects representing the amounts ARE identical, however, and are most likely Value Objects (discussed next)

The key to modeling an entity object is to include only those attributes that are used to establish its identity or are commonly used to find or match it; include only those behaviors that support the task of maintaining its identity. All other behaviors and attributes should be placed in separate objects (some of which may also be Entities)

Each Entity must have a way of establishing its identity such that two instances of the same entity can be distinguished from one another, even if they both contain the same descriptive attributes (like our bank deposits from above) Identity is often operationally established by ensuring that a single attribute has a unique id or ensuring that some combination of attribute values always produce a unique key.

Often the means for establishing identity require a careful study of the domain; what is it that humans do to distinguish the real-world counterparts of the entity object?

Lastly, remember that an Object which is primary defined by its identity (not by attributes), eg "Person", "Car", "Costumer" or "BankTransaction", has the following properties:

1. Continuity through live cycle
2. Defined by its own not by attributes

Please, keep the class definition simple. Focus on live cycle. Be alert to requirements that require a matching through attributes (because of possible problems with the need for identity).Use of some Identifier (IDs) is often useful

4.6.1.3 Value Objects

Some objects have no conceptual identity; these objects describe some aspect of a thing. A person may be modeled as an Entity with an identity, but that persons NAME is a Value object Values are instantiated to represent elements of a design that we care about only for WHAT they are, not WHO they are Example values Colors, Dates, Numbers, Strings, etc. Values are immutable; once created their values do not change create values via factory methods; do not provide setter methods operations that manipulate values produce new values as a result Benefits: such objects can be easily shared

Once again, value objects describe the characteristic of a thing and identity is not required. we care what they are not who or which. For example "address", "color". Value objects have these properties:

1. Value objects are allowed to reference an entity
2. Value objects are often passed as parameter in messages between other objects
3. Value objects are immutable! Ideally the only have getter methods and their attributes are set during construction (constructor method).
4. Value objects are a whole. Make them a whole value- meaning a conceptual whole (e.g. address)
5. Value objects have no identity. Lack of identity gives freedom in design we dont need to care of identity: we can simply delete and create new ones if required.

A good example is "color" that can be implemented as value object. That means it is an object that represents a certain color. Since value objects are immutable it is not allowed to change the color object. So if you need to get a color that is lighter than another color: You throw the old color away and build a new color - that is possible because you don't need to take care about identity. (Typically implemented as method in a colorService or colorFactory class for example)

4.6.1.4 Associations between elements

For every traversable association in the model, there is a mechanism in the software with the same properties. For example, an association between a customer and a sales representative represents, on one hand, domain knowledge. On the other hand, it also represents a pointer between two objects, or the result of a database lookup, etc.

Associations can be implemented in many ways. A one-to-many association can be implemented as a collection class pointed to by an instance variable, it might be a getter method that queries a database. Associations in the real world have lots of many-to-many relationships, with many being bidirectional, really hard to implement!

There are three techniques for making associations manageable. Avoid many association and use only a minimum of relations because they decrease maintainability!

1. impose a traversal direction instead of a bidirectional (bidirectional means both objects can only exist together. Ask if this is really the case)
2. adding qualifier, effectively reduces multiplicity: If you have much associations and/or multiplicity for both ends of the association, it can be a sign that a class is missing in your model.
3. eliminate non-essential associations

4.6.1.5 Services

In some situations, the clearest and most pragmatic design includes operations that do not conceptually belong to a single object; Rather than force the issue, we can follow the natural contours of the problem space and include SERVICES explicitly in the model. Be weary of Slippery slope: if you give up too often on finding a home for an operation, you will end up with a procedural programming solution. On the other hand, if you force an operation into an object that doesnt fit that objects definition, you weaken that objects cohesion and make it more difficult to understand

A service is an operation offered as an interface that stands alone in the model; it is defined purely in terms of what it can do for a client. Services tend to be named for what they can do (verbs rather than nouns). A good service has three characteristics. The operation relates to a domain concept that is not a natural part of an entity or value object The interface is defined in terms of other elements of the domain model. The operation is stateless (does not maintain or update its own internal state in response to being invoked)

If a single process or transformation in domain is not a natural responsibility of an entity or value, then make it a standalone service with a nice interface.

A service decouple entities and values from client (client in this case= the object that want to use another object) and therefore, they define an easy interface to use in client objects.

A service should

1. be stateless
2. be defined in the common language
3. use entity/value objects as parameters

4.6.1.6 Modules

Modules are groupings of model elements; They provide two views on a model one view provides details within an individual module. The second view provides information about relationships between modules. We shoot for modules with high cohesion and low coupling. . High cohesion: elements within a module all support the same purpose. Low coupling: elements within a module primarily reference.themselves; references to objects outside the module are kept to a minimum

If your model tells a story a module is a chapter. Modules:

1. helps understanding of large systems
2. provide low coupling between modules
3. make independent development possible
4. if you group objects in a module you want others to think of this objects together. Group by meaning in the domain layer

4.6.2 Domain Object Life Cycle

Every object has a life cycle. It is created. It moves through various states. It is then deleted or archived. If the latter, it can eventually be restored and live again. For transient objects, this life cycle is simple to manage But for domain objects, this life cycle can be complicated. You need to keep track of state changes and each object may have complex relationships with other objects.

Two challenges occur in Model-Driven Design with respect to managing object life cycles. First, maintaining object integrity throughout the life cycle making sure constraints/rules/invariants are maintained. Secondly, preventing the model from getting swamped by the complexity of managing the life cycle.

We reduce complexity via the use of three patterns

Aggregates: which provide clear boundaries within the model and thereby reduce complexity Factories: used to encapsulate the complexities of creating and reconstituting complex objects (aggregates) Repositories: use to encapsulate the complexities of dealing with persistent complex objects (aggregates)

4.6.3 Aggregates

Model objects often participate in complex relationships. Managing the consistency of these relationships can be difficult. Compounding this problem is the fact that the real world often gives no hints as to the location of sharp boundaries in this web of concepts and relationships.

Lets looks at an example of Deleting a Person from a Database. Do you delete the Persons associated value objects? What if other Person objects need those value objects. If so, they may end up pointing to garbage. If not, you may litter the database with unreferenced value objects. Further compounding this problem is that these objects are often accessed by multiple users concurrently. We need to prevent simultaneous changes to interdependent objects

An Aggregate is a cluster of associated objects that we treat as a unit for the purpose of data changes. Each aggregate has a root and a boundary. The root is a single, specific entity object contained in the aggregate. The boundary defines what is IN the aggregate and what is OUT. Clients can only hold references to the root object of an aggregate.

Objects within the aggregate are allowed to hold references to one another. Objects within the aggregate have local identity but not global identity. Thus a Car object may have global identity but its tires do not. Aggregates can have invariants associated with them. These invariants are typically enforced/maintained by the root object. A delete operation on an aggregate must delete everything within the boundary at once

In summary, it is difficult to guarantee the consistency of changes in a model with many associations. Aggregates helps to limit dependencies. An aggregate is a group of objects that belong together (a group of individual objects that represents a unit). Each aggregate has a aggregate root. The client-objects only ”talk” to the aggregate root.

1. Invariants and rules have to be maintained. The aggregate root normaly takes care of invariants.
2. cluster entities/values into aggregates and define boundaries around each.
3. One entity is the aggregate root and controls all changes and access to the objects inside
4. Delete have to delete complete boundary

4.6.4 Factories

Factories are key elements in the domain layer that manage the creation of complex domain objects. Car engines are hard to build. Humans and robots are used to accomplish the task. Once built, the car engine can focus on what it does best. It doesnt need to know how to build itself. Furthermore, you dont need the humans/robots that created it, in order to use it.

The same is true of complex domain objects. We can create them with factories and then use them for their purpose without need for the factory. This approach can keep complex object construction and rule invariant code out of the domain objects themselves.

While they are considered part of the domain layer, Factories typically do NOT belong to the model.The factory defines a method with all the parameters needed to create a particular class of domain object. A client invokes the method providing the required parameters. The factory creates the new object and makes sure that all class invariants are valid.

The factory returns the newly created object to the client. Factories are thus ideal for creating Entities and Aggregates They are less necessary for Value Objects

Three Factory-related methods appear in the Design Patterns book Abstract Factory, Factory Method (see Chapter on Design), Builder.

Two basic requirements on Factories are that, first, each creation method is atomic (cannot be interrupted in the presence of multiple threads) and enforces all invariants of the created object or Aggregate. If something goes wrong during creation, the method should throw an exception than allow an object to be created in an inconsistent state. Placing invariant logic in a factory can often save a lot of space in the domain class itself; because typically a domain objects methods will not allow the object to be transformed into an illegal state after its created.

Secondly Factory methods should return abstract types, not concrete classes. Thus a Factory for a linked list in Java would return the type List and not the type LinkedList or ArrayList

Factories can be used to reconstitute archived objects. Reconstituting a persistent object can be a complex process There are two differences in this situation however. Entity objects are not given new identities; rather the stored information is used to reconstitute their previous identities. Rule violations will be handled differently. when first creating an object, a violation can safely cause an exception but if information from an archive causes a violation, it means either there is a bug in our domain class, allowing an object that was valid after creation to enter an invalid state or, there is a bug in our persistence code, that takes a valid object and stores it incorrectly OR, the persistent information was modified outside of our application regardless, rather than throwing an exception, we must attempt repair

Remember that a car does not build itself. A car is a useful and powerful object because of its associations and behaviour, but to overload it with logic for creating it is unlikely! A factory hides logic for building objects. This is especially relevant for aggregates! A factory can be implemented as a separate object or also as a embedded factory method.

Factory Method: Instead of a separate factory object, a factory method can be located in existing elements: for example in an aggregate root or in a host class where it is natural (e.g. createOrder method in a BookingClass)-

Factory functionality should only be located in the constructor if creation is simple and if there are good reasons. A constructor should always be atomic (never call other constructors).

A good factory:

1. is atomic (you need to pass all parameters that are need to build a valid "thing")
2. are predictable

3. not allowed to give wrong results (instead throw exception)
4. the factory will be coupled to its arguments, therefore carefully choose the parameters:
    - there is the danger of too much dependencies - better use parameters that are allready in the conceptual group or in dependencies already
    - use abstract types as parameter
5. the factory is responsible for invariants (maybe delegate it)
6. there are also factories for reconstitution

4.6.4.1 Entity factory

Factories for entities takes just essential attributes that are required to make a valid entity or aggregate. Details can be added later if they are not required by an invariant. The factory knows from where to get the identifier for an entity.

4.6.4.2 Value factory

Since value objects are immutable a factory or factory method for a value object takes the full description as parameters.

Get references of entities and objects: To do anything with an object you need to have a reference to it:

So how to get it? There are several options:

1. build a new object - e.g. with the help of a factory
2. reconstitution of an object (relevant when the id is known - done by the factory)
3. traversal between objects, means to ask other objects for objects. For example that have to be used for inner aggregates of course.

But there are some entities that need to be accessible through search based on attributes. One way to deal with this is the ”query and build” technique but this tends on a too technical focus and views objects as datacontainer (that is not optimal for DDD). That is where repositories enter the scene.

4.6.5 Repositories

To do anything with an object, you must have a reference to it. How do you get that reference? You could create the object or you could traverse an association from an object you already have or you could execute a query to find that object in a database based on its attributes. Most designs will use a combination of search and traversal to keep model-driven designs manageable. You need to be careful with search however, because it becomes easy to think of objects as just containers for the information stored in the database; your design can start to lose its OO feel. In particular, you can decide not to create aggregates and entities, and just use queries to grab the objects you need directly

To limit the scope of the object access problem, we can not worry about transient objects objects used only in the method that created them not provide query access to objects that can more easily be found via traversal thus, we dont need a search function for, e.g., a persons address. Instead we will traverse an association of the Person class to get that not provide query access to objects that are internal to an aggregate; you need to go through the aggregates root

Instead, we will use a Repository that provides search capabilities based on object attributes to find, typically, the roots of aggregates that are not convenient to reach by traversal.

A Repository represents a collection of objects of a certain type. Clients can add and remove objects from this set; the repository will take care of adding/removing the corresponding object to/from a particular persistence mechanism. Clients provide attributes to a repositorys search methods to gain access to particular objects in the domain. The repository takes care of creating the query needed to retrieve such objects from the persistence mechanism.

Benefits:

1. Repositories provide a simple model for accessing persistent objects
2. Repositories decouple applications from persistence mechanisms
3. Repositories can be defined abstractly and then be implemented in multiple ways (such as in-memory collections, XML, RDMS, etc.)

Query Types:

1. Repositories can support hard-coded queries and specification-based queries

Implementation Concerns

1. Client code ignores a repositorys implementation, developers do not.
2. Developers need to understand how queries bring
3. objects into memory and how that memory is reclaimed. Keep factories and repositories distinct
4. Have repositories use factories to reconstitute objects
5. With new objects, have clients add the object to a repository;
6. do not have a factory create and then add an object directly

In summary, for each object where you need global access create a repository object that can provide the illusion of an in memory collection of all objects of that type. Setup access through a well known global interface.

A Repository typically has methods for add, remove and find (=select based on some criteria). Advantages:

1. Decouple client from technical storage
2. Performance tuning possible
3. Communicate design decisions related to data access
4. keep model focus
5. dummy implementation for unit testing is possible.
6. Repositories can hide the OR mapping.

If the repository is responsible for retrieving a certain entity that is build by a factory, the repository normally calls the "reconstitution" method of the factory.

4.7 Making Supple Design

This section reviews a few of the techniques that are recomended for use in order to make his design supple. Supple is an adjective that means any of the folowing: Readily bent; pliant. Moving and bending with agility; limber.Yielding or changing readily; compliant or adaptable.

The ultimate purpose of software is to serve users. But first, that same software has to serve developers. This is especially true in a process that emphasizes refactoring. As a program evolves, developers will rearrange and rewrite every part. They will integrate the domain objects into the application and with new domain objects. Even years later, maintenance programmers will be changing and extending the code.

When software with complex behavior lacks a good design, it becomes hard to refactor or combine elements. Duplication starts to appear as soon as a developer isn't confident of predicting the full implications of a computation. Duplication is forced when design elements are monolithic, so that the parts cannot be recombined. Classes and methods can be broken down for better reuse, but it gets hard to keep track of what all the little parts do. When software doesn't have a clean design, developers dread even looking at the existing mess, much less making a change that could aggravate the tangle or break something through an unforeseen dependency. In any but the smallest systems, this fragility places a ceiling on the richness of behavior it is feasible to build. It stops refactoring and iterative refinement.

The following techniques make design supple

1. Intention-Revealing Interfaces
2. Side-Effect Free Functions

3. Assertions
4. Standalone Classes
5. Closure of Operations
6. Conceptual Contours

4.7.1 Intention-Revealing Interfaces

The interface of a class should reveal how that class is to be used. If developers dont understand the interface (and have access to source), they will look at the implementation to understand the class. At that point, the value of encapsulation is lost

So, name classes and methods to describe their effect and purpose, without reference to their implementation. These names should be drawn from the Ubiquitous Language. Write a test case before the methods are implemented to force your thinking into how the code is going to be used by clients

4.7.2 Side-Effect Free Functions

Operations (methods) can be broadly divided into two categories: commands and queries.

Commands are operations that affect the state of the system. Side effects are changes to the state of the system that are not obvious from the name of the operation. They can occur when a command calls other commands which call other commands etc. The developer invoked one command, but ends up changing multiple aspects of the system

Queries are read-only operations that obtain information from the system but do not change its state. Operations that return results without producing side effects are called functions. A function can call other functions without worrying about the depth of nesting; this makes it easier to test than operations with side effects.

To increase your use of side-effect-free functions, you can separate all query operations from all command operations. commands should not return domain information and be kept simple. Queries and calculations should not modify system state. Use Value Objects when possible to avoid having to modify domain objects. Operations on value objects typically create new value objects.

4.7.3 Assertions

After you have performed work on creating as many side-effect free functions and value objects as possible, you are still going to have command operations on Entity objects. To help developers understand the effects of these commands, use intention-revealing interfaces AND assertions.

Assertions typically state three things

1. the pre-conditions that must be true before an operation

2. the post-conditions that will be true after the operation
3. invariants that must always be true of a particular object (these are typically not associated with any particular operation)

More on Assertions in section on Testing.

4.7.4 Standalone Classes

Interdependencies make models and designs hard to understand. They also make them hard to test. We should do as much as possible to minimize dependencies in our models. Modules and Aggregates are two techniques already discussed for doing this. They dont eliminate dependencies but tend to reduce and/or limit them in some way.

Another technique is to identify opportunities for creating standalone classes for domain concepts. Such classes do not make use of any other domain concept.

4.7.5 Closure of Operations

If we take two real numbers and multiply them together, we get another real number. [The real numbers are all the rational numbers and all the irrational numbers.] Because this is always true, we say that the real numbers are "closed under the operation of multiplication": there is no way to escape the set. When you combine any two elements of the set, the result is also included in the set.

"CLOSURE OF OPERATIONS." The name comes from that most refined of conceptual systems, mathematics. $1 + 1 = 2$. The addition operation is closed under the set of real numbers. Mathematicians are fanatical about not introducing extraneous concepts, and the property of closure provides them a way of defining an operation without involving any other concepts. We are so accustomed to the refinement of mathematics that it can be hard to grasp how powerful its little tricks are.

Therefore: Where it fits, define an operation whose return type is the same as the type of its argument(s). If the implementer has state that is used in the computation, then the implementer is effectively an argument of the operation, so the argument(s) and return value should be of the same type as the implementer. Such an operation is closed under the set of instances of that type. A closed operation provides a high-level interface without introducing any dependency on other concepts.

Many Value Objects work in this way

java.lang.BigDecimal has examples of others; public BigDecimal add(BigDecimal value) public BigDecimal multiply(BigDecimal value)

Even partial closure is good. This refers to the situation of an operation that almost meets the definition, such as public BigDecimal divide(BigDecimal value, int roundingMode) or where the arguments are the same as the host class but the return type is different

4.7.6 Conceptual Contours

Sometimes people chop functionality fine to allow flexible combination. Sometimes they lump it large to encapsulate complexity. Sometimes they seek a consistent granularity, making all classes and operations to a similar scale. These are oversimplifications that don't work well as general rules. But they are motivated by a basic set of problems.

When elements of a model or design are embedded in a monolithic construct, their functionality gets duplicated. The external interface doesn't say everything a client might care about. Their meaning is hard to understand, because different concepts are mixed together.

On the other hand, breaking down classes and methods can pointlessly complicate the client, forcing client objects to understand how tiny pieces fit together. Worse, a concept can be lost completely. Half of a uranium atom is not uranium. And of course, it isn't just grain size that counts, but just where the grain runs.

Cookbook rules don't work. But there is a logical consistency deep in most domains, or else they would not be viable in their own sphere. This is not to say that domains are perfectly consistent, and certainly the ways people talk about them are not consistent. But there is rhyme and reason somewhere, or else modeling would be pointless. Because of this underlying consistency, when we find a model that resonates with some part of the domain, it is more likely to be consistent with other parts that we discover later. Sometimes the new discovery isn't easy for the model to adapt to, in which case we refactor to deeper insight, and hope to conform to the next discovery. This is one reason why repeated refactoring eventually leads to suppleness.

Find the conceptually meaningful unit of functionality, and the resulting design will be both flexible and understandable. For example, if an "addition" of two objects has a coherent meaning in the domain, then implement methods at that level. Don't break the add() into two steps. Don't proceed to the next step within the same operation. On a slightly larger scale, each object should be a single complete concept.

Therefore: Decompose design elements (operations, interfaces, classes, and AGGREGATES) into cohesive units, taking into consideration your intuition of the important divisions in the domain. Observe the axes of change and stability through successive refactorings and look for the underlying CONCEPTUAL CONTOURS that explain these shearing patterns. Align the model with the consistent aspects of the domain that make it a viable area of knowledge in the first place. The goal is a simple set of interfaces that combine logically to make sensible statements in the UBIQUITOUS LANGUAGE, and without the distraction and maintenance burden of irrelevant options.

4.8 Chapter Exercise and Assignment

Find the domain or business whose operations you really understand well. Draw a UML Domain Model for the business. Your UML Model should have at least 10 additional classes and correct relationships. Show at least five attributes of each class, but do not show the methods. It is assumed that your domain will deal with people and have a user management system. Below is the Model you can start with. The five classes have been done for you.

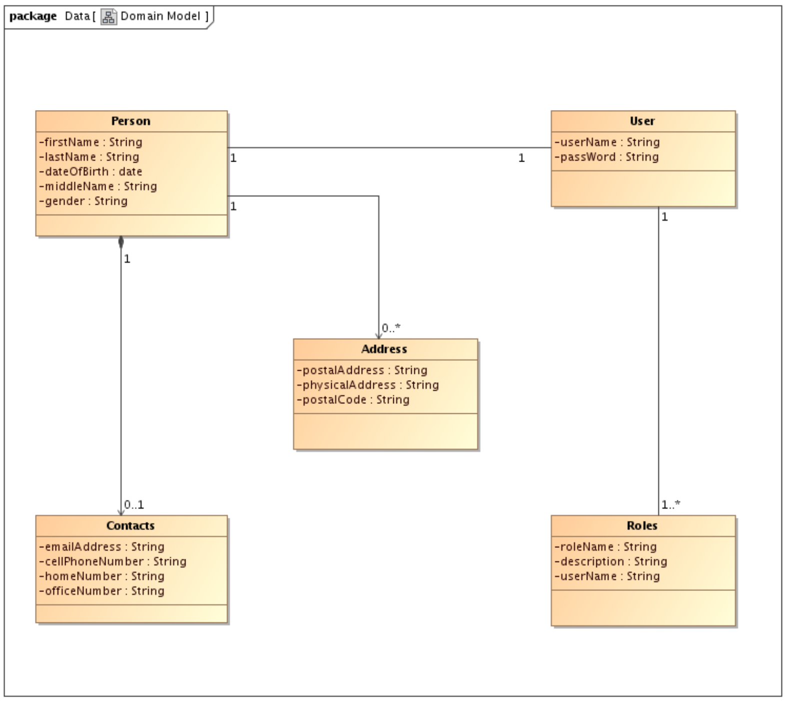


FIGURE 4.4: Domain Model for you to start with

Please take care that you pick the right model from start. This is the model you are going to work with for the rest of the semester. You will not be allowed to change your model. So pick a domain you are passionate about and understand very well.

Chapter 5

Design Patterns and Software Refactoring

5.1 Chapter Objectives

1. Be able to know what design patterns are.
2. From a given list, select the most appropriate pattern for a given scenario and demonstrate its applicability
3. From a list, select the most appropriate pattern for a given scenario. Patterns are limited to those documented in the book - Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software and are named using the names given in that book.
4. Understand the idea of refactoring and re-factor software code to Design Patterns

5.2 What are Patterns

Christopher Alexander talking about buildings and towns *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice "* Alexander, et al., A Pattern Language. Oxford University Press, 1977

Patterns can have different levels of abstraction. In Design Patterns (the book), Patterns are not classes, Patterns are not frameworks. Instead, Patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context

So, patterns are formalized solutions to design problems. They describe techniques for maximizing flexibility, extensibility, abstraction, etc. These solutions can typically be translated to code in a straightforward manner.

Its important note the fact that Patterns makes our life easy by giving guidelines to apply in a given context and for a specific problem.

5.3 Elements of Patterns

Pattern Name

This is more than just a handle for referring to the pattern. Each name adds to a designers vocabulary. It enables the discussion of design at a higher abstraction.

The Problem

This gives a detailed description of the problem addressed by the pattern. It describes when to apply a pattern, often with a list of preconditions.

The Solution This describes the elements that make up the design, their relationships,responsibilities, and collaborations. It does not describe a concrete solution. Instead a template to be applied in many situations

The consequences

This describes the results and tradeoffs of applying the pattern. It is critical for evaluating design alternatives Typically include impact on flexibility, extensibility, or portability, Space and Time tradeoffs, Language and Implementation issues

5.4 Design Patterns Templates

1. Pattern Name and Classification
   - Creational
   - Structural
   - Behavioural
2. Intent Also Known As Motivation and Applicability
3. Structure and Participants
4. Collaborations
5. Consequences
6. Implementation
7. Sample Code
8. Known Uses
9. Related Patterns

5.5 Documenting Design Patterns Selected Examples

The selected patterns are, Singleton and Factory Method

5.5.1 Singleton

Intent

Ensure a class has only one instance, and provide a global point of access to it

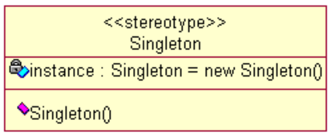
Motivation

Some classes represent objects where multiple instances do not make sense or can lead to a security risk (e.g. Java security managers)

Applicability

Use the Singleton pattern when there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point, when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code

Structure



Participants

Just the Singleton class

Collaborations

Clients access a Singleton instance solely through Singletons Instance operation

Consequences

Controlled access to sole instance. Reduced name space (versus global variables) Permits a variable number of instances (if desired)

Implementation

```java
import java.util.Date;

public class Singleton {
    private static Singleton theOnlyOne;
    private Date d = new Date();

    private Singleton() {
    }

    public synchronized static Singleton instance() {
        if (theOnlyOne == null) {
            theOnlyOne = new Singleton();
        }
    return theOnlyOne;
    }

    public Date getDate() {
    return d;
    }
}
```

Listing 5.1: Singleton Implementation

Usage

```
public class UseSingleton {
    public static void main(String[] args) {
        Singleton a = Singleton.instance();
        Singleton b = Singleton.instance();
        System.out.println("" + a.getDate());
        System.out.println("" + b.getDate());
        System.out.println("" + a);
        System.out.println("" + b);
    }
}
```

Listing 5.2: Singleton Usage

Output:

```
Sun Apr 07 13:03:34 MDT 2002
Sun Apr 07 13:03:34 MDT 2002
Singleton@136646
Singleton@136646
```

5.5.2 Factory Method

Intent

Define an interface for creating an object, but let subclasses decide which class to instantiate. Also Known As Virtual Constructor

Motivation

Frameworks define abstract classes, but any particular domain needs to use specific subclasses; how can the framework create these subclasses?

Applicability

Use the Factory Method pattern when a class cant anticipate the class of objects it must create. A class wants its subclasses to specify the objects it creates. Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate. In a nutshell A "Factory" object creates "Products" for a client; the type of products created depends on the subclass of the factory object used; the client knows only about the factory, not its subclasses

Participants

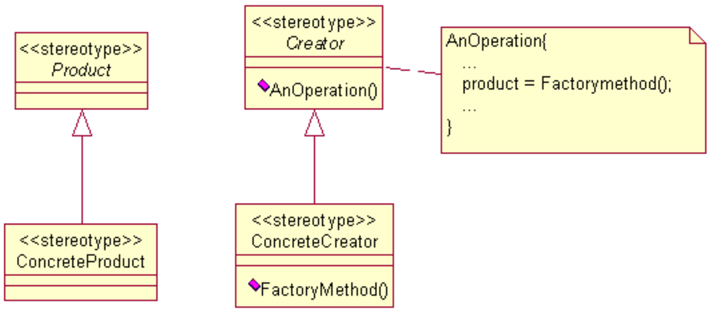
Product :Defines the interface of objects the factory method creates

Concrete Product: Implements the Product

Interface Creator: declares the Factory method which returns an object of type Product

Concrete Creator:overrides the factory method to return an instance of a Concrete Product

Structure



<<stereotype>> Product
<<stereotype>> ConcreteProduct
<<stereotype>> Creator
AnOperation()
<<stereotype>> ConcreteCreator
FactoryMethod()
AnOperation{ ... product = Factorymethod(); ... }

Consequences

Factory methods eliminate the need to bind application-specific classes into your code. Potential disadvantage is that clients must use subclassing in order to create a particular ConcreteProduct. In single-inherited systems, this constrains your partitioning choices.

Also the FM Provides hooks for subclasses Connects parallel class hierarchies S

Implementation

To illustrate how to use factory design pattern with class level implementation, here is a real world example. A company has a website to display testing result from a plain text file. Recently, the company purchased a new machine which produces a binary data file, another new machine on the way, it is possible that one will produce different data file. How to write a system to deal with such change. The website just needs data to display. Your job is to provide the specified data format for the website.

Here comes a solution. Use an interface type to converge the different data file format. The following is a skeleton of implementation.

```
//Let's say the interface is Display
interface Display {

   //load a file
   public void load(String fileName);

   //parse the file and make a consistent data type
   public void formatConsistency();

}

//deal with plain text file
class CSVFile implements Display{

    public void load(String textfile) {
        System.out.println("load from a txt file");
    }
    public void formatConsistency() {
        System.out.println("txt file format changed");
    }
}

//deal with XML format file
class XMLFile implements Display {

    public void load(String xmlfile) {
        System.out.println("load from an xml file");
    }
    public void formatConsistency() {
        System.out.println("xml file format changed");
    }
}

//deal with binary format file
class DBFile implements Display {

    public  void load(String dbfile) {
        System.out.println("load from a db file");
    }
    public void formatConsistency() {
        System.out.println("db file format changed");
    }
}
```

Listing 5.3: Singleton Usage

Usage

//Test the functionality

```
class TestFactory {

    public static void main(String[] args) {
        Display display = null;

        //use a command line data as a trigger
        if (args[0].equals("1"))
           display = new CSVFile();
        else if (args[0].equals("2"))
           display = new XMLFile();
        else if (args[0].equals("3"))
           display = new DBFile();
        else
           System.exit(1);

        //converging code follows
        display.load("");
        display.formatConsistency();
   }
}
```

Listing 5.4: Singleton Usage

Output:

java TestFactory 1 *load from a txt file txt file format changed*

java TestFactory 2 *load from an xml file xml file format changed* **java TestFactory 3** *load from a db file db file format changed*

5.6 GOF patterns and Their Classification

There are totally 23 design patterns in GoF (Gang of Four). All these 23 design patters are mapped to 3 main categories:

1. Creational patterns : Makes Object creation/instantiation job easy.
2. Structural patterns : Makes Objects available to other class by changing the interfaces or contract.
3. Behavioral patterns: Deals with object state changes and it's interaction with other classes/objects.

5.6.1 Creational patterns

Creational patterns involve the construction of new objects by often hiding the constructors in the classes being created, and provides alternate methods to return instances of the desired class.

The most common reason for using a creational pattern is that you need to change the class that's instantiated to fit the situation. Clearly, a constructor in a single class is an inadequate method to return instances of possibly different classes.

Almost always the objects returned are instances of some common superclass.

Object creational patterns generally delegate the actual construction to a different object that is responsible for deciding which class is required and invoking the necessary constructor.

5.6.2 Structural patterns

A structural design pattern serves as a blueprint for how different classes and objects are combined to form larger structures. Unlike creational patterns, which are mostly different ways to fulfill the same fundamental purpose, each structural pattern has a different purpose.

They all involve connections between objects.

In some sense, structural patterns are similar to the simpler concept of data structures. However, structural design patterns also specify the methods that connect objects, not merely the references between them. Furthermore, data structures are fairly static entities. They only describe how data is arranged in the structure.

A structural design pattern also describes how data moves through the pattern. Structural class patterns use inheritance to combine the interfaces or implementations of multiple classes.

Structural object patterns use object composition to combine the implementations of multiple objects. They can combine the interfaces of all the composed objects into one unified interface or they can provide a completely new interface.

5.6.3 Behavioral patterns

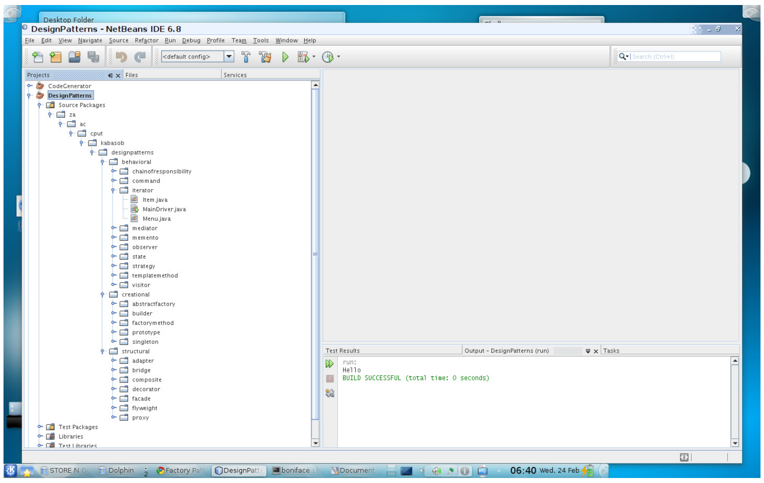
A behavioral pattern explains how objects interact. It describes how different objects and classes send messages to each other to make things happen and how the steps of a task are divided among different objects. It describes how different objects work together to accomplish a task

Where creational patterns mostly describe a moment of time (the instant of creation), and structural patterns describe a more or less static structure, behavioral patterns describe a process or a flow.

Behavioral class patterns use inheritance, subclassing, and polymorphism to adjust the steps taken during a process. Behavioral class patterns focus on changing the exact algorithm used or task performed depending on circumstances.

5.7 GOF Design Patterns Examples in Java

The following section has exercise for the 23 GOF patterns. Create a java application and folders as shown in the diagram below.



```
CodeGenerator
DesignPatterns
  Source Packages
    za
      ac
        cput
          kabasob
            designpatterns
              behavioral
                chainofresponsibility
                command
                iterator
                  Item.java
                  MainDriver.java
                  Menu.java
                mediator
                memento
                observer
                state
                strategy
                templatemethod
                visitor
              creational
                abstractfactory
                builder
                factorymethod
                prototype
                singleton
              structural
                adapter
                bridge
                composite
                decorator
                facade
                flyweight
                proxy
  Test Packages
  Libraries
  Test Libraries
```

```
run:
Hello
BUILD SUCCESSFUL (total time: 0 seconds)
```

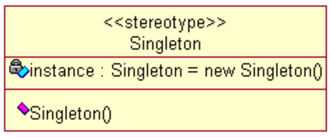
And then do the exercises below. Record your output in the log book or word document. zip your application together with your output and upload to WEBCT

5.7.1 Creational Design Patterns

Creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

All the creational patterns define the best possible way in which an object can be created considering reuse and changeability. These describes the best way to handle instantiation. Hard coding the actual instantiation is a pitfall and should be avoided if reuse and changeability are desired. In such scenarios, we can make use of patterns to give this a more general and flexible approach.

5.7.1.1 Singleton Pattern



A singleton is a class that is instantiated only once. This is typically accomplished by creating a static field in the class representing the class. A static method exists on the class to obtain the instance of the class and is typically named something such as getInstance(). The creation of the object referenced by the static field can be done either when the class is initialized or the first time that getInstance() is called. The singleton class typically has a private constructor to prevent the singleton class from being instantiated via a constructor. Rather, the instance of the singleton is obtained via the static getInstance() method.

The SingletonExample class is an example of a typical singleton class. It contains a private static SingletonExample field. It has a private constructor so that the class can't be instantiated by outside classes. It has a public static getInstance() method that returns the one and only SingletonExample instance. If this instance doesn't already exist, the getInstance() method creates it. The SingletonExample class has a public sayHello() method that can be used to test the singleton.

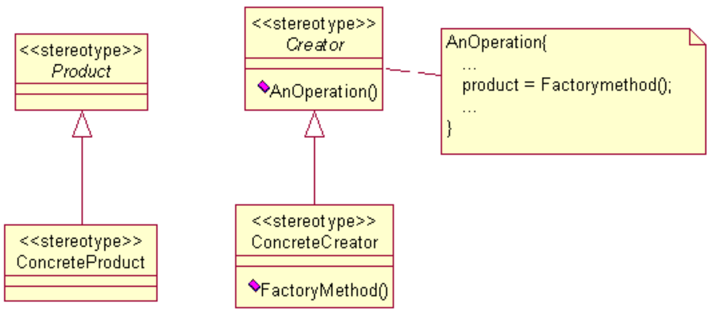
```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package za.ac.cput.kabasob.designpatterns.creational.singleton;

/**
 *
 * @author boniface
 */
public class SingletonExample {

    private static SingletonExample singletonExample = null;

    private SingletonExample() {
    }

    public static SingletonExample getInstance() {
        if (singletonExample == null) {
            singletonExample = new SingletonExample();
        }
        return singletonExample;
    }

    public void sayHello() {
        System.out.println("Hello");
    }
}
```

The MainDriver class obtains a SingletonExample singleton class via the call to the static SingletonExample.getInstance(). We call the sayHello() method on the singleton class.

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package za.ac.cput.kabasob.designpatterns.creational.singleton;

/**
 *
 * @author boniface
 */
public class MainDriver {

    public static void main(String[] args) {
        SingletonExample singletonExample = SingletonExample.getInstance();
        singletonExample.sayHello();
    }
}
```

Record output in your log.

5.7.1.2 Factory Method Pattern



The **Factory Pattern** (also known as the **Factory Method Pattern**) is a creational design pattern. A factory is a Java class that is used to encapsulate object creation code. A factory class instantiates and returns a particular type of object based on data passed to the factory. The different types of objects that are returned from a factory typically are subclasses of a common parent class.

The data passed from the calling code to the factory can be passed either when the factory is created or when the method on the factory is called to create an object. This creational method is often called something such as getInstance or getClass .

As a simple example, let's create an AnimalFactory class that will return an animal object based on some data input. To start, here is an abstract Animal class. The factory will return an instantiated subclass of Animal. Animal has a single abstract method, makeSound().

```
package za.ac.cput.kabasob.designpatterns.creational.factorymethod;

/**
 *
 * @author boniface
 */
public abstract class Animal {
    public abstract String makeSound();
}
```

The Dog class is a subclass of Animal. It implements makeSound() to return ”Woof”.

```
package za.ac.cput.kabasob.designpatterns.creational.factorymethod;
/**
 *
 * @author boniface
 */
public class Dog extends Animal {

    @Override
    public String makeSound() {
        return "Woof";
    }
}
```

The Cat class is a subclass of Animal. It implements makeSound() to return ”Meow”.

```
package za.ac.cput.kabasob.designpatterns.creational.factorymethod;
/**
 *
 * @author boniface
 */
public class Cat extends Animal {
    @Override
    public String makeSound() {
        return "Meow";
    }
}
```

Now, let's implement our factory. We will call our factory's object creation method getAnimal. This method takes a String as a parameter. If the String is "canine", it returns a Dog object. Otherwise, it returns a Cat object.

```
package za.ac.cput.kabasob.designpatterns.creational.factorymethod;

/**
 *
 * @author boniface
 */
public class AnimalFactory {

    public Animal getAnimal(String type) {
        if ("canine".equals(type)) {
            return new Dog();
        } else {
            return new Cat();
        }
    }
}
```

The MainDriver class demonstrates the use of our factory. It creates an AnimalFactory factory. The factory creates an Animal object and then another Animal object. The first object is a Cat and the second object is a Dog. The output of each object's makeSound() method is displayed.

```
package za.ac.cput.kabasob.designpatterns.creational.factorymethod;
/**
 *
 * @author boniface
 */
public class MainDriver {

    public static void main(String[] args) {
        AnimalFactory animalFactory = new AnimalFactory();

        Animal a1 = animalFactory.getAnimal("feline");
        System.out.println("a1 sound: " + a1.makeSound());

        Animal a2 = animalFactory.getAnimal("canine");
        System.out.println("a2 sound: " + a2.makeSound());
    }
}
```

Record your output in your log book.

Notice that the factory has encapsulated our Animal object creation code, thus resulting in clean code in Demo, the class that creates the factory. Additionally, notice the use of polymorphism. We obtain different Animal objects (Cat and Dog) based on data passed to the factory.

Note that it is common to pass data that determines the type of object to be created to the factory when the factory is created (via the factory constructor). However, if multiple objects are being created by the factory, it may make sense to pass this data to the factory's creational method rather than to the constructor, since it might not make sense to create a new factory object each time we wanted to have the factory instantiate a new object.

A factory may also be used in conjunction with the singleton pattern. It is common to have a singleton return a factory instance. To do this, we could replace:

```
AnimalFactory animalFactory = new AnimalFactory();
```

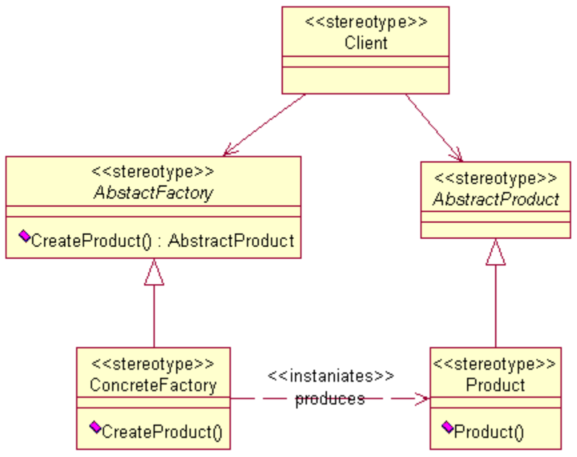
with

```
AnimalFactory animalFactory = AnimalFactory.getAnimalFactoryInstance();
```

In this example, AnimalFactory.getAnimalFactoryInstance() would be implemented to return a static AnimalFactory object. This results in a single factory being instantiated and used rather than requiring a new factory to be instantiated each time the factory needs to be used.

Redo the AnimalFactory to make it a singleton, run the code and include you new modified code in your submission.

5.7.1.3 Abstract Factory Pattern



The abstract factory pattern is a creational design pattern. An abstract factory is a factory that returns factories. Why is this layer of abstraction useful? A normal factory can be used to create sets of related objects. An abstract factory returns factories. Thus, an abstract factory is used to return factories that can be used to create sets of related objects.

As an example, you could have a BMW factory that returns car part objects (seats, air filters, etc) associated with a BMW. You could also have a Toyota factory that returns car part objects associated with a Toyota. We could create an abstract factory that returns these different types of car factories depending on the car that we were interested in. We could then obtain car part objects from the car factory. Via polymorphism, we can use a common interface to get the different factories, and we could could then use a common interface to get the different car parts.

Now, we'll look at the code for another simple example of the abstract factory design pattern.

Our AbstractFactory will return either a MammalFactory or a ReptileFactory via the SpeciesFactory return type. MammalFactory and ReptileFactory are subclasses of Species-Factory.

```java
package za.ac.cput.kabasob.designpatterns.creational.abstractfactory;
/**
 *
 * @author boniface
 */
public class AbstractFactory {

    public SpeciesFactory getSpeciesFactory(String type) {
        if ("mammal".equals(type)) {
            return new MammalFactory();
        } else {
            return new ReptileFactory();
        }
    }
}
```

SpeciesFactory is an abstract class with the getAnimal() abstract method. This method returns an Animal object. The polymorphism of AbstractFactory is achieved because its getSpeciesFactory() method returns a SpeciesFactory, regardless of the actual underlying class. This polymorphism could also be achieved via an interface rather than an abstract class.

```java
package za.ac.cput.kabasob.designpatterns.creational.abstractfactory;
/**
 *
 * @author boniface
 */
public abstract class SpeciesFactory {
    public abstract Animal getAnimal(String type);
}
```

MammalFactory implements getAnimal(). It returns an Animal, which is either a Dog or a Cat.

```java
package za.ac.cput.kabasob.designpatterns.creational.abstractfactory;

/**
 *
 * @author boniface
 */
public class MammalFactory extends SpeciesFactory {

    @Override
    public Animal getAnimal(String type) {
        if ("dog".equals(type)) {
            return new Dog();
        } else {
            return new Cat();
        }
    }
}
```

ReptileFactory implements getAnimal(). It returns an Animal, which is either a Snake or a Tyrannosaurus.

```java
package za.ac.cput.kabasob.designpatterns.creational.abstractfactory;

/**
 *
 * @author boniface
 */
public class ReptileFactory extends SpeciesFactory {

    @Override
    public Animal getAnimal(String type) {
        if ("snake".equals(type)) {
            return new Snake();
        } else {
            return new Tyrannosaurus();
        }
    }
}
```

Animal is an abstract class with the makeSound() abstract method. Subclasses of Animal implement the makeSound() method.

```
package za.ac.cput.kabasob.designpatterns.creational.factorymethod;

/**
 *
 * @author boniface
 */
public abstract class Animal {
    public abstract String makeSound();
}
```

Cat Class

```
package za.ac.cput.kabasob.designpatterns.creational.factorymethod;
/**
 *
 * @author boniface
 */
public class Cat extends Animal {

    @Override
    public String makeSound() {
        return "Meow";
    }
}
```

The Dog class

```
package za.ac.cput.kabasob.designpatterns.creational.factorymethod;
/**
 *
 * @author boniface
 */
public class Dog extends Animal {

    @Override
    public String makeSound() {
        return "Woof";
    }
}
```

The Snake Class

```
package za.ac.cput.kabasob.designpatterns.creational.factorymethod;
/**
 *
 * @author boniface
 */
public class Cat extends Animal {

    @Override
    public String makeSound() {
        return "Hiss";
    }
}
```

The Tyrannosaurus class is shown here.

```
package za.ac.cput.kabasob.designpatterns.creational.factorymethod;
/**
 *
 * @author boniface
 */
public class Tyrannosaurus extends Animal {

    @Override
    public String makeSound() {
        return "Roar";
    }
}
```

The DriverMain class contains our main() method. It creates an AbstractFactory object. From the AbstractFactory, we obtain a SpeciesFactory (a ReptileFactory) and get two

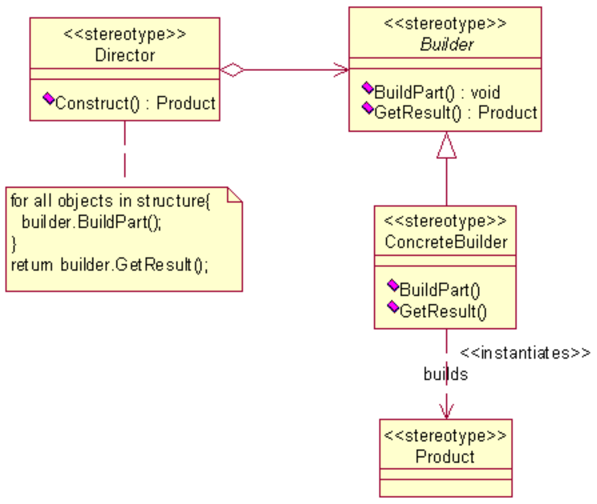
Animal objects (Tyrannosaurus and Snake) from the SpeciesFactory. After this, we obtain another SpeciesFactory (a MammalFactory) and then obtain two more Animal objects (Dog and Cat).

```java
package za.ac.cput.kabasob.designpatterns.creational.abstractfactory;

/**
 *
 * @author boniface
 */
public abstract class MainDriver {

    public static void main(String[] args) {
        AbstractFactory abstractFactory = new AbstractFactory();

        SpeciesFactory speciesFactory1 = abstractFactory.getSpeciesFactory("reptile");
        Animal a1 = speciesFactory1.getAnimal("tyrannosaurus");
        System.out.println("a1 sound: " + a1.makeSound());
        Animal a2 = speciesFactory1.getAnimal("snake");
        System.out.println("a2 sound: " + a2.makeSound());

        SpeciesFactory speciesFactory2 = abstractFactory.getSpeciesFactory("mammal");
        Animal a3 = speciesFactory2.getAnimal("dog");
        System.out.println("a3 sound: " + a3.makeSound());
        Animal a4 = speciesFactory2.getAnimal("cat");
        System.out.println("a4 sound: " + a4.makeSound());

    }
}
```

Execute the code and record output in your log book.

Notice the use of polymorphism. We obtain different factories via the common Species-Factory superclass. We also obtain different animals via the common Animal superclass.

5.7.1.4 Builder Pattern



The builder pattern is a creational design pattern used to assemble complex objects. With the builder pattern, the same object construction process can be used to create different objects. The builder has 4 main parts: a Builder, Concrete Builders, a Director, and a Product.

A Builder is an interface (or abstract class) that is implemented (or extended) by Concrete Builders. The Builder interface sets forth the actions (methods) involved in assembling a Product object. It also has a method for retrieving the Product object (ie, getProduct()). The Product object is the object that gets assembled in the builder pattern.

Concrete Builders implement the Builder interface (or extend the Builder abstract class). A Concrete Builder is responsible for creating and assembling a Product object. Different Concrete Builders create and assemble Product objects differently.

A Director object is responsible for constructing a Product. It does this via the Builder interface to a Concrete Builder. It constructs a Product via the various Builder methods.

There are various uses of the builder pattern. For one, if we'd like the construction process to remain the same but we'd like to create a different type of Product, we can create a new Concrete Builder and pass this to the same Director. If we'd like to alter the construction process, we can modify the Director to use a different construction process.

Now, let's look at an example of the builder pattern. Our example will build different kinds of restaurant meals.

First off, our Product will be a Meal class, which represents food items in a meal. It represents a drink, main course, and side item.

```java
package za.ac.cput.kabasob.designpatterns.creational.builder;

/**
 *
 * @author boniface
 */
public class Meal {

    private String drink;
    private String mainCourse;
    private String side;

    public String getDrink() {
        return drink;
    }

    public void setDrink(String drink) {
        this.drink = drink;
    }

    public String getMainCourse() {
        return mainCourse;
    }

    public void setMainCourse(String mainCourse) {
        this.mainCourse = mainCourse;
    }

    public String getSide() {
        return side;
    }

    public void setSide(String side) {
        this.side = side;
    }

    public String toString() {
        return "drink:" + drink + ", main course:" + mainCourse + ", side:" + side;
    }
}
```

Our Builder interface is MealBuilder. It features methods used to build a meal and a method to retrieve the meal.

```java
package za.ac.cput.kabasob.designpatterns.creational.builder;

/**
 *
 * @author boniface
 */
public interface MealBuilder {
    public void buildDrink();

    public void buildMainCourse();

    public void buildSide();

    public Meal getMeal();
}
```

Our first Concrete Builder is ItalianMealBuilder. Its constructor creates a meal. Its methods are implemented to build the various parts of the meal. It returns the meal via getMeal().

```
package za.ac.cput.kabasob.designpatterns.creational.builder;

/**
 *
 * @author boniface
 */
public class ItalianMealBuilder implements MealBuilder {

    private Meal meal;

    public ItalianMealBuilder() {
        meal = new Meal();
    }

    @Override
    public void buildDrink() {
        meal.setDrink("red wine");
    }

    @Override
    public void buildMainCourse() {
        meal.setMainCourse("pizza");
    }

    @Override
    public void buildSide() {
        meal.setSide("bread");
    }

    @Override
    public Meal getMeal() {
        return meal;
    }
}
```

Our second Concrete Builder is JapaneseMealBuilder. Its constructor creates a meal. Its methods are implemented to build the various parts of a Japanese meal. It returns the meal via getMeal().

```
package za.ac.cput.kabasob.designpatterns.creational.builder;

/**
 *
 * @author boniface
 */
public class JapaneseMealBuilder implements MealBuilder {

    private Meal meal;

    public JapaneseMealBuilder() {
        meal = new Meal();
    }

    @Override
    public void buildDrink() {
        meal.setDrink("sake");
    }

    @Override
    public void buildMainCourse() {
        meal.setMainCourse("chicken teriyaki");
    }

    @Override
    public void buildSide() {
        meal.setSide("miso soup");
    }

    @Override
    public Meal getMeal() {
        return meal;
    }
}
```

Our Director class is MealDirector. It takes a MealBuilder as a parameter in its constructor. Thus, a different type of meal will be assembled by the MealDirector depending on the Concrete Builder passed in to the constructor. The assembly of the meal (Product) takes place in the constructMeal() method of the Director. This method spells out the parts of the meal that will be assembled.

```
package za.ac.cput.kabasob.designpatterns.creational.builder;

/**
 *
 * @author boniface
 */
public class MealDirector {

    private MealBuilder mealBuilder = null;

    public MealDirector(MealBuilder mealBuilder) {
        this.mealBuilder = mealBuilder;
    }
    public void constructMeal() {
        mealBuilder.buildDrink();
        mealBuilder.buildMainCourse();
        mealBuilder.buildSide();
    }
    public Meal getMeal() {
        return mealBuilder.getMeal();
    }
}
```

The MainDriver class lets us demonstrate our builder pattern. First, our director builds an Italian meal. An ItalianMealBuilder is passed to the MealDirector's constructor. The meal is constructed via mealDirector.constructMeal(). The meal is obtained from mealDirector via mealDirector.getMeal(). The Italian meal is displayed. After this, we perform the same process to build and display a Japanese meal.

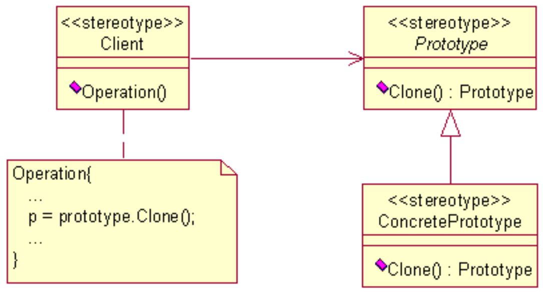
```
package za.ac.cput.kabasob.designpatterns.creational.builder;

/**
 *
 * @author boniface
 */
public class MainDriver {

    public static void main(String[] args) {

        MealBuilder mealBuilder = new ItalianMealBuilder();
        MealDirector mealDirector = new MealDirector(mealBuilder);
        mealDirector.constructMeal();
        Meal meal = mealDirector.getMeal();
        System.out.println("meal is: " + meal);

        mealBuilder = new JapaneseMealBuilder();
        mealDirector = new MealDirector(mealBuilder);
        mealDirector.constructMeal();
        meal = mealDirector.getMeal();
        System.out.println("meal is: " + meal);
    }
}
```

Record Output in your log book

Notice that if we'd like to create a new type of meal, we can do so by implementing a new Concrete Builder (ie, SwedishMealBuilder, FrenchMealBuilder, etc), which we'd pass to the MealDirector. If we'd like the meal to be constructed of different parts (ie, no drink), we can alter the construction process in MealDirector. Thus, construction process has been separated to the Director, and the data representation is controlled by the Concrete Builders that implement the Builder interface.

Create a two extra meals for your Example and include the code in your submission

5.7.1.5 Prototype Pattern



The prototype pattern is a creational design pattern. In the prototype pattern, a new object is created by cloning an existing object. In Java, the clone() method is an implementation of this design pattern. The prototype pattern can be a useful way of creating copies of objects.

One example of how this can be useful is if an original object is created with a resource such as a data stream that may not be available at the time that a clone of the object is needed.

Another example is if the original object creation involves a significant time commitment, such as reading data from a database or over a network. An added benefit of the prototype pattern is that it can reduce class proliferation in a project by avoiding factory proliferation.

Normally in Java, if you'd like to use cloning (ie, the prototype pattern), you can utilize the clone() method and the Cloneable interface. By default, clone() performs a shallow copy

However, we can implement our own prototype pattern. To do so, we'll create a Prototype interface that features a doClone() method.

```
public interface Prototype {

    public Prototype doClone();

}
```

The Person class implements the doClone() method. This method creates a new Person object and clones the name field. It returns the newly cloned Person object.

```
package za.ac.cput.kabasob.designpatterns.creational.prototype;
/**
 *
 * @author boniface
 */
public class Person implements Prototype {

    String name;

    public Person(String name) {
        this.name = name;
    }
    @Override
    public Prototype doClone() {
        return new Person(name);
    }

    public String toString() {
        return "This person is named " + name;
    }
}
```

The Dog class also implements the doClone() method. This method creates a new Dog object and clones the sound field. The cloned Dog object is returned.

```
package za.ac.cput.kabasob.designpatterns.creational.prototype;
/**
 *
 * @author boniface
 */
public class Dog implements Prototype {

    String sound;

    public Dog(String sound) {
        this.sound = sound;
    }

    @Override
    public Prototype doClone() {
        return new Dog(sound);
    }

    public String toString() {
        return "This dog says " + sound;
    }
}
```

The MainDriver creates a Person object and then clones it to a second Person object. It then creates a Dog object and clones it to a second Dog object.

```
package za.ac.cput.kabasob.designpatterns.creational.prototype;

/**
 *
 * @author boniface
 */
public class MainDriver {

    public static void main(String[] args) {

        Person person1 = new Person("Fred");
        System.out.println("person 1:" + person1);
        Person person2 = (Person) person1.doClone();
        System.out.println("person 2:" + person2);

        Dog dog1 = new Dog("Wooof!");
        System.out.println("dog 1:" + dog1);
        Dog dog2 = (Dog) dog1.doClone();
        System.out.println("dog 2:" + dog2);

    }
}
```

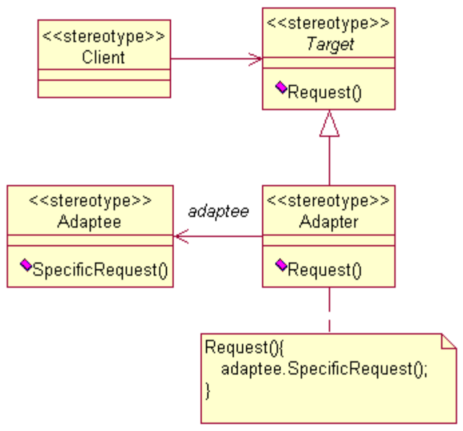
Record output in your log book

5.7.2 Structural Design Patterns

In software engineering, structural design patterns are design patterns that ease the design by identifying a simple way to realize relationships between entities.

Structural Patterns describe how objects and classes can be combined to form structures. We distinguish between object patterns and class patterns. The difference is that class patterns describe relationships and structures with the help of inheritance. Object patterns, on other hand, describe how objects can be associated and aggregated to form larger, more complex structures.

5.7.2.1 Adapter Pattern



The adapter pattern is a structural design pattern. In the adapter pattern, a wrapper class (ie, the adapter) is used to translate requests from it to another class (ie, the adaptee). In effect, an adapter provides particular interactions with an adaptee that are not offered directly by the adaptee.

The adapter pattern takes two forms. In the first form, a "class adapter" utilizes inheritance. The class adapter extends the adaptee class and adds the desired methods to the adapter. These methods can be declared in an interface (ie, the "target" interface).

In the second form, an "object adapter" utilizes composition. The object adapter contains an adaptee and implements the target interface to interact with the adaptee.

Now let's look at simple examples of a class adapter and an object adapter. First, we have an adaptee class named CelciusReporter. It stores a temperature value in Celcius.

```
package za.ac.cput.kabasob.designpatterns.structural.adapter;

/**
 *
 * @author boniface
 */
public class CelciusReporter {

    double temperatureInC;

    public CelciusReporter() {
    }

    public double getTemperature() {
        return temperatureInC;
    }

    public void setTemperature(double temperatureInC) {
        this.temperatureInC = temperatureInC;
    }
}
```

Here is our target interface that will be implemented by our adapter. It defines actions that our adapter will perform.

```
package za.ac.cput.kabasob.designpatterns.structural.adapter;

/**
 *
 * @author boniface
 */
public interface TemperatureInfo {

    public double getTemperatureInF();

    public void setTemperatureInF(double temperatureInF);

    public double getTemperatureInC();

    public void setTemperatureInC(double temperatureInC);
}
```

TemperatureClassReporter is a class adapter. It extends CelciusReporter (the adaptee) and implements TemperatureInfo (the target interface). If a temperature is in Celcius, TemperatureClassReporter utilizes the temperatureInC value from CelciusReporter. Fahrenheit requests are internally handled in Celcius.

```
package za.ac.cput.kabasob.designpatterns.structural.adapter;

/**
 *
 * @author boniface
 */
// example of a class adapter
public class TemperatureClassReporter extends CelciusReporter implements TemperatureInfo {

    @Override
    public double getTemperatureInC() {
        return temperatureInC;
    }

    @Override
    public double getTemperatureInF() {
        return cToF(temperatureInC);
    }

    @Override
    public void setTemperatureInC(double temperatureInC) {
        this.temperatureInC = temperatureInC;
    }

    @Override
    public void setTemperatureInF(double temperatureInF) {
        this.temperatureInC = fToC(temperatureInF);
    }

    private double fToC(double f) {
        return ((f - 32) * 5 / 9);
    }

    private double cToF(double c) {
        return ((c * 9 / 5) + 32);
    }

}
```

TemperatureObjectReporter is an object adapter. It is similar in functionality to TemperatureClassReporter, except that it utilizes composition for the CelciusReporter rather than inheritance.

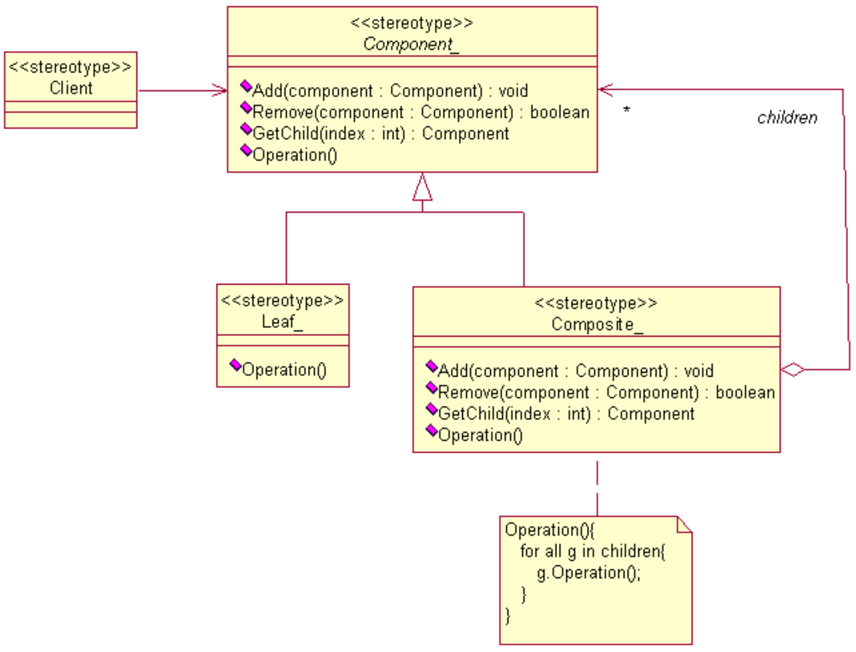
```
package za.ac.cput.kabasob.designpatterns.structural.adapter;

/**
 *
 * @author boniface
 */

// example of an object adapter
public class TemperatureObjectReporter implements TemperatureInfo {

    CelciusReporter celciusReporter;

    public TemperatureObjectReporter() {
        celciusReporter = new CelciusReporter();
    }

    @Override
    public double getTemperatureInC() {
        return celciusReporter.getTemperature();
    }

    @Override
    public double getTemperatureInF() {
        return cToF(celciusReporter.getTemperature());
    }

    @Override
    public void setTemperatureInC(double temperatureInC) {
        celciusReporter.setTemperature(temperatureInC);
    }

    @Override
    public void setTemperatureInF(double temperatureInF) {
        celciusReporter.setTemperature(fToC(temperatureInF));
    }

    private double fToC(double f) {
        return ((f − 32) ∗ 5 / 9);
    }

    private double cToF(double c) {
        return ((c ∗ 9 / 5) + 32);
    }

}
```

The MainDriver class is a client class that demonstrates the adapter pattern. First, it creates a TemperatureClassReporter object and references it via a TemperatureInfo reference. It demonstrates calls to the class adapter via the TemperatureInfo interface. After this, it creates a TemperatureObjectReporter object and references it via the same TemperatureInfo reference. It then demonstrates calls to the object adapter.

```
package za.ac.cput.kabasob.designpatterns.structural.adapter;

/**
 *
 * @author boniface
 */
public class DriverMain {

    public static void main(String[] args) {

        // class adapter
        System.out.println("class adapter test");
        TemperatureInfo tempInfo = new TemperatureClassReporter();
        testTempInfo(tempInfo);

        // object adapter
        System.out.println("\nobject adapter test");
        tempInfo = new TemperatureObjectReporter();
        testTempInfo(tempInfo);

    }

    public static void testTempInfo(TemperatureInfo tempInfo) {
        tempInfo.setTemperatureInC(0);
        System.out.println("temp in C:" + tempInfo.getTemperatureInC());
        System.out.println("temp in F:" + tempInfo.getTemperatureInF());

        tempInfo.setTemperatureInF(85);
        System.out.println("temp in C:" + tempInfo.getTemperatureInC());
        System.out.println("temp in F:" + tempInfo.getTemperatureInF());
    }
}
```

Write output in your log book

5.7.2.2 Composite Pattern



<<stereotype>>
Client
<<stereotype>>
Component_
Add(component : Component) : void
Remove(component : Component) : boolean
GetChild(index : int) : Component
Operation()
*
children
<<stereotype>>
Leaf_
Operation()
<<stereotype>>
Composite_
Add(component : Component) : void
Remove(component : Component) : boolean
GetChild(index : int) : Component
Operation()
Operation(){
for all g in children{
g.Operation();
}
}

The composite pattern is a structural design pattern. In the composite pattern, a tree structure exists where identical operations can be performed on leaves and nodes. A node in a tree is a class that can have children. A node class is a 'composite' class. A leaf in a tree is a 'primitive' class that does not have children. The children of a composite can be leaves or other composites.

The leaf class and the composite class share a common 'component' interface that defines the common operations that can be performed on leaves and composites. When an operation on a composite is performed, this operation is performed on all children of the composite, whether they are leaves or composites. Thus, the composite pattern can be used to perform common operations on the objects that compose a tree.

The Gang of Four description of the composite pattern defines a client's interaction with the tree struture via a Component interface, where this interface includes the common operations on the composite and leaf classes, and the add/remove/get operations on the composites of the tree. This seems slightly awkward since a leaf does not implement the add/remove/get operations. In Java it makes more sense to define the common leaf/composite operations on a Component interface, but to put the add/remove/get composite operations in a separate interface or to simply implement them in the composite class.

Now we'll look at an example of the composite pattern. First, we'll declare a Component interface that declares the operations that are common for the composite class and the leaf class. This allows us to perform operations on composites and leaves using one standard interface.

```
package za.ac.cput.kabasob.designpatterns.structural.composite;

/**
 *
 * @author boniface
 */
public interface Component {

    public void sayHello();

    public void sayGoodbye();
}
```

The Leaf class has a name field and implements the sayHello() and sayGoodbye() methods of the Component interface by outputting messages to standard output.

```
package za.ac.cput.kabasob.designpatterns.structural.composite;

/**
 *
 * @author boniface
 */
public class Leaf implements Component {

    String name;

    public Leaf(String name) {
        this.name = name;
    }

    @Override
    public void sayHello() {
        System.out.println(name + " leaf says hello");
    }

    @Override
    public void sayGoodbye() {
        System.out.println(name + " leaf says goodbye");
    }
}
```

The Composite class implements the Component interface. It implements the sayHello() and sayGoodbye() methods by calling these same methods on all of its children, which are Components (since they can be both Leaf objects and Composite objects, which both implement the Component interface).

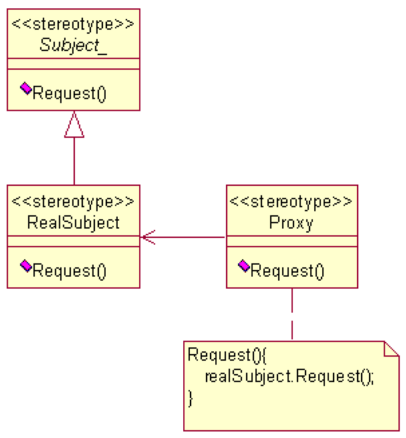
```
package za.ac.cput.kabasob.designpatterns.structural.composite;

import java.util.ArrayList;
import java.util.List;

/**
 *
 * @author boniface
 */
public class Composite implements Component {

    List<Component> components = new ArrayList<Component>();

    @Override
    public void sayHello() {
        for (Component component : components) {
            component.sayHello();
        }
    }

    @Override
    public void sayGoodbye() {
        for (Component component : components) {
            component.sayGoodbye();
        }
    }

    public void add(Component component) {
        components.add(component);
    }

    public void remove(Component component) {
        components.remove(component);
    }

    public List<Component> getComponents() {
        return components;
    }

    public Component getComponent(int index) {
        return components.get(index);
    }
}
```

The MainDriver class demonstrates the composite pattern. It creates 5 Leaf objects. It adds two of these two a Composite object and two of these to another Composite object. It adds these two Composite objects and the last Leaf object to another Composite object. It calls sayHello() on leaf1, then sayHello() on composite1, then sayHello() on composite2, and then sayGoodbye() on composite3.

```java
package za.ac.cput.kabasob.designpatterns.structural.composite;

/**
 *
 * @author boniface
 */
public class MainDriver {

    public static void main(String[] args) {

        Leaf leaf1 = new Leaf("Bob");
        Leaf leaf2 = new Leaf("Fred");
        Leaf leaf3 = new Leaf("Sue");
        Leaf leaf4 = new Leaf("Ellen");
        Leaf leaf5 = new Leaf("Joe");

        Composite composite1 = new Composite();
        composite1.add(leaf1);
        composite1.add(leaf2);

        Composite composite2 = new Composite();
        composite2.add(leaf3);
        composite2.add(leaf4);

        Composite composite3 = new Composite();
        composite3.add(composite1);
        composite3.add(composite2);
        composite3.add(leaf5);

        System.out.println("Calling 'sayHello' on leaf1");
        leaf1.sayHello();

        System.out.println("\nCalling 'sayHello' on composite1");
        composite1.sayHello();

        System.out.println("\nCalling 'sayHello' on composite2");
        composite2.sayHello();

        System.out.println("\nCalling 'sayGoodbye' on composite3");
        composite3.sayGoodbye();

    }
}
```

Write your output in the log book

5.7.2.3 Proxy Pattern



The proxy pattern is a structural design pattern. In the proxy pattern, a proxy class is used to control access to another class. The reasons for this control can vary. As one example, a proxy may avoid instantiation of an object until the object is needed. This can be useful if the object requires a lot of time or resources to create. Another reason to use a proxy is to control access rights to an object. A client request may require certain credentials in order to access the object.

Now, we'll look at an example of the proxy pattern. First, we'll create an abstract class called Thing with a basic sayHello() message that includes the date/time that the message is displayed.

```
package za.ac.cput.kabasob.designpatterns.structural.proxy;

import java.util.Date;

/**
 *
 * @author boniface
 */
public abstract class Thing {

    public void sayHello() {
        System.out.println(this.getClass().getSimpleName() + " says howdy at " + new Date()) ↩
    ;
    }
}
```

FastThing subclasses Thing.

```
package za.ac.cput.kabasob.designpatterns.structural.proxy;

/**
 *
 * @author boniface
 */
public class FastThing extends Thing {
    public FastThing() {
    }
}
```

SlowThing also subclasses Thing. However, its constructor takes 5 seconds to execute.

```
package za.ac.cput.kabasob.designpatterns.structural.proxy;
```

```
/**
 *
 * @author boniface
 */
public class SlowThing extends Thing {

    public SlowThing() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

The Proxy class is a proxy to a SlowThing object. Since a SlowThing object takes 5 seconds to create, we'll use a proxy to a SlowThing so that a SlowThing object is only created on demand. This occurs when the proxy's sayHello() method is executed. It instantiates a SlowThing object if it doesn't already exist and then calls sayHello() on the SlowThing object.

```
package za.ac.cput.kabasob.designpatterns.structural.proxy;

import java.util.Date;

/**
 *
 * @author boniface
 */
public class Proxy {

    SlowThing slowThing;

    public Proxy() {
        System.out.println("Creating proxy at " + new Date());
    }

    public void sayHello() {
        if (slowThing == null) {
            slowThing = new SlowThing();
        }
        slowThing.sayHello();
    }
}
```

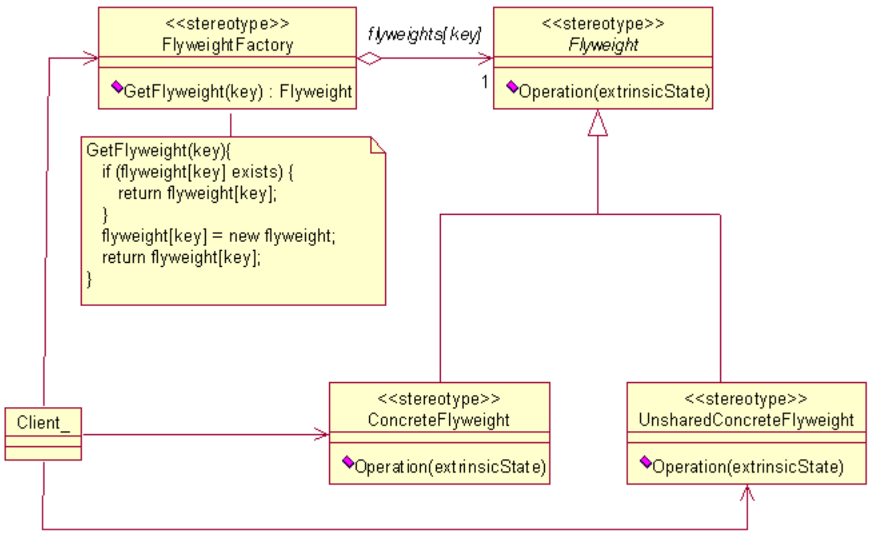
The MainDriver class demonstrates the use of our proxy. It creates a Proxy object and then creates a FastThing object and calls sayHello() on the FastThing object. Next, it calls sayHello() on the Proxy object.

```
package za.ac.cput.kabasob.designpatterns.structural.proxy;

/**
 *
 * @author boniface
 */
public class MainDriver {

    public static void main(String[] args) {
        Proxy proxy = new Proxy();
        FastThing fastThing = new FastThing();
        fastThing.sayHello();
        proxy.sayHello();

    }
}
```

Record your output in the log book.

From the output, notice that creating the Proxy object and calling sayHello() on the FastThing object occurred at a specific time , and calling sayHello() on the Proxy object did not call sayHello() on the SlowThing object until another time. We can see that the SlowThing creation was a time-consuming process. However, this did not slow down the execution of our application until the SlowThing object was actually required. We can see here that the proxy pattern avoids the creation of time-consuming objects until they are actually needed.

5.7.2.4 Flyweight Pattern



<<stereotype>>
FlyweightFactory
GetFlyweight(key) : Flyweight
flyweights[key]
1
<<stereotype>>
Flyweight
Operation(extrinsicState)
GetFlyweight(key){
if (flyweight[key] exists) {
return flyweight[key];
}
flyweight[key] = new flyweight;
return flyweight[key];
}
Client_
<<stereotype>>
ConcreteFlyweight
Operation(extrinsicState)
<<stereotype>>
UnsharedConcreteFlyweight
Operation(extrinsicState)

The flyweight pattern is a structural design pattern. In the flyweight pattern, instead of creating large numbers of similar objects, objects are reused. This can be used to reduce memory requirements and instantiation time and related costs. Similarities between objects are stored inside of the objects, and differences are moved outside of the objects and placed in client code. These differences are passed in to the objects when needed via method calls on the objects. A Flyweight interface declares these methods. A Concrete Flyweight class implements the Flyweight interface which is used to perform operations based on external state and it also stores common state. A Flyweight Factory is used create and return Flyweight objects.

Now, let's look at an example of the flyweight design pattern. We'll create a Flyweight interface with a doMath() method that will be used to perform a mathematical operation on two integers passed in as parameters.

```java
package za.ac.cput.kabasob.designpatterns.structural.flyweight;
/**
 *
 * @author boniface
 */
public interface Flyweight {

    public void doMath(int a, int b);
}
```

The FlyweightAdder class is a concrete flyweight class. It contains an "operation" field that is used to store the name of an operation that is common to adder flyweights. Notice the call to Thread.sleep(3000). This simulates a construction process that is expensive in terms of time. Each FlyweightAdder object that is created takes 3 seconds to create, so we definitely want to minimize the number of flyweight objects that are created. The doMath() method is implemented. It displays the common "operation" field and displays the addition of a and b, which are external state values that are passed in and used by the FlyweightAdder when doMath() is executed.

```java
package za.ac.cput.kabasob.designpatterns.structural.flyweight;

/**
 *
 * @author boniface
 */
public class FlyweightAdder implements Flyweight {

    String operation;

    public FlyweightAdder() {
        operation = "adding";
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void doMath(int a, int b) {
        System.out.println(operation + " " + a + " and " + b + ": " + (a + b));
    }
}
```

The FlyweightMultiplier class is similar to the FlyweightAdder class, except that it performs multiplication rather than addition.

```java
package za.ac.cput.kabasob.designpatterns.structural.flyweight;

/**
 *
 * @author boniface
 */
public class FlyweightMultiplier implements Flyweight {

    String operation;

    public FlyweightMultiplier() {
        operation = "multiplying";
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void doMath(int a, int b) {
        System.out.println(operation + " " + a + " and " + b + ": " + (a * b));
    }

}
```

The FlyweightFactory class is our flyweight factory. It utilizes the singleton pattern so that we only have once instance of the factory, which we obtain via its static getInstance() method. The FlyweightFactory creates a hashmap pool of flyweights. If a request is made for a flyweight object and that object doesn't exist, it is created and placed in the flyweight pool. The flyweight pool of the FlyweightFactory stores all the instances of the different types of flyweights (ie, FlyweightAdder object, FlyweightMultiplier object, etc). Thus, only one instance of each type is created, and this occurs on-demand.

```java
package za.ac.cput.kabasob.designpatterns.structural.flyweight;

import java.util.HashMap;
import java.util.Map;

/**
 *
 * @author boniface
 */
public class FlyweightFactory {
    private static FlyweightFactory flyweightFactory;
    private Map<String, Flyweight> flyweightPool;

    private FlyweightFactory() {
        flyweightPool = new HashMap<String, Flyweight>();
    }
    public static FlyweightFactory getInstance() {
        if (flyweightFactory == null) {
            flyweightFactory = new FlyweightFactory();
        }
        return flyweightFactory;
    }

    public Flyweight getFlyweight(String key) {
        if (flyweightPool.containsKey(key)) {
            return flyweightPool.get(key);
        } else {
            Flyweight flyweight;
            if ("add".equals(key)) {
                flyweight = new FlyweightAdder();
            } else {
                flyweight = new FlyweightMultiplier();
            }
            flyweightPool.put(key, flyweight);
            return flyweight;
        }
    }
}
```

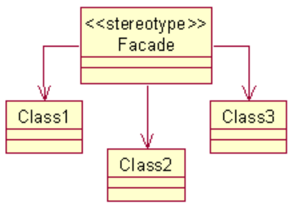
The MainDriver class demonstrates our flyweight pattern. It obtains a FlyweightFactory object via FlyweightFactory.getInstance(). After this, in a loop, it obtains a FlyweightAdder from the FlyweightFactory and calls its doMath() operation with the current loop index as the two parameter values. Next, it does the same thing with a FlyweightMultiplier.

```
package za.ac.cput.kabasob.designpatterns.structural.flyweight;


/**
 *
 * @author boniface
 */

 public class MainDriver {

    public static void main(String[] args) {

        FlyweightFactory flyweightFactory = FlyweightFactory.getInstance();

        for (int i = 0; i < 5; i++) {
            Flyweight flyweightAdder = flyweightFactory.getFlyweight("add");
            flyweightAdder.doMath(i, i);

            Flyweight flyweightMultiplier = flyweightFactory.getFlyweight("multiply");
            flyweightMultiplier.doMath(i, i);
        }
    }
}
```

Write output in your log book

When executing MainDriver, we see that it takes some seconds to create the FlyweightAdder object and the same thing happens for the FlyweightMultiplier object. After this, no more delays occur since we keep reusing the flyweight objects rather than instantiating new objects. This highlights how the flyweight pattern can be used to minimize resource requirements by avoiding unnecessary object instantiations for similar objects.

5.7.2.5 Facade Pattern



The facade pattern is a structural design pattern. In the facade pattern, a facade classes is used to provide a single interface to set of classes. The facade simplifies a clients interaction with a complex system by localizing the interactions into a single interface. As a result, the client can interact with a single object rather than being required to interact directly in complicated ways with the objects that make up the subsystem.

As an example, supposed we have three horribly written classes. For based on the class and method names (and the lack of documentation), it would be very difficult for a client to interact with these classes.

Class1's doSomethingComplicated() method takes an integer and returns its cube.

```java
package za.ac.cput.kabasob.designpatterns.structural.facade;

/**
 *
 * @author boniface
 */
public class Class1 {

    public int doSomethingComplicated(int x) {
        return x * x * x;
    }
}
```

Class2's doAnotherThing() method doubles the cube of an integer and returns it.

```java
package za.ac.cput.kabasob.designpatterns.structural.facade;

/**
 *
 * @author boniface
 */
public class Class2 {

    public int doAnotherThing(Class1 class1, int x) {
        return 2 * class1.doSomethingComplicated(x);
    }
}
```

Class3's doMoreStuff() takes a Class1 object, a Class2 object, and an integer and returns twice the sixth power of the integer.

```java
package za.ac.cput.kabasob.designpatterns.structural.facade;

/**
 *
 * @author boniface
 */
public class Class3 {

    public int doMoreStuff(Class1 class1, Class2 class2, int x) {
        return class1.doSomethingComplicated(x) * class2.doAnotherThing(class1, x);
    }
}
```

For a client unfamiliar with Class1, Class2, and Class3, it would be very difficult to figure out how to interact with these classes. The classes interact and perform tasks in unclear ways. As a result, we need to be able to simplify interaction with this system of classes so that clients can interact with these classes in a simple, standardized manner.

We do this with the Facade class. The Facade class has three methods: cubeX(), cubeXTimes2(), and xToSixthPowerTimes2(). The names of these methods clearly indicate what they do, and these methods hide the interactions of Class1, Class2, and Class3 from client code.

```java
package za.ac.cput.kabasob.designpatterns.structural.facade;

/**
 *
 * @author boniface
 */
public class Facade {

    public int cubeX(int x) {
        Class1 class1 = new Class1();
        return class1.doSomethingComplicated(x);
    }

    public int cubeXTimes2(int x) {
        Class1 class1 = new Class1();
        Class2 class2 = new Class2();
        return class2.doAnotherThing(class1, x);
    }

    public int xToSixthPowerTimes2(int x) {
        Class1 class1 = new Class1();
        Class2 class2 = new Class2();
        Class3 class3 = new Class3();
        return class3.doMoreStuff(class1, class2, x);
    }
}
```

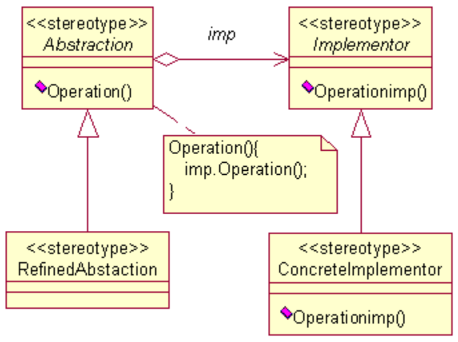
The MainDriver class contains our client code. It creates a Facade object and then calls its three methods with a parameter value of 3. It displays the returned results.

```java
package za.ac.cput.kabasob.designpatterns.structural.facade;

/**
 *
 * @author boniface
 */
public class MainDriver {

    public static void main(String[] args) {

        Facade facade = new Facade();
        int x = 3;
        System.out.println("Cube of " + x + ":" + facade.cubeX(3));
        System.out.println("Cube of " + x + " times 2:" + facade.cubeXTimes2(3));
        System.out.println(x + " to sixth power times 2:" + facade.xToSixthPowerTimes2(3));

    }
}
```

Record your output in your log book

This example demonstrates how the facade pattern can be used to simplify interactions with a system of classes by providing a single point of interaction with the subsystem and hiding the complex details of subsystem interactions from client code. This is accomplished with a Facade class.

5.7.2.6 Bridge Pattern



The bridge pattern is a structural design pattern. In the bridge pattern, we separate an abstraction and its implementation and develop separate inheritance structures for both the abstraction and the implementor. The abstraction is an interface or abstract class, and the implementor is likewise an interface or abstract class. The abstraction contains a reference to the implementor. Children of the abstraction are referred to as refined abstractions, and children of the implementor are concrete implementors. Since we can change the reference to the implementor in the abstraction, we are able to change the abstraction's implementor at run-time. Changes to the implementor do not affect client code.

The bridge pattern can be demonstrated with an example. Suppose we have a Vehicle class. We can extract out the implementation of the engine into an Engine class. We can reference this Engine implementor in our Vehicle via an Engine field. We'll declare Vehicle to be an abstract class. Subclasses of Vehicle need to implement the drive() method. Notice that the Engine reference can be changed via the setEngine() method.

```java
package za.ac.cput.kabasob.designpatterns.structural.bridge;

/**
 *
 * @author boniface
 */
public abstract class Vehicle {

    Engine engine;
    int weightInKilos;

    public abstract void drive();

    public void setEngine(Engine engine) {
        this.engine = engine;
    }

    public void reportOnSpeed(int horsepower) {
        int ratio = weightInKilos / horsepower;
        if (ratio < 3) {
            System.out.println("The vehicle is going at a fast speed.");
        } else if ((ratio >= 3) && (ratio < 8)) {
            System.out.println("The vehicle is going an average speed.");
        } else {
            System.out.println("The vehicle is going at a slow speed.");
        }
    }

}
```

BigBus is a subclass of Vehicle. It has a weight of 3000 kg. Its drive() method displays a message, calls the engine's go() method, and then calls reportOnSpeed() with the horsepower of the engine to report on how fast the vehicle is moving.

```
package za.ac.cput.kabasob.designpatterns.structural.bridge;

/**
 *
 * @author boniface
 */
public class BigBus extends Vehicle {

    public BigBus(Engine engine) {
        this.weightInKilos = 3000;
        this.engine = engine;
    }

    @Override
    public void drive() {
        System.out.println("\nThe big bus is driving");
        int horsepower = engine.go();
        reportOnSpeed(horsepower);
    }

}
```

SmallCar is similar to BigBus but is much lighter.

```
package za.ac.cput.kabasob.designpatterns.structural.bridge;

/**
 *
 * @author boniface
 */
public class SmallCar extends Vehicle {

    public SmallCar(Engine engine) {
        this.weightInKilos = 600;
        this.engine = engine;
    }

    @Override
    public void drive() {
        System.out.println("\nThe small car is driving");
        int horsepower = engine.go();
        reportOnSpeed(horsepower);
    }

}
```

Our implementor interface is the Engine interface, which declares the go() method.

```
package za.ac.cput.kabasob.designpatterns.structural.bridge;

/**
 *
 * @author boniface
 */
public interface Engine {
    public int go();
}
```

```
package za.ac.cput.kabasob.designpatterns.structural.bridge;

/**
 *
 * @author boniface
 */
public class BigEngine implements Engine {

    int horsepower;

    public BigEngine() {
        horsepower = 350;
    }

    @Override
    public int go() {
        System.out.println("The big engine is running");
        return horsepower;
    }
}
```

```java
package za.ac.cput.kabasob.designpatterns.structural.bridge;

/**
 *
 * @author boniface
 */
public class SmallEngine implements Engine {

    int horsepower;

    public SmallEngine() {
        horsepower = 100;
    }

    @Override
    public int go() {
        System.out.println("The small engine is running");
        return horsepower;
    }
}
```

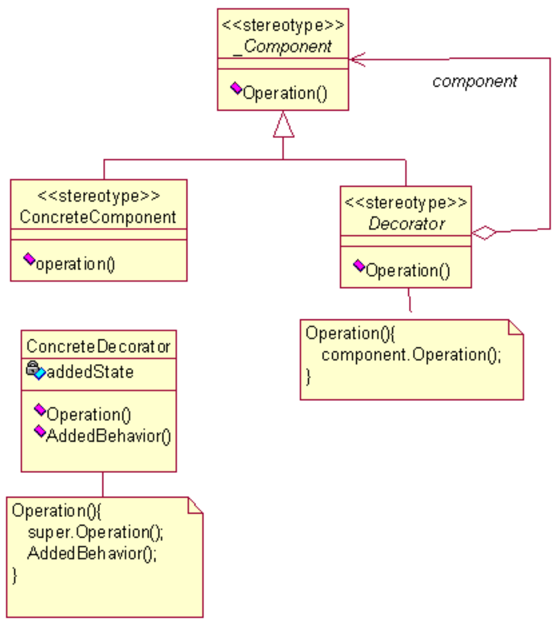
The MainDriver class demonstrates our bridge pattern. We create a BigBus vehicle with a SmallEngine implementor. We call the vehicle's drive() method. Next, we change the implementor to a BigEngine and once again call drive(). After this, we create a SmallCar vehicle with a SmallEngine implementor. We call drive(). Next, we change the engine to a BigEngine and once again call drive().

```java
package za.ac.cput.kabasob.designpatterns.structural.bridge;

/**
 *
 * @author boniface
 */
public class MainDriver {

    public static void main(String[] args) {

        Vehicle vehicle = new BigBus(new SmallEngine());
        vehicle.drive();
        vehicle.setEngine(new BigEngine());
        vehicle.drive();

        vehicle = new SmallCar(new SmallEngine());
        vehicle.drive();
        vehicle.setEngine(new BigEngine());
        vehicle.drive();
    }
}
```

Record your output in the lab book

Notice that we were able to change the implementor (engine) dynamically for each vehicle. These changes did not affect the client code in MainDriver. In addition, since BigBus and SmallCar were both subclasses of the Vehicle abstraction, we were even able to point the vehicle reference to a BigBus object and a SmallCar object and call the same drive() method for both types of vehicles.

5.7.2.7 Decorator Pattern



<<stereotype>> _Component
Operation()
component
<<stereotype>> ConcreteComponent
operation()
<<stereotype>> Decorator
Operation()
Operation(){ component.Operation(); }
ConcreteDecorator
addedState
Operation()
AddedBehavior()
Operation(){ super.Operation(); AddedBehavior(); }

The decorator pattern is a structural design pattern. Whereas inheritance adds functionality to classes, the decorator pattern adds functionality to objects by wrapping objects in other objects. Each time additional functionality is required, the object is wrapped in another object. Java I/O streams are a well-known example of the decorator pattern.

For the decorator pattern, we require a class that serves as the base object that we add functionality to. This is a Concrete Component, and it implements a Component interface. The Component interface declares the common operations that are to be performed by the concrete component and all the decorators that wrap the concrete component object. A Decorator is an abstract class that implements the Component interface and contains a reference to a Component. Concrete Decorators are classes that extend Decorator.

We can illustrate the decorator pattern with an example. We'll start by creating an Animal interface. The Animal interface is our component interface.

```
package za.ac.cput.kabasob.designpatterns.structural.decorator;

/**
 *
 * @author boniface
 */
public interface Animal {

    public void describe();

}
```

LivingAnimal implements Animal and is our concrete component. Its describe() method displays a message indicating that it is an animal.

```
package za.ac.cput.kabasob.designpatterns.structural.decorator;
```

```java
/**
 *
 * @author boniface
 */
public class LivingAnimal implements Animal {

    @Override
    public void describe() {
        System.out.println("\nI am an animal.");
    }
}
```

AnimalDecorator is our decorator abstract class. It implements Animal but since it is an abstract class, it does not have to implement describe(). Its constructor sets its Animal reference field.

```java
package za.ac.cput.kabasob.designpatterns.structural.decorator;

/**
 *
 * @author boniface
 */
public abstract class AnimalDecorator implements Animal {

    Animal animal;

    public AnimalDecorator(Animal animal) {
        this.animal = animal;
    }
}
```

LegDecorator is a concrete decorator. Its constructor passes an Animal reference to AnimalDecorator's constructor. Its describe() method call's the Animal reference's describe() method and then outputs an additional message. It then calls its dance() method, showing that additional functionality can be added by the concrete decorator.

```java
package za.ac.cput.kabasob.designpatterns.structural.decorator;

/**
 *
 * @author boniface
 */
public class LegDecorator extends AnimalDecorator {

    public LegDecorator(Animal animal) {
        super(animal);
    }

    @Override
    public void describe() {
        animal.describe();
        System.out.println("I have legs.");
        dance();
    }

    public void dance() {
        System.out.println("I can dance.");
    }
}
```

WingDecorator is a concrete decorator very similar to LegDecorator.

```java
package za.ac.cput.kabasob.designpatterns.structural.decorator;

/**
 *
 * @author boniface
 */
public class WingDecorator extends AnimalDecorator {

    public WingDecorator(Animal animal) {
        super(animal);
    }

    @Override
    public void describe() {
        animal.describe();
        System.out.println("I have wings.");
        fly();
    }

    public void fly() {
        System.out.println("I can fly.");
    }
}
```

GrowlDecorator is another concrete decorator.

```
package za.ac.cput.kabasob.designpatterns.structural.decorator;

/**
 *
 * @author boniface
 */
public class GrowlDecorator extends AnimalDecorator {

    public GrowlDecorator(Animal animal) {
        super(animal);
    }

    @Override
    public void describe() {
        animal.describe();
        growl();
    }

    public void growl() {
        System.out.println("Grrrrr.");
    }
}
```

The MainDriver class demonstrates the decorator design pattern. First, a LivingAnimal object is created and is referenced via the Animal reference. The describe() method is called. After this, we wrap the LivingAnimal in a LegDecorator object and once again call describe. We can see that legs have been added. After that, we wrap the LegDecorator in a WingDecorator, which adds wings to our animal. Later, we wrap the animal in two GrowlDecorators. From the describe() output, we can see that two growl messages are displayed since we wrapped the animal twice.

```
package za.ac.cput.kabasob.designpatterns.structural.decorator;

/**
 *
 * @author boniface
 */
public class MainDriver {

    public static void main(String[] args) {

        Animal animal = new LivingAnimal();
        animal.describe();

        animal = new LegDecorator(animal);
        animal.describe();

        animal = new WingDecorator(animal);
        animal.describe();

        animal = new GrowlDecorator(animal);
        animal = new GrowlDecorator(animal);
        animal.describe();
    }
}
```

Record your output in Lab Book

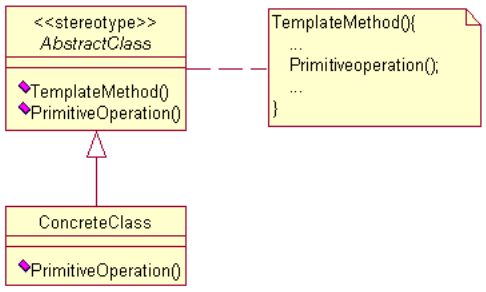
In this example, we just used the Animal reference to refer to the objects. Using this approach, we could only access the shared operations of the Animal interface (ie, the describe() method). Instead of referencing this interface, we could have referenced the concrete decorator class. This would have exposed the unique functionality of the concrete decorator.

5.7.3 Behavioral Design Patterns

In software engineering, behavioral design patterns are design patterns that identify common communication patterns between objects and realize these patterns. By doing so, these patterns increase flexibility in carrying out this communication.

Behavioral patterns are patterns that focuses on the interactions between cooperating objects. The interactions between cooperating objects should be such that they are communicating while maintaining as loose coupling as possible. The loose coupling is the key to n-tier architectures. In this, the implementation and the client should be loosely coupled in order to avoid hard-coding and dependencies.

5.7.3.1 Template Method Pattern



The template method pattern is a behavioral class pattern. A behavioral class pattern uses inheritance for distribution of behavior. In the template method pattern, a method (the 'template method') defines the steps of an algorithm. The implementation of these steps (ie, methods) can be deferred to subclasses. Thus, a particular algorithm is defined in the template method, but the exact steps of this algorithm can be defined in subclasses. The template method is implemented in an abstract class. The steps (methods) of the algorithm are declared in the abstract class, and the methods whose implementations are to be delegated to subclasses are declared abstract.

Here is an example of the template method pattern. Meal is an abstract class with a template method called doMeal() that defines the steps involved in a meal. We declare the method as final so that it can't be overridden. The algorithm defined by doMeal() consists of four steps: prepareIngredients(), cook(), eat(), and cleanUp(). The eat() method is implemented although subclasses can override the implementation. The prepareIngredients(), cook(), and cleanUp() methods are are declared abstract so that subclasses need to implement them.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.templatemethod;

/**
 *
 * @author boniface
 */
public abstract class Meal {
    // template method

    public final void doMeal() {
        prepareIngredients();
        cook();
        eat();
        cleanUp();
    }

    public abstract void prepareIngredients();
    public abstract void cook();

    public void eat() {
        System.out.println("Mmm, that's good");
    }

    public abstract void cleanUp();
}
```

The HamburgerMeal class extends Meal and implements Meal's three abstract methods.

```
package za.ac.cput.kabasob.designpatterns.behavioral.templatemethod;

/**
 *
 * @author boniface
 */
public class HamburgerMeal extends Meal {

    @Override
    public void prepareIngredients() {
        System.out.println("Getting burgers, buns, and french fries");
    }

    @Override
    public void cook() {
        System.out.println("Cooking burgers on grill and fries in oven");
    }

    @Override
    public void cleanUp() {
        System.out.println("Throwing away paper plates");
    }
}
```

The CHeeseBurgerMeal class implements Meal's three abstract methods and also overrides the eat() method.

```
package za.ac.cput.kabasob.designpatterns.behavioral.templatemethod;

/**
 *
 * @author boniface
 */
public class CheeseBurgerMeal extends Meal {
    @Override
    public void prepareIngredients() {
        System.out.println("Getting ground beef and Cheese");
    }

    @Override
    public void cook() {
        System.out.println("Cooking ground beef in pan");
    }

    @Override
    public void eat() {
        System.out.println("The Cheese Burgers are tasty");
    }

    @Override
    public void cleanUp() {
        System.out.println("Doing the dishes");
    }

}
```

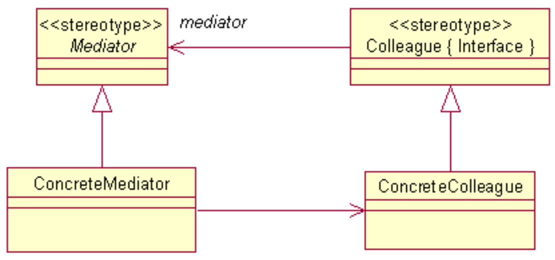
The MainDriver creates a HamburgerMeal object and calls its doMeal() method. It creates a TacoMeal object and calls doMeal() on the TacoMeal object.

```
package za.ac.cput.kabasob.designpatterns.behavioral.templatemethod;

/**
 *
 * @author boniface
 */
public class MainDriver {
    public static void main(String[] args) {
        Meal meal1 = new HamburgerMeal();
        meal1.doMeal();

        System.out.println();

        Meal meal2 = new CheeseBurgerMeal();
        meal2.doMeal();

    }
}
```

Record your output in your log Book

As you can see, the template method design pattern allows us to define the steps in an algorithm and pass the implementation of these steps to subclasses.

5.7.3.2 Mediator Pattern



The mediator pattern is a behavioral object design pattern. The mediator pattern centralizes communication between objects into a mediator object. This centralization is useful since it localizes in one place the interactions between objects, which can increase code maintainability, especially as the number of classes in an application increases. Since communication occurs with the mediator rather than directly with other objects, the mediator pattern results in a loose coupling of objects.

The classes that communicate with the mediator are known as Colleagues. The mediator implementation is known as the Concrete Mediator. The mediator can have an interface that spells out the communication with Colleages. Colleagues know their mediator, and the mediator knows its colleagues.

Now, let's look at an example of this pattern. We'll create a Mediator class (without implementing a mediator interface in this example). This mediator will mediate the communication between two buyers (a Swedish buyer and a French buyer), an American seller, and a currency converter.

The Mediator has references to the two buyers, the seller, and the converter. It has methods so that objects of these types can be registered. It also has a placeBid() method. This method takes a bid amount and a unit of currency as parameters. It converts this amount to a dollar amount via communication with the dollarConverter. It then asks the seller if the bid has been accepted, and it returns the answer.

```
package za.ac.cput.kabasob.designpatterns.behavioral.mediator;

/**
 *
 * @author boniface
 */
public class Mediator {
    Buyer swedishBuyer;
    Buyer frenchBuyer;
    AmericanSeller americanSeller;
    DollarConverter dollarConverter;

    public Mediator() {
    }

    public void registerSwedishBuyer(SwedishBuyer swedishBuyer) {
        this.swedishBuyer = swedishBuyer;
    }

    public void registerFrenchBuyer(FrenchBuyer frenchBuyer) {
        this.frenchBuyer = frenchBuyer;
    }

    public void registerAmericanSeller(AmericanSeller americanSeller) {
        this.americanSeller = americanSeller;
    }

    public void registerDollarConverter(DollarConverter dollarConverter) {
        this.dollarConverter = dollarConverter;
    }

    public boolean placeBid(float bid, String unitOfCurrency) {
        float dollarAmount = dollarConverter.convertCurrencyToDollars(bid, unitOfCurrency);
        return americanSeller.isBidAccepted(dollarAmount);
    }
}
```

Here is the Buyer class. The SwedishBuyer and FrenchBuyer classes are subclasses of Buyer. The buyer has a unit of currency as a field, and it also has a reference to the mediator. The Buyer class has a attemptToPurchase() method. This method submits a bid to the mediator's placeBid() method. It returns the mediator's response.

```
package za.ac.cput.kabasob.designpatterns.behavioral.mediator;

/**
 *
 * @author boniface
 */
class Buyer {

    Mediator mediator;
    String unitOfCurrency;

    public Buyer(Mediator mediator, String unitOfCurrency) {
        this.mediator = mediator;
        this.unitOfCurrency = unitOfCurrency;
    }

    public boolean attemptToPurchase(float bid) {
        System.out.println("Buyer attempting a bid of " + bid + " " + unitOfCurrency);
        return mediator.placeBid(bid, unitOfCurrency);
    }
}
```

The SwedishBuyer class is a subclass of Buyer. In the constructor, we set the unitOfCurrency to be ”krona”. We also register the SwedishBuyer with the mediator so that the mediator knows about the SwedishBuyer object.

```
package za.ac.cput.kabasob.designpatterns.behavioral.mediator;

/**
 *
 * @author boniface
 */
class SwedishBuyer extends Buyer {

    public SwedishBuyer(Mediator mediator) {
        super(mediator, "krona");
        this.mediator.registerSwedishBuyer(this);
    }
}
```

The FrenchBuyer class is similar to the SwedishBuyer class, except the unitOfCurrency is ”euro”, and it registers with the mediator as the FrenchBuye

```java
package za.ac.cput.kabasob.designpatterns.behavioral.mediator;

/**
 *
 * @author boniface
 */
class FrenchBuyer extends Buyer {

    public FrenchBuyer(Mediator mediator) {
        super(mediator, "euro");
        this.mediator.registerFrenchBuyer(this);
    }
}
```

In the constructor of the AmericanSeller class, the class gets a reference to the mediator and the priceInDollars gets set. This is the price of some good being sold. The seller registers with the mediator as the AmericanSeller. The seller's isBidAccepted() method takes a bid (in dollars). If the bid is over the price (in dollars), the bid is accepted and true is returned. Otherwise, false is returned.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.mediator;

/**
 *
 * @author boniface
 */
class AmericanSeller {

    Mediator mediator;
    float priceInDollars;

    public AmericanSeller(Mediator mediator, float priceInDollars) {
        this.mediator = mediator;
        this.priceInDollars = priceInDollars;
        this.mediator.registerAmericanSeller(this);
    }

    public boolean isBidAccepted(float bidInDollars) {
        if (bidInDollars >= priceInDollars) {
            System.out.println("Seller accepts the bid of " + bidInDollars + " dollars\n");
            return true;
        } else {
            System.out.println("Seller rejects the bid of " + bidInDollars + " dollars\n");
            return false;
        }
    }
}
```

The DollarConverter class is another colleague class. When created, it gets a reference to the mediator and registers itself with the mediator as the DollarConverter. This class has methods to convert amounts in euros and kronor to dollars.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.mediator;

/**
 *
 * @author boniface
 */
class DollarConverter {

    Mediator mediator;
    public static final float DOLLAR_UNIT = 1.0f;
    public static final float EURO_UNIT = 0.7f;
    public static final float KRONA_UNIT = 8.0f;

    public DollarConverter(Mediator mediator) {
        this.mediator = mediator;
        mediator.registerDollarConverter(this);
    }

    private float convertEurosToDollars(float euros) {
        float dollars = euros * (DOLLAR_UNIT / EURO_UNIT);
        System.out.println("Converting " + euros + " euros to " + dollars + " dollars");
        return dollars;
    }

    private float convertKronorToDollars(float kronor) {
        float dollars = kronor * (DOLLAR_UNIT / KRONA_UNIT);
        System.out.println("Converting " + kronor + " kronor to " + dollars + " dollars");
        return dollars;
    }

    public float convertCurrencyToDollars(float amount, String unitOfCurrency) {
        if ("krona".equalsIgnoreCase(unitOfCurrency)) {
            return convertKronorToDollars(amount);
        } else {
            return convertEurosToDollars(amount);
        }
    }
}
```

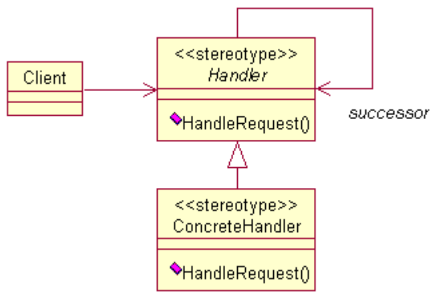
The MainDriver demonstrates our mediator pattern. It creates a SwedishBuyer object and a FrenchBuyer object. It creates an AmericanSeller object with a selling price set to 10 dollars. It then creates a DollarConverter. All of these objects register themselves with the mediator in their constructors. The Swedish buyer starts with a bid of 55 kronor and keeps bidding up in increments of 15 kronor until the bid is accepted. The French buyer starts bidding at 3 euros and keeps bidding in increments of 1.50 euros until the bid is accepted.

```
package za.ac.cput.kabasob.designpatterns.behavioral.mediator;

/**
 *
 * @author boniface
 */
public class MainDriver {
    public static void main(String[] args) {
        Mediator mediator = new Mediator();

        Buyer swedishBuyer = new SwedishBuyer(mediator);
        Buyer frenchBuyer = new FrenchBuyer(mediator);
        float sellingPriceInDollars = 10.0f;
        AmericanSeller americanSeller = new AmericanSeller(mediator, sellingPriceInDollars);
        DollarConverter dollarConverter = new DollarConverter(mediator);

        float swedishBidInKronor = 55.0f;
        while (!swedishBuyer.attemptToPurchase(swedishBidInKronor)) {
            swedishBidInKronor += 15.0f;
        }

        float frenchBidInEuros = 3.0f;
        while (!frenchBuyer.attemptToPurchase(frenchBidInEuros)) {
            frenchBidInEuros += 1.5f;
        }
    }
}
```

Record your output in the log Book

In this example of the mediator pattern, notice that all communication between our objects (buyers, seller, and converter) occurs via the mediator. The mediator pattern helps reduce the number of object references needed (via composition) as classes proliferate in a project as a project grows.

5.7.3.3 Chain of Responsibility Pattern



The chain of responsibility pattern is a behavioral object design pattern. In the chain of responsibility pattern, a series of handler objects are chained together to handle a request made by a client object. If the first handler can't handle the request, the request is forwarded to the next handler, and it is passed down the chain until the request reaches a handler that can handle the request or the chain ends. In this pattern, the client is decoupled from the actual handling of the request, since it does not know what class will actually handle the request.

In this pattern, a Handler is an interface for handling a request and accessing a handler's successor. A Handler is implemented by a Concrete Handler. The Concrete Handler will handle the request or pass it on to the next Concrete Handler. A Client makes the request to the start of the handler chain.

Now, let's look at an example of the chain of responsibility pattern. Rather than an interface, We'll use an abstract base class as the handler so that subclasses can utilize the implemented setSuccessor() method. This abstract class is called PlanetHandler. Concrete handlers that subclass PlanetHandler need to implement the handleRequest() method.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.chainofresponsibility;

/**
 *
 * @author boniface
 */
public abstract class PlanetHandler {

    PlanetHandler successor;

    public void setSuccessor(PlanetHandler successor) {
        this.successor = successor;
    }

    public abstract void handleRequest(PlanetEnum request);
}
```

This example will utilize an enum of the planets called PlanetEnum.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.chainofresponsibility;

/**
 *
 * @author boniface
 */
public enum PlanetEnum {

    MERCURY, VENUS, EARTH, MARS, JUPITER, SATURN, URANUS, NEPTUNE;
}
```

MercuryHandler subclasses PlanetHandler and implements the handleRequest() method. If the request is a PlanetEnum.MERCURY, it will handle the request. Otherwise, the request is passed on to this handler's successor if the successor exists.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.chainofresponsibility;

/**
 *
 * @author boniface
 */
public class MercuryHandler extends PlanetHandler {

    public void handleRequest(PlanetEnum request) {
        if (request == PlanetEnum.MERCURY) {
            System.out.println("MercuryHandler handles " + request);
            System.out.println("Mercury is hot.\n");
        } else {
            System.out.println("MercuryHandler doesn't handle " + request);
            if (successor != null) {
                successor.handleRequest(request);
            }
        }
    }
}
```

VenusHandler is similar to MercuryHandler, except it handles PlanetEnum.VENUS requests.

```
package za.ac.cput.kabasob.designpatterns.behavioral.chainofresponsibility;

/**
 *
 * @author boniface
 */
public class VenusHandler extends PlanetHandler {

    public void handleRequest(PlanetEnum request) {
        if (request == PlanetEnum.VENUS) {
            System.out.println("VenusHandler handles " + request);
            System.out.println("Venus is poisonous.\n");
        } else {
            System.out.println("VenusHandler doesn't handle " + request);
            if (successor != null) {
                successor.handleRequest(request);
            }
        }
    }
}
```

EarthHandler similarly handles PlanetEnum.EARTH requests.

```
package za.ac.cput.kabasob.designpatterns.behavioral.chainofresponsibility;

/**
 *
 * @author boniface
 */
public class EarthHandler extends PlanetHandler {

    public void handleRequest(PlanetEnum request) {
        if (request == PlanetEnum.EARTH) {
            System.out.println("EarthHandler handles " + request);
            System.out.println("Earth is comfortable.\n");
        } else {
            System.out.println("EarthHandler doesn't handle " + request);
            if (successor != null) {
                successor.handleRequest(request);
            }
        }
    }
}
```

The MainDriver is the client class. It creates the chain of handlers, starting with MercuryHandler, then VenusHandler, and then EarthHandler. The setUpChain() method returns the chain to main() via a PlanetHandler reference. Four requests are made of the chain, where the requests are VENUS, MERCURY, EARTH, and JUPITER.

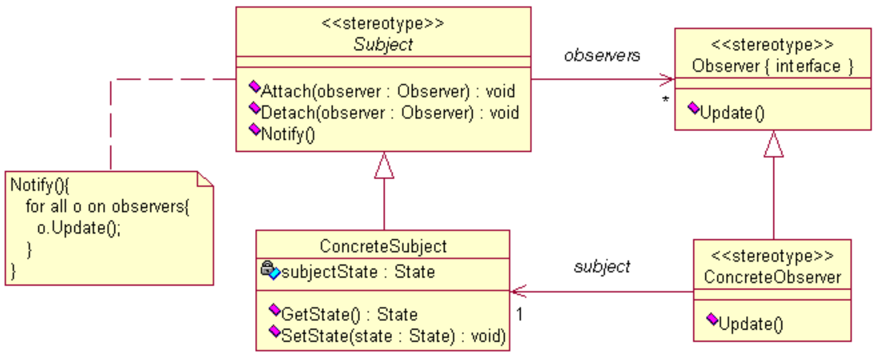
```
package za.ac.cput.kabasob.designpatterns.behavioral.chainofresponsibility;

/**
 *
 * @author boniface
 */
public class MainDriver {

    public static void main(String[] args) {
        PlanetHandler chain = setUpChain();
        //Submit Requests
        chain.handleRequest(PlanetEnum.VENUS);
        chain.handleRequest(PlanetEnum.MERCURY);
        chain.handleRequest(PlanetEnum.EARTH);
        chain.handleRequest(PlanetEnum.JUPITER);
    }

    public static PlanetHandler setUpChain() {
        PlanetHandler mercuryHandler = new MercuryHandler();
        PlanetHandler venusHandler = new VenusHandler();
        PlanetHandler earthHandler = new EarthHandler();
        mercuryHandler.setSuccessor(venusHandler);
        venusHandler.setSuccessor(earthHandler);


        return mercuryHandler;
    }
}
```

Record the output in your lab book

Notice that if a handler can't handle the request, it passes the request on to the next handler.

Notice that the last request made of the chain is JUPITER. This request is not handled by any handlers, demonstrating that a request does not have to be handled by any handlers. If we had wanted to, we could have also written an OtherPlanets handler and placed it at the end of the chain to handle planet requests not handled by other previous handlers. This would demonstrate that we can make our handlers specific at the beginning of the chain and more general at the end of the chain, thus handling broader categories of requests as we approach the end of the chain.

5.7.3.4 Observer Pattern



The observer pattern is a behavioral object design pattern. In the observer pattern, an object called the subject maintains a collection of objects called observers. When the subject changes, it notifies the observers. Observers can be added or removed from the collection of observers in the subject. The changes in state of the subject can be passed to the observers so that the observers can change their own state to reflect this change.

The subject has an interface that defines methods for attaching and detaching observers from the subject's collection of observers. This interface also features a notification method. This method should be called when the state of the subject changes. This notifies the observers that the subject's state has changed. The observers have an interface with a method to update the observer. This update method is called for each observer in the subject's notification method. Since this communication occurs via an interface, any concrete observer implementing the observer interface can be updated by the subject. This results in loose coupling between the subject and the observer classes.

Now we'll look at an example of the observer pattern. We'll start by creating an interface for the subject called WeatherSubject. This will declare three methods: addObserver(), removeObserver(), and doNotify().

```java
package za.ac.cput.kabasob.designpatterns.behavioral.observer;

/**
 *
 * @author boniface
 */
public interface WeatherSubject {

    public void addObserver(WeatherObserver weatherObserver);

    public void removeObserver(WeatherObserver weatherObserver);

    public void doNotify();
}
```

We'll also create an interface for the observers called WeatherObserver. It features one method, a doUpdate() method.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.observer;

/**
 *
 * @author boniface
 */
public interface WeatherObserver {

    public void doUpdate(int temperature);
}
```

The WeatherStation class implements WeatherSubject. It is our subject class. It maintains a set of WeatherObservers which are added via addObserver() and removed via removeObserver(). When WeatherSubject's state changes via setTemperature(), the doNotify() method is called, which contacts all the WeatherObservers with the temperature via their doUpdate() methods.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.observer;

import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

/**
 *
 * @author boniface
 */
public class WeatherStation implements WeatherSubject {

    Set<WeatherObserver> weatherObservers;
    int temperature;

    public WeatherStation(int temperature) {
        weatherObservers = new HashSet<WeatherObserver>();
        this.temperature = temperature;
    }

    @Override
    public void addObserver(WeatherObserver weatherObserver) {
        weatherObservers.add(weatherObserver);
    }

    @Override
    public void removeObserver(WeatherObserver weatherObserver) {
        weatherObservers.remove(weatherObserver);
    }

    @Override
    public void doNotify() {
        Iterator<WeatherObserver> it = weatherObservers.iterator();
        while (it.hasNext()) {
            WeatherObserver weatherObserver = it.next();
            weatherObserver.doUpdate(temperature);
        }
    }

    public void setTemperature(int newTemperature) {
        System.out.println("\nWeather station setting temperature to " + newTemperature);
        temperature = newTemperature;
        doNotify();
    }
}
```

WeatherCustomer1 is an observer that implements WeatherObserver. Its doUpdate() method gets the current temperature from the WeatherStation and displays it.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.observer;

/**
 *
 * @author boniface
 */
public class WeatherCustomer1 implements WeatherObserver {

    @Override
    public void doUpdate(int temperature) {
        System.out.println("Weather customer 1 just found out the temperature is:" + ↩
     temperature);
    }
}
```

WeatherCustomer2 performs similar functionality as WeatherCustomer1.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.observer;

/**
 *
 * @author boniface
 */
public class WeatherCustomer2 implements WeatherObserver {

    @Override
    public void doUpdate(int temperature) {
        System.out.println("Weather customer 2 just found out the temperature is:" + ↩
     temperature);
    }
}
```

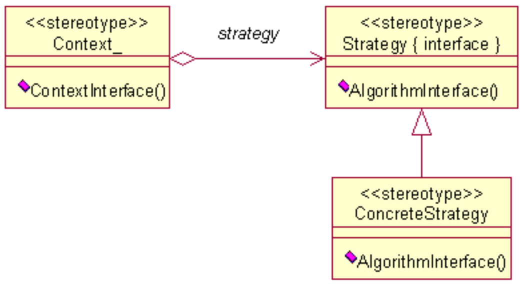
The MainDriver class demonstrates the observer pattern. It creates a WeatherStation and then a WeatherCustomer1 and a WeatherCustomer2. The two customers are added as observers to the weather station. Then the setTemperature() method of the weather station is called. This changes the state of the weather station and the customers are notified of this temperature update. Next, the WeatherCustomer1 object is removed from the station's collection of observers. Then, the setTemperature() method is called again. This results in the notification of the WeatherCustomer2 object.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.observer;

/**
 *
 * @author boniface
 */
public class MainDriver {

    public static void main(String[] args) {
        WeatherStation weatherStation = new WeatherStation(33);

        WeatherCustomer1 wc1 = new WeatherCustomer1();
        WeatherCustomer2 wc2 = new WeatherCustomer2();
        weatherStation.addObserver(wc1);
        weatherStation.addObserver(wc2);

        weatherStation.setTemperature(34);

        weatherStation.removeObserver(wc1);

        weatherStation.setTemperature(35);
    }
}
```

Record output in your log book

In a more advanced case, we might have given each observer a reference to the weather station object. This could allow the observer the ability to compare the state of the subject in detail with its own state and make any necessary updates to its own state.

5.7.3.5 Strategy Pattern



The strategy pattern is a behavioral object design pattern. In the strategy pattern, different algorithms are represented as Concrete Strategy classes, and they share a common Strategy interface. A Context object contains a reference to a Strategy. By changing the Context's Strategy, different behaviors can be obtained. Although these behaviors are different, the different strategies all operate on data from the Context.

The strategy pattern is one way that composition can be used as an alternative to subclassing. Rather than providing different behaviors via subclasses overriding methods in superclasses, the strategy pattern allows different behaviors to be placed in Concrete Strategy classes which share the common Strategy interface. A Context class is composed of a reference to a Strategy.

Here is an example of the strategy pattern. First, we'll define a Strategy interface. It declares a checkTemperature() method.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.strategy;

/**
 *
 * @author boniface
 */
public interface Strategy {
    boolean checkTemperature(int temperatureInF);

}
```

The HikeStrategy class is a concrete strategy class that implements the Strategy interface. The checkTemperature method is implemented so that if the temperature is between 50 and 90, it returns true. Otherwise it returns false.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.strategy;

/**
 *
 * @author boniface
 */
public class HikeStrategy implements Strategy {

    @Override
    public boolean checkTemperature(int temperatureInF) {
        if ((temperatureInF >= 50) && (temperatureInF <= 90)) {
            return true;
        } else {
            return false;
        }
    }
}
```

The SkiStrategy implements the Strategy interface. If the temperature is 32 or less, the checkTemperature method returns true. Otherwise it returns false.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.strategy;

/**
 *
 * @author boniface
 */
public class SkiStrategy implements Strategy {

    @Override
    public boolean checkTemperature(int temperatureInF) {
        if (temperatureInF <= 32) {
            return true;
        } else {
            return false;
        }
    }
}
```

The Context class contains a temperature and a reference to a Strategy. The Strategy can be changed, resulting in different behavior that operates on the same data in the Context. The result of this can be obtained from the Context via the getResult() method.

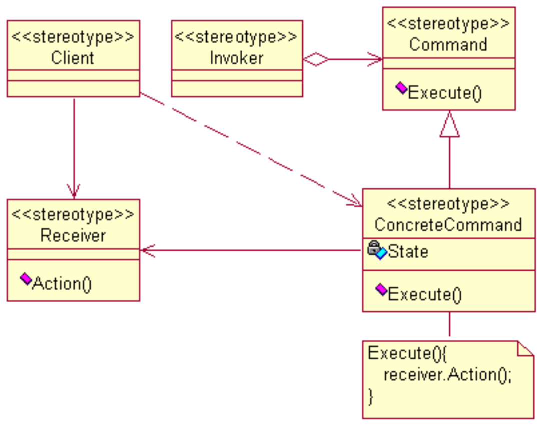
```java
package za.ac.cput.kabasob.designpatterns.behavioral.strategy;

/**
 *
 * @author boniface
 */
public class Context {

    int temperatureInF;
    Strategy strategy;

    public Context(int temperatureInF, Strategy strategy) {
        this.temperatureInF = temperatureInF;
        this.strategy = strategy;
    }

    public void setStrategy(Strategy strategy) {
        this.strategy = strategy;
    }

    public int getTemperatureInF() {
        return temperatureInF;
    }

    public boolean getResult() {
        return strategy.checkTemperature(temperatureInF);
    }
}
```

The MainDriver class creates a Context object with a temperature of 60 and with a SkiStrategy. It displays the temperature from the context and whether that temperature is OK for skiing. After that, it sets the Strategy in the Context to HikeStrategy. It then displays the temperature from the context and whether that temperature is OK for hiking.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.strategy;

/**
 *
 * @author boniface
 */
public class MainDriver {

    public static void main(String[] args) {

        int temperatureInF = 60;

        Strategy skiStrategy = new SkiStrategy();
        Context context = new Context(temperatureInF, skiStrategy);

        System.out.println("Is the temperature (" + context.getTemperatureInF() + "F) good ↩
    for skiing? " + context.getResult());

        Strategy hikeStrategy = new HikeStrategy();
        context.setStrategy(hikeStrategy);

        System.out.println("Is the temperature (" + context.getTemperatureInF() + "F) good ↩
    for hiking? " + context.getResult());
    }
}
```

Record output in your log book

5.7.3.6 Command Pattern



The command pattern is a behavioral object design pattern. In the command pattern, a Command interface declares a method for executing a particular action. Concrete Command classes implement the execute() method of the Command interface, and this execute() method invokes the appropriate action method of a Receiver class that the Concrete Command class contains. The Receiver class performs a particular action. A Client class is responsible for creating a Concrete Command and setting the Receiver of the Concrete Command. An Invoker class contains a reference to a Command and has a method to execute the Command.

In the command pattern, the invoker is decoupled from the action performed by the receiver. The invoker has no knowledge of the receiver. The invoker invokes a command, and the command executes the appropriate action of the receiver. Thus, the invoker can invoke commands without knowing the details of the action to be performed. In addition, this decoupling means that changes to the receiver's action don't directly affect the invocation of the action.

The command pattern can be used to perform 'undo' functionality. In this case, the Command interface should include an unexecute() method.

Here is an example of the command pattern. We have a Command interface with an execute() method.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.command;

/**
 *
 * @author boniface
 */
public interface Command {

    public void execute();
}
```

LunchCommand implements Command. It contains a reference to Lunch, a receiver. Its execute() method invokes the appropriate action on the receiver.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.command;

/**
 *
 * @author boniface
 */
public class LunchCommand implements Command {

    Lunch lunch;

    public LunchCommand(Lunch lunch) {
        this.lunch = lunch;
    }

    @Override
    public void execute() {
        lunch.makeLunch();
    }

}
```

The DinnerCommand is similar to LunchCommand. It contains a reference to Dinner, a receiver. Its execute() method invokes the makeDinner() action of the Dinner object.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.command;

/**
 *
 * @author boniface
 */
public class DinnerCommand implements Command {

    Dinner dinner;

    public DinnerCommand(Dinner dinner) {
        this.dinner = dinner;
    }

    @Override
    public void execute() {
        dinner.makeDinner();
    }
}
```

Lunch is a receiver.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.command;

/**
 *
 * @author boniface
 */
public class Lunch {

    public void makeLunch() {
        System.out.println("Lunch is being made");
    }
}
```

Dinner is also a receiver.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.command;

/**
 *
 * @author boniface
 */
public class Dinner {

    public void makeDinner() {
        System.out.println("Dinner is being made");
    }
}
```

MealInvoker is the invoker class. It contains a reference to the Command to invoke. Its invoke() method calls the execute() method of the Command.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.command;

/**
 *
 * @author boniface
 */
public class MealInvoker {

    Command command;

    public MealInvoker(Command command) {
        this.command = command;
    }

    public void setCommand(Command command) {
        this.command = command;
    }

    public void invoke() {
        command.execute();
    }
}
```

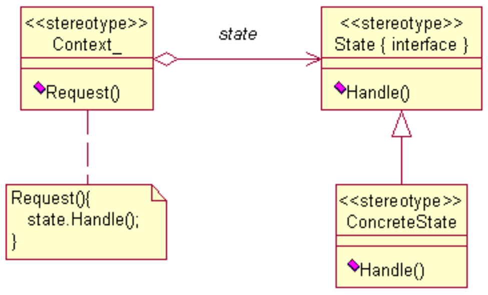
The MainDriver class demonstrates the command pattern. It instantiates a Lunch (receiver) object and creates a LunchCommand (concrete command) with the Lunch. The LunchCommand is referenced by a Command interface reference. Next, we perform the same procedure on the Dinner and DinnerCommand objects. After this, we create a MealInvoker object with lunchCommand, and we call the invoke() method of mealInvoker. After this, we set mealInvoker's command to dinnerCommand, and once again call invoke() on mealInvoker.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.command;

/**
 *
 * @author boniface
 */
public class MainDriver {
    public static void main(String[] args) {

        Lunch lunch = new Lunch(); // receiver
        Command lunchCommand = new LunchCommand(lunch); // concrete command

        Dinner dinner = new Dinner(); // receiver
        Command dinnerCommand = new DinnerCommand(dinner); // concrete command

        MealInvoker mealInvoker = new MealInvoker(lunchCommand); // invoker
        mealInvoker.invoke();

        mealInvoker.setCommand(dinnerCommand);
        mealInvoker.invoke();

    }


}
```

Record output in your log book

As you can see, the invoker invokes a command, but has no direct knowledge of the action being performed by the receiver.

5.7.3.7 State Pattern



The state pattern is a behavioral object design pattern. The idea behind the state pattern is for an object to change its behavior depending on its state. In the state pattern, we have a Context class, and this class has a State reference to a Concrete State instance. The State interface declares particular methods that represent the behaviors of a particular state. Concrete States implement these behaviors. By changing a Context's Concrete State, we change its behavior. In essence, in the state pattern, a class (the Context) is supposed to behave like different classes depending on its state. The state pattern avoids the use of switch and if statements to change behavior.

Let's look at an example of the state pattern. First off, we'll define the EmotionalState interface. It declares two methods, sayHello() and sayGoodbye().

```java
package za.ac.cput.kabasob.designpatterns.behavioral.state;

/**
 *
 * @author boniface
 */
public interface EmotionalState {

    public String sayHello();

    public String sayGoodbye();
}
```

The HappyState class is a Concrete State that implements sayHello() and sayGoodbye() of EmotionalState. These messages are cheerful (representing a happy state).

```java
package za.ac.cput.kabasob.designpatterns.behavioral.state;

/**
 *
 * @author boniface
 */
// Concrete State
public class HappyState implements EmotionalState {

    @Override
    public String sayGoodbye() {
        return "Bye, friend!";
    }

    @Override
    public String sayHello() {
        return "Hello, friend!";
    }

}
```

The SadState class also implements the EmotionalState interface. The messages are sad (representing a sad state).

```
package za.ac.cput.kabasob.designpatterns.behavioral.state;

/**
 *
 * @author boniface
 */
//Concrete State
public class SadState implements EmotionalState {

    @Override
    public String sayGoodbye() {
        return "Bye. Sniff, sniff.";
    }

    @Override
    public String sayHello() {
        return "Hello. Sniff, sniff.";
    }
}
```

The Person class is the Context class. It contains an EmotionalState reference to a concrete state. In this example, we have Person implement the EmotionalState reference, and we pass the calls to Person's sayHello() and sayGoodbye() methods on to the corresponding methods on the emotionalState reference. As a result of this, a Person object behaves differently depending on the state of Person (ie, the current EmotionalState reference).

```
package za.ac.cput.kabasob.designpatterns.behavioral.state;

/**
 *
 * @author boniface
 */
// Context
public class Person implements EmotionalState {

    EmotionalState emotionalState;

    public Person(EmotionalState emotionalState) {
        this.emotionalState = emotionalState;
    }

    public void setEmotionalState(EmotionalState emotionalState) {
        this.emotionalState = emotionalState;
    }

    @Override
    public String sayGoodbye() {
        return emotionalState.sayGoodbye();
    }

    @Override
    public String sayHello() {
        return emotionalState.sayHello();
    }
}
```

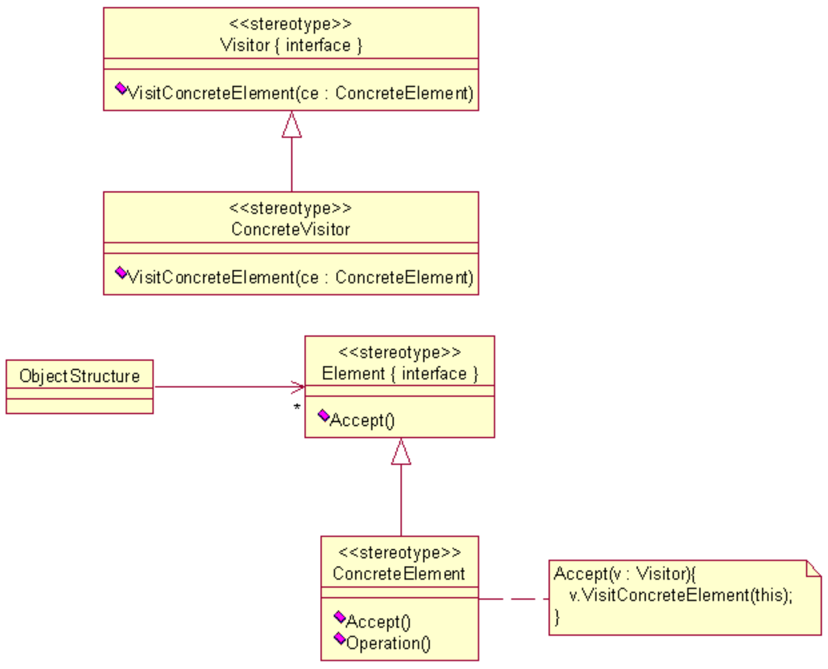
The MainDriver class demonstrates the state pattern. First, it creates a Person object with a HappyState object. We display the results of sayHello() and sayGoodbyte() when the person object is in the happy state. Next, we change the person object's state with a SadState object. We display the results of sayHello() and sayGoodbyte(), and we see that in the sad state, the person object's behavior is different.

```
package za.ac.cput.kabasob.designpatterns.behavioral.state;

/**
 *
 * @author boniface
 */
public class MainDriver {
    public static void main(String[] args) {

        Person person = new Person(new HappyState());
        System.out.println("Hello in happy state: " + person.sayHello());
        System.out.println("Goodbye in happy state: " + person.sayGoodbye());

        person.setEmotionalState(new SadState());
        System.out.println("Hello in sad state: " + person.sayHello());
        System.out.println("Goodbye in sad state: " + person.sayGoodbye());
    }
}
```

Record output in your log book

Note that we don't necessarily need to have the Context (ie, Person) implement the EmotionalState interface. The behavioral changes could have been internal to the Context rather than exposing EmotionalState's methods to the outside. However, having the Context class implement the State interface allows us to directly access the different behaviors that result from the different states of the Context.

5.7.3.8 Visitor Pattern



The visitor pattern is a behavioral object design pattern. The visitor pattern is used to simplify operations on groupings of related objects. These operations are performed by the visitor rather than by placing this code in the classes being visited. Since the operations are performed by the visitor rather than by the classes being visited, the operation code gets centralized in the visitor rather than being spread out across the grouping of objects, thus leading to code maintainability. The visitor pattern also avoids the use of the instanceof operator in order to perform calculations on similar classes.

In the visitor pattern, we have a Visitor interface that declares visit() methods for the various types of elements that can be visited. Concrete Visitors implement the Visitor interface's visit() methods. The visit() methods are the operations that should be performed by the visitor on an element being visited.

The related classes that will be visited implement an Element interface that declares an accept() method that takes a visitor as an argument. Concrete Elements implement the Element interface and implement the accept() method. In the accept() method, the visitor's visit() method is called with 'this', the current object of the Concrete Element type.

The elements to visit all implement the accept() method that takes a visitor as an argument. In this method, they call the visitor's visit() method with 'this'. As a result of this, an element takes a visitor and then the visitor performs its operation on the element.

Let's illustrate the visitor pattern with an example. First, we'll define a NumberVisitor interface. This interface declares three visit methods which take different types as arguments. Note that if we only wrote one visit method, we'd have to use the instanceof operator or a similar technique to handle the different element types. However, since we have separate visit methods, we don't need the instanceof operator, since each visit method handles a different type.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.visitor;

import java.util.List;

/**
 *
 * @author boniface
 */
public interface NumberVisitor {

    public void visit(TwoElement twoElement);

    public void visit(ThreeElement threeElement);

    public void visit(List<NumberElement> elementList);
}
```

All of the elements classes to be visited will implement the NumberElement interface. This interface has a single method that takes a NumberVisitor as an argument

```java
package za.ac.cput.kabasob.designpatterns.behavioral.visitor;

/**
 *
 * @author boniface
 */
public interface NumberElement {

    public void accept(NumberVisitor visitor);
}
```

Let's create a TwoElement class that implements NumberElement. It has two int fields. Its accept() method calls the visitor's visit() method with 'this'. The operator to be performed on TwoElement is performed by the visitor.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.visitor;

/**
 *
 * @author boniface
 */
public class TwoElement implements NumberElement{

    int a;
    int b;

    public TwoElement(int a, int b) {
        this.a = a;
        this.b = b;
    }

    @Override
    public void accept(NumberVisitor visitor) {
        visitor.visit(this);
    }
}
```

The ThreeElement class is similar to TwoElement, except that it has three int fields.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.visitor;

/**
 *
 * @author boniface
 */
public class ThreeElement implements NumberElement {

    int a;
    int b;
    int c;

    public ThreeElement(int a, int b, int c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }

    @Override
    public void accept(NumberVisitor visitor) {
        visitor.visit(this);
    }
}
```

Now, let's create a visitor called SumVisitor that implements the NumberVisitor interface. For TwoElement and ThreeElement objects, this visitor will sum up the int fields. For a List of NumElements (ie, TwoElement and ThreeElement objects), this visitor will iterate over the elements and call their accept() methods. As a result of this, the visitor will perform visit operations on all the TwoElement and ThreeElement objects that make up the list, since the call to accept() in turn calls the visitor's visit methods for the TwoElement and ThreeElement objects.

```java
 package za.ac.cput.kabasob.designpatterns.behavioral.visitor;

import java.util.List;

/**
 *
 * @author boniface
 */
public class SumVisitor implements NumberVisitor {

    @Override
    public void visit(TwoElement twoElement) {
        int sum = twoElement.a + twoElement.b;
        System.out.println(twoElement.a + "+" + twoElement.b + "=" + sum);
    }

    @Override
    public void visit(ThreeElement threeElement) {
        int sum = threeElement.a + threeElement.b + threeElement.c;
        System.out.println(threeElement.a + "+" + threeElement.b + "+" + threeElement.c + "=↩
    " + sum);
    }

    @Override
    public void visit(List<NumberElement> elementList) {
        for (NumberElement ne : elementList) {
            ne.accept(this);
        }
    }
}
```

Here is another visitor, TotalSumVisitor. In addition to summing up the int fields and displaying the sum, this visitor will keep track of the total sums of all the elements that are visited.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.visitor;

import java.util.List;

/**
 *
 * @author boniface
 */
public class TotalSumVisitor implements NumberVisitor {

    int totalSum = 0;

    @Override
    public void visit(TwoElement twoElement) {
        int sum = twoElement.a + twoElement.b;
        System.out.println("Adding " + twoElement.a + "+" + twoElement.b + "=" + sum + " to ↩
 total");
        totalSum += sum;
    }

    @Override
    public void visit(ThreeElement threeElement) {
        int sum = threeElement.a + threeElement.b + threeElement.c;
        System.out.println("Adding " + threeElement.a + "+" + threeElement.b + "+" + ↩
 threeElement.c + "=" + sum + " to total");
        totalSum += sum;
    }

    @Override
    public void visit(List<NumberElement> elementList) {
        for (NumberElement ne : elementList) {
            ne.accept(this);
        }
    }

    public int getTotalSum() {
        return totalSum;
    }

}
```

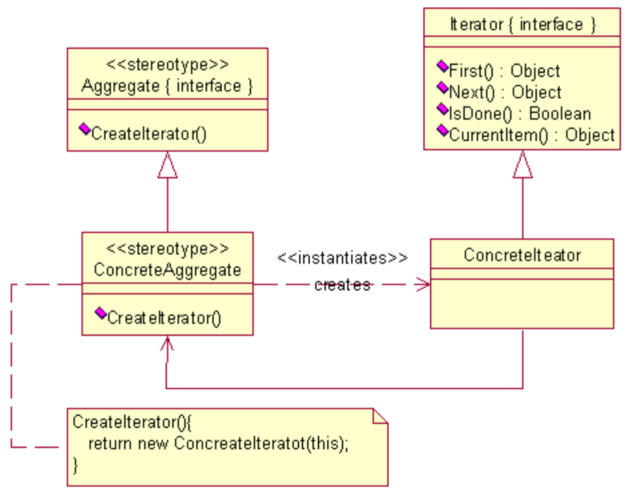
Let's see the visitor pattern in action. The MainDriver class creates two TwoElement objects and one ThreeElement object. It creates a list of NumberElements and adds the TwoElement object and the ThreeElement object to the list. Next, we create a SumVisitor and we visit the list with the SumVisitor. After this, we create a TotalSumVisitor and visit the list with the TotalSumVisitor. We display the total sum via the call to TotalSumVisitor's getTotalSum() method.

```
package za.ac.cput.kabasob.designpatterns.behavioral.visitor;

import java.util.ArrayList;
import java.util.List;

/**
 *
 * @author boniface
 */
public class MainDriver {
    public static void main(String[] args) {

        TwoElement two1 = new TwoElement(3, 3);
        TwoElement two2 = new TwoElement(2, 7);
        ThreeElement three1 = new ThreeElement(3, 4, 5);

        List<NumberElement> numberElements = new ArrayList<NumberElement>();
        numberElements.add(two1);
        numberElements.add(two2);
        numberElements.add(three1);

        System.out.println("Visiting element list with SumVisitor");
        NumberVisitor sumVisitor = new SumVisitor();
        sumVisitor.visit(numberElements);

        System.out.println("\nVisiting element list with TotalSumVisitor");
        TotalSumVisitor totalSumVisitor = new TotalSumVisitor();
        totalSumVisitor.visit(numberElements);
        System.out.println("Total sum:" + totalSumVisitor.getTotalSum());
    }
}
```

Record output in your log book

Notice that if we'd like to perform new operations on the grouping of elements, all we would need to do is write a new visitor class. We would not have to make any additions to the existing element classes, since they provide the data but none of the code for the operations.

5.7.3.9 Iterator Pattern



The iterator pattern is a behavioral object design pattern. The iterator pattern allows for the traversal through the elements in a grouping of objects via a standardized interface. An Iterator interface defines the actions that can be performed. These actions include being able to traverse the objects and also obtain the objects.

Java features the widely used java.util.Iterator interface which is used to iterate through things such as Java collections. We can write our own iterator by implementing java.util.Iterator. This interface features the hasNext(), next(), and remove() methods. When writing an iterator for a class, it is very common for the iterator class to be an inner class of the class that we'd like to iterate through.

Let's look at an example of this. We have an Item class, which represents an item on a menu. An item has a name and a price.

```
package za.ac.cput.kabasob.designpatterns.behavioral.iterator;

/**
 *
 * @author boniface
 */
public class Item {

    String name;
    float price;

    public Item(String name, float price) {
        this.name = name;
        this.price = price;
    }

    public String toString() {
        return name + ": \$" + price;
    }
}
```

Here is the Menu class. It has a list of menu items of type Item. Items can be added via the addItem() method. The iterator() method returns an iterator of menu items. The MenuIterator class is an inner class of Menu that implements the Iterator interface for Item objects. It contains basic implementations of the hasNext(), next(), and remove() methods.

```
package za.ac.cput.kabasob.designpatterns.behavioral.iterator;
```

```java
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

/**
 *
 * @author boniface
 */
public class Menu {

    List<Item> menuItems;

    public Menu() {
        menuItems = new ArrayList<Item>();
    }

    public void addItem(Item item) {
        menuItems.add(item);
    }

    public Iterator<Item> iterator() {
        return new MenuIterator();
    }

    class MenuIterator implements Iterator<Item> {
        int currentIndex = 0;

        @Override
        public boolean hasNext() {
            if (currentIndex >= menuItems.size()) {
                return false;
            } else {
                return true;
            }
        }

        @Override
        public Item next() {
            return menuItems.get(currentIndex++);
        }

        @Override
        public void remove() {
            menuItems.remove(--currentIndex);
        }

    }

}
```

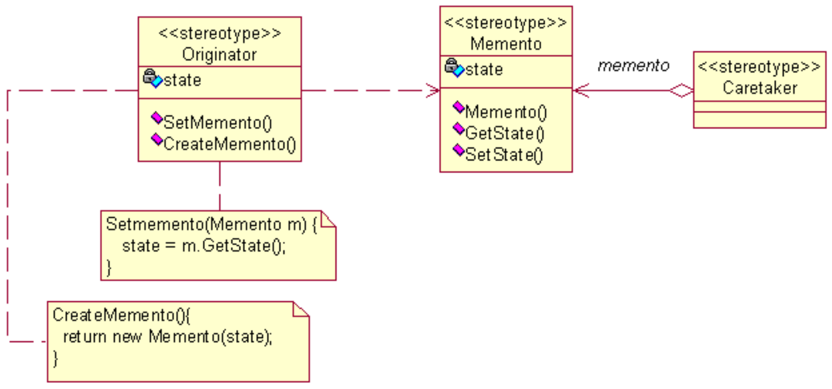
The MainDriver class demonstrates the iterator pattern. It creates three items and adds them to the menu object. Next, it gets an Item iterator from the menu object and iterates over the items in the menu. After this, it calls remove() to remove the last item obtained by the iterator. Following this, it gets a new iterator object from the menu and once again iterates over the menu items.

```java
package za.ac.cput.kabasob.designpatterns.behavioral.iterator;

import java.util.Iterator;

/**
 *
 * @author boniface
 */
public class MainDriver {

    public static void main(String[] args) {

        Item i1 = new Item("spaghetti", 7.50f);
        Item i2 = new Item("hamburger", 6.00f);
        Item i3 = new Item("chicken sandwich", 6.50f);

        Menu menu = new Menu();
        menu.addItem(i1);
        menu.addItem(i2);
        menu.addItem(i3);

        System.out.println("Displaying Menu:");
        Iterator<Item> iterator = menu.iterator();
        while (iterator.hasNext()) {
            Item item = iterator.next();
            System.out.println(item);
        }

        System.out.println("\nRemoving last item returned");
        iterator.remove();

        System.out.println("\nDisplaying Menu:");
        iterator = menu.iterator();
        while (iterator.hasNext()) {
            Item item = iterator.next();
            System.out.println(item);
        }

    }
}
```

Record output in your log book

Note that since the menu utilizes a Java collection, we could have used an iterator obtained for the menu list rather than write our own iterator as an inner class.

5.7.3.10 Memento Pattern



The memento pattern is a behavioral design pattern. The memento pattern is used to store an object's state so that this state can be restored at a later point. The saved state data in the memento object is not accessible outside of the object to be saved and restored. This protects the integrity of the saved state data.

In this pattern, an Originator class represents the object whose state we would like to save. A Memento class represents an object to store the state of the Originator. The Memento class is typically a private inner class of the Originator. As a result, the Originator has access to the fields of the memento, but outside classes do not have access to these fields. This means that state information can be transferred between the Memento and the Originator within the Originator class, but outside classes do not have access to the state data stored in the Memento.

The memento pattern also utilizes a Caretaker class. This is the object that is responsible for storing and restoring the Originator's state via a Memento object. Since the Memento is a private inner class, the Memento class type is not visible to the Caretaker. As a result, the Memento object needs to be stored as an Object within the Caretaker.

Now, let's look at an example. The DietInfo class is our Originator class. We'd like to be able to save and restore its state. It contains 3 fields: a dieter name field, the day number of the diet, and the weight of the dieter on the specified day of the diet.

This class contains a private inner class called Memento. This is our Memento class that is used to save the state of DietInfo. Memento has 3 fields representing the dieter name, the day number, and the weight of the dieter.

Notice the save() method of DietInfo. This creates and returns a Memento object. This returned Memento object gets stored by the caretaker. Note that DietInfo.Memento is not visible, so the caretaker can't reference DietInfo.Memento. Instead, it stores the reference as an Object.

The restore() method of DietInfo is used to restore the state of the DietInfo. The caretaker passes in the Memento (as an Object). The memento is cast to a Memento object and then the DietInfo object's state is restored by copying over the values from the memento.

```
package za.ac.cput.kabasob.designpatterns.behavioral.memento;

/**
 *
 * @author boniface
 */

// originator - object whose state we want to save
public class DietInfo {

    String personName;
    int dayNumber;
    int weight;

    public DietInfo(String personName, int dayNumber, int weight) {
        this.personName = personName;
        this.dayNumber = dayNumber;
        this.weight = weight;
    }

    public String toString() {
        return "Name: " + personName + ", day number: " + dayNumber + ", weight: " + weight;
    }

    public void setDayNumberAndWeight(int dayNumber, int weight) {
        this.dayNumber = dayNumber;
        this.weight = weight;
    }

    public Memento save() {
        return new Memento(personName, dayNumber, weight);
    }

    public void restore(Object objMemento) {
        Memento memento = (Memento) objMemento;
        personName = memento.mementoPersonName;
        dayNumber = memento.mementoDayNumber;
        weight = memento.mementoWeight;
    }

    // memento - object that stores the saved state of the originator
    private class Memento {
        String mementoPersonName;
        int mementoDayNumber;
        int mementoWeight;

        public Memento(String personName, int dayNumber, int weight) {
            mementoPersonName = personName;
            mementoDayNumber = dayNumber;
            mementoWeight = weight;
        }
    }
}
```

DietInfoCaretaker is the caretaker class that is used to store the state (ie, the memento) of a DietInfo object (ie, the originator). The memento is stored as an object since DietInfo.Memento is not visible to the caretaker. This protects the integrity of the data stored in the Memento object. The caretaker's saveState() method saves the state of the DietInfo object. The caretaker's restoreState() method restores the state of the DietInfo object.

```
package za.ac.cput.kabasob.designpatterns.behavioral.memento;

/**
 *
 * @author boniface
 */
// caretaker - saves and restores a DietInfo object's state via a memento
// note that DietInfo.Memento isn't visible to the caretaker so we need to cast the memento ↩
    to Object
public class DietInfoCaretaker {

    Object objMemento;

    public void saveState(DietInfo dietInfo) {
        objMemento = dietInfo.save();
    }

    public void restoreState(DietInfo dietInfo) {
        dietInfo.restore(objMemento);
    }
}
```

The MainDriver class demonstrates the memento pattern. It creates a caretaker and then a DietInfo object. The DietInfo object's state is changed and displayed. At one point, the caretaker saves the state of the DietInfo object. After this, the DietInfo object's state is further changed and displayed. After this, the caretaker restores the state of the DietInfo object. We verify this restoration by displaying the DietInfo object's state.

```
package za.ac.cput.kabasob.designpatterns.behavioral.memento;

/**
 *
 * @author boniface
 */
public class MainDriver {

    public static void main(String[] args) {

        // caretaker
        DietInfoCaretaker dietInfoCaretaker = new DietInfoCaretaker();

        // originator
        DietInfo dietInfo = new DietInfo("Fred", 1, 100);
        System.out.println(dietInfo);

        dietInfo.setDayNumberAndWeight(2, 99);
        System.out.println(dietInfo);

        System.out.println("Saving state.");
        dietInfoCaretaker.saveState(dietInfo);

        dietInfo.setDayNumberAndWeight(3, 98);
        System.out.println(dietInfo);

        dietInfo.setDayNumberAndWeight(4, 97);
        System.out.println(dietInfo);

        System.out.println("Restoring saved state.");
        dietInfoCaretaker.restoreState(dietInfo);
        System.out.println(dietInfo);

    }
}
```

Record output in your log book

Notice how the state changes, and how we are able to save and restore the state of the originator via the caretaker's reference to the memento.

5.8 What is Software Re-factoring

Re factoring is the process of changing a software system such that the external behavior of the system does not change e.g. functional requirements are maintained but the internal structure of the system is improved

This is sometimes called Improving the design after it has been written

The purpose of refactoring is to make software easier to understand and modify. Contrast this with performance optimization. Again functionality is not changed, only internal structure. However performance optimizations often involve making code harder to understand (but faster!)

When you systematically apply refactoring, you wear two hats adding function, functionality is added to the system without spending any time cleaning the code.

During refactoring no functionality is added, but the code is cleaned up, made easier to understand and modify, and sometimes is reduced in size

The process of refactoring involves the removal of duplication, the simplification of complex logic, and the clarification of unclear code. When you refactor, you relentlessly poke and prod your code to improve its design. Such improvements may involve something as small as changing a variable name or as large as unifying two hierarchies.

To refactor safely, you must either manually test that your changes didn't break anything or run automated tests. You will have more courage to refactor and be more willing to try experimental designs if you can quickly run automated tests to confirm that your code still works. Refactoring in small steps helps prevent the introduction of defects. Most refactorings take seconds or minutes to perform. Some large refactorings can require a sustained effort for days, weeks, or months until a transformation has been completed. Even such large refactorings are implemented in small steps.

It's best to refactor continuously, rather than in phases. When you see code that needs improvement, improve it. On the other hand, if your manager needs you to finish a feature before a demo that just got scheduled for tomorrow, finish the feature and refactor later. Business is well served by continuous refactoring, yet the practice of refactoring must coexist harmoniously with business priorities.

5.8.1 Why Refactor

The idea behind refactoring is to acknowledge that it will be difficult to get a design right the first time and, as a programs requirements change, the design may need to change. Refactoring provides techniques for evolving the design in small incremental steps.

Benefits includes:

- To reduce code size
- Transform Confusing structures into simpler structures which are easier to maintain and understand
- Make it easier to add new code.
- Improve the design of existing code.
- Gain a better understanding of code.

5.8.1.1 Signs of Bad Code

Refactoring, or improving the design of existing code, requires that you know what code needs improvement. It's therefore necessary to learn common design problems so you can recognize them in your own code. The most common design problems result from code that

- Is duplicated
- Is unclear
- Is complicated

Below are some of the pointers that can help you discover places in code that need improvement.

Duplicated Code

Duplicated code is the most pervasive and biggest problem in software. It tends to be either explicit or subtle. Explicit duplication exists in identical code, while subtle duplication exists in structures or processing steps that are outwardly different yet essentially the same.

You can often remove explicit and/or subtle duplication in subclasses of a hierarchy by applying the **Template Method**. If a method in the subclasses is implemented similarly, except for an object creation step, applying Introduce Polymorphic Creation with **Factory Method** will pave the way for removing more duplication by means of a **Template Method**.

If the constructors of a class contain duplicated code, you can often eliminate the duplication by applying Chain Constructors of using a Builder Pattern. If you have separate code for processing a single object or a collection of objects, you may be able to remove duplication by applying Replace One/Many Distinctions with **Composite**..

If subclasses of a hierarchy each implement their own Composite, the implementations may be identical, in which case you can use Extract **Composite**.

If you process objects differently merely because they have different interfaces, applying Unify Interfaces with **Adapter** will pave the way for removing duplicated processing logic.

If you have conditional logic to deal with an object when it is null and the same null logic is duplicated throughout your system, applying Introduce Null Object, an object that checks for null and return some sensible value, will eliminate the duplication and simplify the system.

Long Method

Short methods are superior to long methods beacause of a principal reason involving the sharing of logic. Two long methods may very well contain duplicated code. Yet if you break those methods into smaller ones, you can often find ways for them to share logic.

If you don't understand what a chunk of code does and you extract that code to a small, well-named method, it will be easier to understand the original code. Systems that have a majority of small methods tend to be easier to extend and maintain because they're easier to understand and contain less duplication.

What is the preferred size of small methods? **I would say ten lines of code or fewer**, with the majority of your methods using one to five lines of code. If you make the vast majority of a system's methods small, you can have a few methods that are larger, as long as they are simple to understand and don't contain duplication.

When I'm faced with a long method, one of your first impulses should be to break it down into a Composed Method by applying the refactoring Compose Method. This work usually involves applying Extract Method.

If your method is long because it contains a large switch statement for dispatching and handling requests, you can shrink the method by using Replace Conditional Dispatcher with Command.

If you use a switch statement to gather data from numerous classes with different interfaces, you can shrink the size of the method by applying Move Accumulation to Visitor.

If a method is long because it contains numerous versions of an algorithm and conditional logic to choose which version to use at runtime, you can shrink the size of the method by applying Replace Conditional Logic with Strategy Pattern.

Conditional Complexity

Conditional logic is innocent in its infancy, when it is simple to understand and contained within a few lines of code. Unfortunately, it rarely ages well. For example, you implement several new features and suddenly your conditional logic becomes complicated and expansive.

If conditional logic controls which of several variants of a calculation to execute, consider applying Replace Conditional Logic with Strategy Pattern.

If conditional logic controls which of several pieces of special-case behavior must be executed in addition to the class's core behavior, you may use the Decorator Pattern.

If the conditional expressions that control an object's state transitions are complex, consider simplifying the logic by applying Replace State-Altering Conditionals with State Pattern

Dealing with null cases often leads to the creation of conditional logic. If the same null conditional logic is duplicated throughout your system, you can clean it up by using Introduce Null Object.

Primitive Obsession

Primitives, which include integers, strings, doubles, arrays, and other low-level language elements, are generic because many people use them. Classes, on the other hand, may be as specific as you need them to be because you create them for specific purposes.

In many cases, classes provide a simpler and more natural way to model things than primitives. In addition, once you create a class, you'll often discover that other code in a system belongs in that class.

How does Primitive Obsession manifests itself when code relies too much on primitives? This typically occurs when you haven't yet seen how a higher-level abstraction can clarify or simplify your code.

If a primitive value controls logic in a class and the primitive value isn't type-safe (i.e., clients can assign it to an unsafe or incorrect value), consider applying Replace Type Code with Class. The result will be code that is type-safe and capable of being extended by new behavior (something you can't do with a primitive).

If an object's state transitions are controlled by complex conditional logic that uses primitive values, you can use Replace State-Altering Conditionals with State. The result will be numerous classes to represent each state and simplified state transition logic.

If complicated conditional logic controls which algorithm to run and that logic relies on primitive values, consider applying Replace Conditional Logic with Strategy.

If you implicitly create a tree structure using a primitive representation, such as a string, your code may be difficult to work with, prone to errors, and/or filled with duplication. Applying Replace Implicit Tree with Composite will reduce these problems.

If many methods of a class exist to support numerous combinations of primitive values, you may have an implicit language. If so, consider applying Replace Implicit Language with Interpreter.

If primitive values exist in a class only to provide embellishments to the class's core responsibility, you may want to use Move Embellishment to Decorator .

Finally, even if you have a class, it may still be too primitive to make life easy for clients. This may be the case if you have a Composite implementation that is tricky to work with. You can simplify how clients build the Composite by applying Encapsulate Composite with Builder.

Indecent Exposure

This problem indicates the lack of what is so famously termed "information hiding" . The problem occurs when methods or classes that ought not be visible to clients are publicly visible to them. Exposing such code means that clients know about code that is unimportant or only indirectly important. This contributes to the complexity of a design.

The refactoring Encapsulate Classes with Factory removes this problem. Not every class that is useful to clients needs to be public (i.e., have a public constructor). Some classes ought to be referenced only via their common interfaces. You can make that happen if you make the class's constructors non-public and use a Factory to produce instances.

Solution Sprawl

When code and/or data used to perform a responsibility becomes sprawled across numerous classes, Solution Sprawl is in the air. This problem often results from quickly adding a feature to a system without spending enough time simplifying and consolidating the design to best accommodate the feature.

Solution Sprawl is the identical twin brother of Shotgun Surgery, a problem described by Fowler and Beck in their refactoring book. You become aware of this problem when adding or updating a system feature causes you to make changes to many different pieces of code. Solution Sprawl and Shotgun Surgery address the same problem, yet are sensed differently. We become aware of Solution Sprawl by observing it, while we become aware of Shotgun Surgery by doing it.

Move Creation Knowledge to Factory is a refactoring that solves the problem of a sprawling object creation responsibility.

Alternative Classes with Different Interfaces

This problem occurs when the interfaces of two classes are different and yet the classes are quite similar. If you can find the similarities between the two classes, you can often refactor the classes to make them share a common interface.

However, sometimes you can't directly change the interface of a class because you don't have control over the code. The typical example is when you're working with a third-party library. In that case, you can apply Unify Interfaces with Adapter to produce a common interface for the two classes.

Lazy Class

This is a class that isn't doing enough to pay for itself and should be eliminated. It's not uncommon to encounter a Singleton that isn't paying for itself. In fact, the Singleton may be costing you something by making your design too dependent on what amounts to global data. Inline Singleton explains a quick, humane procedure for eliminating a Singleton.

Large Class

The presence of too many instance variables usually indicates that a class is trying to do too much. In general, large classes typically contain too many responsibilities.

Extract Class and Extract Subclass , which are some of the main refactorings used to address this problem, help move responsibilities to other classes.

Replace Conditional Dispatcher with Command extracts behavior into Command classes, which can greatly reduce the size of a class that performs a variety of behaviors in response to different requests.

Replace State-Altering Conditionals with State can reduce a large class filled with state transition code into a small class that delegates to a family of State classes.

Replace Implicit Language with Interpreter can reduce a large class into a small one by transforming copious code for emulating a language into a small Interpreter .

Switch Statements

Switch statements **(or their equivalent, ifelseifelseif structures)** aren't inherently bad. They become bad only when they make your design more complicated or rigid than it needs to be. In that case, it's best to refactor away from switch statements to a more object-based or polymorphic solution.

Replace Conditional Dispatcher with Command describes how to break down a large switch statement into a collection of Command objects, each of which may be looked up and invoked without relying on conditional logic.

Move Accumulation to Visitor describes an example where switch statements are used to obtain data from instances of classes that have different interfaces. By refactoring the code to use a Visitor, no conditional logic is needed and the design becomes more flexible.

Combinatorial Explosion

This smell is a subtle form of duplication. It exists when you have numerous pieces of code that do the same thing using different kinds or quantities of data or objects.

For example, say you have numerous methods on a class for performing queries. Each of these methods performs a query using specific conditions and data. The more specialized queries you need to support, the more query methods you must create. Pretty soon you have an explosion of methods to handle the many ways of performing queries. You also have an implicit query language. You can remove all of these methods and the combinatorial explosion smell by applying Replace Implicit Language with Interpreter.

Oddball Solution

When a problem is solved one way throughout a system and the same problem is solved another way in the same system, one of the solutions is the oddball or inconsistent solution. The presence of this smell usually indicates subtly duplicated code.

To remove this duplication, first determine your preferred solution. In some cases, the solution used least often may be your preferred solution if it is better than the solution used most of the time. After determining your preferred solution, you can often apply Substitute Algorithm to produce a consistent solution throughout your system. Given a consistent solution, you may be able to move all instances of the solution to one place, thereby removing duplication.

The Oddball Solution problem is usually present when you have a preferred way to communicate with a set of classes, yet differences in the interfaces of the classes prevent you from communicating with them in a consistent way. In that case, consider applying Unify Interfaces with Adapter to produce a common interface by which you may communicate consistently with all of the classes. Once you do that, you can often discover ways to remove duplicated processing logic

5.9 Chapter Exercises

This exercise has two parts.

Part 1

The First part requires you to read and master 6 design patterns, two from each category. On the date to be announced, you will be required to go to the front of the class and take questions about the patterns you have mastered. These are the areas you are expected to master of respective patterns you pick.

1. Pattern Name and Classification
   - Creational
   - Structural
   - Behavioural
2. Intent Also Known As Motivation and Applicability
3. Structure and Participants
4. Collaborations
5. Consequences
6. Implementation
7. Sample Code
8. Known Uses
9. Related Patterns

Part 2

Re-Write the Design Patterns in Section, but use JUnit Tests in place of the Driver class used in the examples.

Chapter 6

Common Architectures

6.1 Chapter Objectives

1. Explain the advantages and disadvantages of N-tier architectures when examined under the following topics: scalability, maintainability, reliability, availability, extensibility, performance, manageability, and security.
2. Explain the benefits and drawbacks of rich clients and browser-based clients as deployed in a typical Enterprise Application.
3. Explain appropriate and inappropriate uses for Web Services in an enterprise Application.

6.2 Introduction

This chapter will explain the decomposition of the larger system into smaller components and advantages and disadvantages of decomposing by tiers and/or layers. The major theme of architecture is the decomposition of the larger system into smaller components that can be built in relative isolation, as well as provide for the service-level requirements.

6.3 Decomposition Strategies

Decomposition can be broken down into ten basic strategies: layering, distribution, exposure, functionality, generality, coupling and cohesion,volatility, configuration, planning and tracking, and work assignment. These ten strategies can be grouped together, but not all ten are applied in any given architecture. For any strategies that are grouped together, you choose one of the strategies and then move on to the next grouping.

Here are the groups:

- Group 1 Either Layering or Distribution
- Group 2 Either Exposure, Functionality, or Generality

- Group 3 Either Coupling and Cohesion or Volatility
- Group 4 Configuration
- Group 5 Either Planning and Tracking or Work Assignment

Grouping the strategies in this manner enables you to combine strategies that are related and will not be typically applied together.

For example, if you are to decompose by layering, you will not typically decompose by distribution as well. You will notice that the groups are also ordered so that the last decomposition strategy is by Planning or Work Assignment. You would not start decomposing your system by Work Assignment and then move to Functionality.

6.3.1 Group 1 Layering or Distribution

6.3.1.1 Layering

Layering decomposition is some ordering of principles, typically abstraction. The layers may be totally or partially ordered, such that a given layer x uses the services of layer y, and x in turn provides higher-level services to any layer that uses it. Layering can be by layers or tiers, as explained later in the chapter. Layering is usually a top-level decomposition and is followed by one of the other rules.

6.3.1.2 Distribution

Distribution is among computational resources, along the lines of one or more of the following:

- Dedicated tasks own their own thread of control, avoiding the problem of a single process or thread going into a wait state and not being able to respond to its other duties.
- Multiple clients may be required.
- Process boundaries can offer greater fault isolation.
- Distribution for separation may be applied, perhaps with redundancy, for higher reliability.

Distribution is a primary technique for building scalable systems. Because the goals and structure of process/threads is often orthogonal to other aspects of the system, it typically cuts across many subsystems and is therefore often difficult to manage if it is buried deep in a systems structure. More often than not, if you decompose by layering, you will not decompose by distribution and vice versa.

6.3.2 Group 2 Exposure, Functionality, or Generality

6.3.2.1 Exposure

Exposure decomposition is about how the component is exposed and consumes other components. Any given component fundamentally has three different aspects: services, logic, and integration. Services deals with how other components access this component. Logic deals with how the component implements the work necessary to accomplish its task. Integration deals with how it accesses other components services.

6.3.2.2 Functionality

Functionality decomposition is about grouping within the problem spacethat is, order module or customer module. This type of decomposition is typically done with the operational process in mind.

6.3.2.3 Generality

Generality decomposition is determining whether you have a reusable component that can be used across many systems. Some parts of a system are only usable within the existing system, whereas other parts can be used by many systems. Be careful not to make assumptions that a component may be used by another system in the future and build a reusable component for a requirement that does not exist yet.

6.3.3 Group 3 Coupling and Cohesion or Volatility

6.3.3.1 Coupling and Cohesion

Coupling and Cohesion decomposition, as in low coupling and high cohesion, is keeping things together that work together (high cohesion), but setting apart things that work together less often (low coupling).

6.3.3.2 Volatility

Volatility decomposition is about isolating things that are more likely to change. For example, GUI changes are more likely than the underlying business rules. Again, be careful not to make assumptions that are not documented in requirements, as this can create a complex system.

6.3.4 Group 4 Configuration

6.3.4.1 Configuration

Configuration decomposition is having a target system that must support different configurations, maybe for security, performance, or usability. Its like having multiple architectures with a shared core, and the only thing that changes is the configuration.

6.3.5 Group 5 Planning and Tracking or Work Assignment

6.3.5.1 Planning and Tracking

Planning and Tracking decomposition is an attempt to develop a fine grained project plan that takes into account ordering dependencies and size. Ordering is understanding the dependencies between packages and realizing which must be completed first. A good architecture will have few, if any, bi-directional or circular dependencies. Sizing is breaking down the work into small-enough parts so you can develop in a iterative fashion without an iteration taking several months.

6.3.5.2 Work Assignment

Work Assignment decomposition is based on various considerations, including physically distributed teams, skill-set matching, and security areas. As an architect, you need to anticipate and determine composition of teams for design and implementation.

6.3.6 Putting it Together

To start the decomposition process, you would select a decomposition strategy from group 1 and determine if you have decomposed the architecture sufficiently for it to be built.

If not, then you move to group 2 and select a strategy for decomposition and evaluate the architecture again. You continue to decompose using a strategy from each group if it applies until you have the system broken down into small-enough components to start building.

Something else to keep in mind during your decomposition is the notion of tiers and layers.

6.4 Tiers and Layers

6.4.1 Tiers

A tier can be logical or physical organization of components into an ordered chain of service providers and consumers. Components within a tier typically consume the services of those in an adjacent provider tier and provide services to one or more adjacent consumer tiers.

Traditional tiers in an architecture are client, web/presentation, business, integration, and resource.

6.4.2 Client

A client tier is any device or system that manages display and local interaction processing. Enterprises may not have control over the technologies available on the client platform, an important consideration in tier structuring.

For this reason, the client tier should be transient and disposable.

6.4.3 Web

Web tiers consist of services that aggregate and personalize content and services for channel-specific user interfaces. This entails the assembly of content, formatting, conversions, and content transformationsanything that has to do with the presentation of information to end users or external systems. These services manage channel-specific user sessions and translate inbound application requests into calls to the appropriate business services. The web tier is also referred to as the presentation tier.

6.4.4 Business

Business tier services execute business logic and manage transactions. Examples range from low-level services, such as authentication and mail transport, to true line-of-business services, such as order entry, customer profile, payment, and inventory management.

6.4.5 Integration

Integration tier services abstract and provide access to external resources. Due to the varied and external nature of these resources, this tier often employs loosely coupled paradigms, such as queuing, publish/ subscribe communications, and synchronous and asynchronous point-to-point messaging. Upper-platform components in this tier are typically called middleware.

6.4.6 Resource

The resource tier includes legacy systems, databases, external data feeds, specialized hardware devices such as telecommunication switches or factory automation, and so on. These are information sources, sinks, or stores that may be internal or external to the system. The resource tier is accessed and abstracted by the integration tier. The resource tier is also referred to as the data tier.

6.5 Layers

A layer is the hardware and software stack that hosts services within a given tier. Layers, like tiers, represent a well-ordered relationship across interface-mediated boundaries. Whereas tiers represent processing chains across components, layers represent container/component relationships in implementation and deployment of services. Typical layers are application, virtual platform, application infrastructure, enterprise services, compute and storage, and networking infrastructure.

6.5.1 Application

The application layer combines the user and business functionality of a system on a middleware substrate. It is everything left after relegating shared mechanisms (middleware) to the application infrastructure layer, lower-level general purpose capabilities to the enterprise services layer, and the enabling infrastructure to the compute and storage layer. The application layer is what makes any particular system unique.

6.5.2 Virtual Platform (Component APIs)

The virtual platform layer contains interfaces to the middleware modules in the application infrastructure layer. Examples of this layer include the component APIs, such as EJBs, Servlets, and the rest of the Java EE APIs. The application is built on top of the virtual platform component APIs.

6.5.3 Application Infrastructure (Containers)

The application infrastructure layer contains middleware products that provide operational and developmental infrastructure for the application. Glassfish is an example of a container in the application infrastructure. The virtual platform components are housed in an application infrastructure container.

6.5.4 Enterprise Services (OS and Virtualization)

The enterprise services layer is the operating system and virtualization software that runs on top of the compute and storage layer. This layer provides the interfaces to operating system functions needed by the application infrastructure layer.

6.5.5 Compute and Storage

The compute and storage layer consists of the physical hardware used in the architecture. Enterprise services run on the compute and storage layer.

6.5.6 Networking Infrastructure

The networking infrastructure layer contains the physical network infrastructure, including network interfaces, routers, switches, load balancers, connectivity hardware, and other network elements.

6.6 Service-Level Requirements

In addition to the business requirements of a system, you must satisfy the service-level or quality of service (QoS) requirements, also known as non-functional requirements. As an architect, it is your job to work with the stakeholders of the system during the inception and elaboration phases to define a quality of service measurement for each of the service- level requirements. The architecture you create must address the following service-level requirements: performance, scalability, reliability, availability, extensibility, maintainability, manageability, and security. You will have to make trade-offs between these requirements.

For example, if the most important service-level requirement is the performance of the system, you might sacrifice the maintainability and extensibility of the system to ensure that you meet the performance quality of service. As the expanding Internet opens more computing opportunities, the service-level requirements are becoming increasingly more importantthe users of these Internet systems are no longer just the company employees, but they are now the companys customers.

6.6.1 Performance

The performance requirement is usually measured in terms of response time for a given screen transaction per user. In addition to response time, performance can also be measured in transaction throughput, which is the number of transactions in a given time period, usually one second.

For example, you could have a performance measurement that could be no more than three seconds for each screen form or a transaction throughput of one hundred transactions in one second. Regardless of the measurement, you need to create an architecture that allows the designers and developers to complete the system without considering the performance measurement.

6.6.2 Scalability

Scalability is the ability to support the required quality of service as the system load increases without changing the system. A system can be considered scalable if, as the load increases, the system still responds within the acceptable limits. It might be that you have a performance measurement of a response time between two and five seconds. If the system load increases and the system can maintain the performance quality of service of less than a five-second response time, your system is scalable. To understand scalability, you must first understand the capacity of a system, which is defined as the maximum number of processes or users a system can handle and still maintain the quality of service.

If a system is running at capacity and can no longer respond within an acceptable time frame, it has reached its maximum scalability. To scale a system that has met capacity, you must add additional hardware. This additional hardware can be added vertically or horizontally.

Vertical scaling involves adding additional processors, memory, or disks to the current machine(s). Horizontal scaling involves adding more machines to the environment, thus increasing the overall system capacity. The architecture you create must be able to handle the vertical or horizontal scaling of the hardware. Vertical scaling of a software architecture is easier than the horizontal scaling. Why? Adding more processors or memory typically does not have an impact on your architecture, but having your architecture run on multiple machines and still appear to be one system is more difficult.

6.6.3 Reliability

Reliability ensures the integrity and consistency of the application and all its transactions. As the load increases on your system, your system must continue to process requests and handle transactions as accurately as it did before the load increased. Reliability can have a negative impact on scalability. If the system cannot maintain the reliability as the load increases, the system is really not scalable. So, for a system to truly scale, it must be reliable.

6.6.4 Availability

Availability ensures that a service/resource is always accessible. Reliability can contribute to availability, but availability can be achieved even if components fail. By setting up an environment of redundant components and failover, an individual component can fail and have a negative impact on reliability, but the service is still available due to the redundancy.

6.6.5 Extensibility

Extensibility is the ability to add additional functionality or modify existing functionality without impacting existing system functionality. You cannot measure extensibility when the system is deployed, but it shows up the first time you must extend the functionality of the system. You should consider the following when you create the architecture and design to help ensure extensibility: low coupling, interfaces, and encapsulation.

6.6.6 Maintainability

Maintainability is the ability to correct flaws in the existing functionality without impacting other components of the system. This is another of those systemic qualities that you cannot measure at the time of deployment. When creating an architecture and design, you should consider the following to enhance the maintainability of a system: low coupling, modularity, and documentation.

6.6.7 Manageability

Manageability is the ability to manage the system to ensure the continued health of a system with respect to scalability, reliability, availability, performance, and security. Manageability deals with system monitoring of the QoS requirements and the ability to change the system configuration to improve the QoS dynamically without changing the system. Your architecture must have the ability to monitor the system and allow for dynamic system configuration.

6.6.8 Security

Security is the ability to ensure that the system cannot be compromised. Security is by far the most difficult systemic quality to address. Security includes not only issues of confidentiality and integrity, but also relates to Denial-of-Service (DoS) attacks that impact availability. Creating an architecture that is separated into functional components makes it easier to secure the system because you can build security zones around the components. If a component is compromised, it is easier to contain the security violation to that component.

6.7 Impact of Dimensions on Service-Level Requirements

As you are creating your architecture, and from a system computational point of view, you can think of the layout of an architecture (tiers and layers) as having six independent variables that are expressed as dimensions. These variables are as follows:

1. Capacity
2. Redundancy
3. Modularity
4. Tolerance
5. Workload
6. Heterogeneity

6.7.1 Capacity

The capacity dimension is the raw power in an element, perhaps CPU, fast network connection, or large storage capacity. Capacity is increased through vertical scaling and is sometimes referred to as height. Capacity can improve performance, availability, and scalability.

6.7.2 Redundancy

The redundancy dimension is the multiple systems that work on the same job, such as load balancing among several web servers. Redundancy is increased through horizontal scaling and is also known as width. Redundancy can increase performance, reliability, availability, extensibility, and scalability. It can decrease performance, manageability, and security.

6.7.3 Modularity

The modularity dimension is how you divide a computational problem into separate elements and spread those elements across multiple computer systems. Modularity indicates how far into a system you have to go to get the data you need. Modularity can increase scalability, extensibility, maintainability, and security. It can decrease performance, reliability, availability, and manageability.

6.7.4 Tolerance

The tolerance dimension is the time available to fulfill a request from a user. Tolerance is closely bound with the overall perceived performance. Tolerance can increase performance, scalability, reliability, and manageability.

6.7.5 Workload

The workload dimension is the computational work being performed at a particular point within the system. Workload is closely related to capacity in that workload consumes available capacity, which leaves fewer resources available for other tasks. Workload can increase performance, scalability, and availability.

6.7.6 Heterogeneity

The heterogeneity dimension is the diversity in technologies that is used within a system or one of its subsystems. Heterogeneity comes from the variation of technologies that are used within a system. This might come from a gradual accumulation over time, inheritance, or acquisition. Heterogeneity can increase performance and scalability. It can decrease performance, scalability, availability, extensibility, manageability, and security.

6.8 Common Practices for Improving Service-Level Requirements

Over the years, software and system engineering practices have developed many best practices for improving systemic qualities. By applying these practices to the system at the architecture level, you can gain a higher level of assurance for the success of the system development. Introducing Redundancy to the System Architecture Many infrastructure-level practices for improving systemic qualities rely on using redundant components in the system. You can apply these strategies to either the vendor products or the server systems themselves. The choice depends primarily on the cost of implementation and the requirements, such as performance and scalability.

6.8.1 Load Balancing

You can implement load balancing to address architectural concerns, such as throughput and scalability. Load balancing is a feature that allows server systems to redirect a request to one of several servers based on a predetermined load-balancing algorithm. Load balancing is supported by a wide variety of products, from switches to server systems, to application servers. The advantage of load balancing is that it lets you distribute the workload across several smaller machines instead of using one large machine to handle all the incoming requests. This typically results in lower costs and better use of computing resources. To implement load balancing, you usually select a load-balancer implementation based on its performance and availability. Consider the following:

- Load balancers in network switchesLoad balancers that are included with network switches and are commonly implemented in firmware, which gives them the advantage of speed.
- Load balancers in cluster management software and application serversLoad balancers that are implemented with software are managed closer to the application components, which gives greater flexibility and manageability.
- Load balancers based on the server instance DNS configuration Load balancer is configured to distribute the load to multiple server instances that map to the same DNS host name. This approach has the advantage of being simple to set up, but typically it does not address the issue of session affinity. Load balancers also provide a variety of algorithms for the decision-making component. There are several standard solutions from which to choose, as follows:
- Round-robin algorithmPicks each server in turn.
- Response-time or first-available algorithmConstantly monitors the response time of the servers and picks the one that responds the quickest.
- Least-loaded algorithmConstantly monitors server load and selects the server that has the most available capacity.
- Weighted algorithmSpecifies a priority on the preceding algorithms, giving some servers more workload than others.
- Client DNS-based algorithmDistribute the load based on the clients DNS host and domain name information.

In addition to these solutions, most load-balancer implementations enable you to create your own load-balancing strategy and install it for use. Your selection of a load-balancing strategy is largely based on the type of servers you are managing, how you would like to distribute the workload, and what the application domain calls for in performance. For example, if you have equally powerful machines and a fairly even distribution of transaction load in your application, it would make little sense to use a weighted algorithm for load balancing. This approach could result in an overloaded system that might fail. An equal distribution of workload would make more sense.

6.8.2 Failover

Failover is another technique that you can use to minimize the likelihood of system failure. Failover is a system configuration that allows one server to assume the identity of a failing system within a network. If, at any point in time, a server goes down due to overloading, internal component failure, or any other reason, the processes and state of that server are automatically transferred to the failover server. This alternative server then assumes the identity of the failed system and processes any further requests on behalf of that system. One important aspect of failover is available capacity, which can be handled in two ways:

- Designing with extra capacityIf you design a server group with extra capacity, all the systems work for you, but at low usage levels. This means that you are spending money on extra computing resources that will not be used under normal load and operation conditions.
- Maintaining a stand-by serverIf you design a server group to have a stand-by server, you are spending money on a system that does no work whatsoever, unless (or until) it is needed as a failover server.In this approach, the money spent on unused computing resources is not the important thing to keep in mind.

Instead, you should view the expenditure as insurance. You pay for the stand-by server and hope that you will never have to use it, but you can rest easier knowing that the stand-by server is there in case you ever need it.

6.8.3 Clusters

Clusters also minimize the likelihood of system failure. A cluster is a group of server systems and support software that is used to manage the server group. Clusters provide high availability to system resources. Cluster software allows group administration, detects hardware and software failure, handles system failover, and automatically restarts services in the event of failure.

The following cluster configurations are available:

- Two-node clusters (symmetric and asymmetric)A configuration for which you can either run both servers at the same time (symmetric), or use one server as a stand-by failover server for the other (asymmetric).
- Clustered pairsA configuration that places two machines into a cluster, and then uses two of these clusters to manage independent services. This configuration enables you to manage all four machines as a single cluster. This configuration is often used for managing highly coupled data services, such as an application server and its supporting database server.

- Ring (not supported in Sun Cluster 3.0 Cool Stuff software) A configuration topology that allows any individual node to accept the failure of one of its two neighboring nodes.
- N+1 (Star)A configuration that provides N independent nodes, plus 1 backup node to which all the other systems fail over. This system must be large enough to accept the failover of as many systems as you are willing to allow to fail.
- Scalable (N-to-N)A configuration that has several nodes in the cluster, and all nodes have uniform access to the data storage medium. The data storage medium must support the scalable cluster by providing a sufficient number of simultaneous node connections.

6.9 Making Improvements to Non-Functional Requirements

6.9.1 Improving Performance

The two factors that determine the system performance are as follows:

- **Processing time** : The processing time includes the time spent in computing, data marshaling and unmarshaling, buffering, and transporting over a network.
- **Blocked time**: The processing of a request can be blocked due to the contention for resources, or a dependency on other processing. It can also be caused by certain resources not available; for example, an application might need to run an aggressive garbage collection to get more memory available for the processing. The following practices are commonly used to increase the system performance:
- Increase the system capacity by adding more raw processing power.
- Increase the computation efficiency by using efficient algorithms and appropriate component models technologies. Introduce cached copies of data to reduce the computation overhead, as follows:
- Introduce concurrency to computations that can be executed in parallel.
- Limit the number of concurrent requests to control the overall system utilization.
- Introduce intermediate responses to improve the performance perceived by the user.

To improve the system throughput, it is common that a timeout is applied to most of the long-lasting operations, especially those involving the access to an external system.

6.9.2 Improving Availability

The factors that affect the system availability include the following:

- **System downtime**: The system downtime can be caused by a failure in hardware, network, server software, and application component.

- **Long response time:** If a component does not produce a response quick enough, the system can be perceived to be unavailable.

The most common practice to improve the system availability is through one of the following types of replication, in which redundant hardware and software components are introduced and deployed:

- **Active replication** : The request is sent to all the redundant components, which operate in parallel, and only one of the generated responses is used. Because all the redundant components receive the same request and perform the same computation, they are automatically synchronized. In active replication, the downtime can be short because it involves only component switching.
- **Passive replication**: Only one of the replicated components (the primary component) responds to the requests. The states of other components (secondary) are synchronized with the primary component. In the event of a failure, the service can be resumed if a secondary component has a sufficiently fresh state.

6.9.3 Improving Extensibility

The need for extensibility is typically originated from the change of a requirement. One of the most important goals of the architecture is to facilitate the development of the system that can quickly adapt to the changes. When you create the architecture, consider the following practices for the system extensibility:

- **Clearly define the scope in the service-level agreement** : Scope change is one of the most common reasons for project failure. Defining a clear scope is the first step to limiting unexpected changes to a system.
- **Anticipate expected changes**: You should identify the commonly changed areas of the system (for example, the user interface technology), and then isolate these areas into coherent components. By doing this, you can prevent ripple effects of propagating the change across the system.
- **Design a high-quality object model**: The object model of the system typically has an immediate impact on its extensibility and flexibility. Therefore, you should consider applying essential object-oriented (OO) principles and appropriate architectural and design patterns to the architecture.

6.9.4 Improving Scalability

You can configure scalability in the following two ways:

- **Vertical scalability**: Adding more processing power to an existing server system, such as processors, memory, and disks (increase the height of the system). Sometimes, replacing an existing server with a completely new but more capable system is also considered vertical scaling.

- **Horizontal scalability**: Adding additional runtime server instances to host the software system, such as additional application server instances (increase the width of the system).

Vertically scaling a system is transparent to system architecture. However, the physical limitation of a server system and the high cost of buying more powerful hardware can quickly render this option impractical. On the other hand, horizontally scaling a system does not have the physical limitation imposed by an individual servers hardware system. Another consideration you must take into account is the impact that horizontal scaling has on the system architecture. Typically, to make a system horizontally scalable, not only do you need to use a software system that supports the cluster-based configuration, but you also need to design the application such that the components do not depend on the physical location of others.

6.10 Tiers impact on Service Level Agreements

Lets conclude this chapter talking about how tiers impact the service level requirements. When most of the industry is talking about tiers in an architecture, they are referring to the physical tiers such as client, web server, and database server.

An architecture can have multiple logical tiers, as we previously mentioned, and still be deployed in a two-tier architecture. With the advent of virtualization, the physical deployment is not as critical as it was years ago. Virtualization enables you to have what are perceived as physical tiers on the same physical machine. You could be running the web server and application server on the same physical hardware just in different operating systems, so physical tiers are not as important as the logical tiers and the separation of concerns.

When talking about two-tier, three-tier, or n-tier, the client tier is usually not included unless explicitly stated, as in two-tier client/server.

6.10.1 Two-Tier Systems

Two-tier systems are traditionally called client/server systems. Most twotier systems have a thick client that includes both presentation and business logic and a database on the server. The presentation and business logic were typically tightly coupled. You could also have a browser-based two-tier system with business logic and database on the same server.

6.10.1.1 Advantages

Security is an advantage as most of these systems are behind the corporate firewall, so most security breaches are the result of physical security breaking down and non-employees using an unsecured PC. Performance is usually pretty good unless the company uses extremely old laptops that have minimal memory.

6.10.1.2 Disadvantages

Availability is a disadvantage because if one component fails, then the entire system is unavailable. Scalability is a problem, as the only component you can increase is the database. In order to add new functionality in a two-tier system, you will definitely impact the other components therefore, extensibility fails. Manageability is problematic, as it becomes almost impossible to monitor all the PCs that are running the client code. Maintainability has the same problem as extensibility.

Reliability is not really an advantage or disadvantage in a two-tier system. As the load increases, more requests will be coming to the database, and most databases will be able to handle the increased transaction throughput unless it is already at capacity.

6.10.2 Three- and Multi-Tier Systems

Three-tier systems are comprised of web, business logic, and resources tiers. Multi-tier systems have web, business logic, integration, and resource tiers. They share the same advantages and disadvantages when it comes to non-functional requirements.

6.10.2.1 Advantages

Scalability is improved over a two-tier system as you move the presentation logic away from the client PC onto a server that can be clustered. Availability is also improved with the ability to cluster tiers and provide failover. Extensibility is improved because functionality is separated into different tiers. You could modify presentation with minimal to no impact to the business logic. The same is true for maintainability. Manageability is greatly improved because the tiers are deployed on servers, making it easier to monitor the components. Separating the tiers allows for more points to secure the system, but be careful that you do not impact performance. Performance could be an advantage or disadvantage. Primarily, it is an advantage, as you can spread out the processing over many servers, but it can become a disadvantage if you have to transfer large amounts of data between the servers.

6.10.2.2 Disadvantages

Multi-tier systems are inherently more complex, but when it comes to the ilities, there are no real disadvantages to have a multi-tier system. With that said, just because you have multiple tiers does not mean you have a great architecture. Just remember to not overdo the number of tiers.

Chapter 7

Business Tier Transactions overview

7.1 Chapter Objectives

1. Understand the importance and value of transactions in an enterprise application.
2. Describe the need for transaction control and explain isolation levels
3. Discuss and understand different ways of demarcating transactions enterprise application environment

7.2 Introduction

Up to now our examples have, for the most part, used transactions by default. Most of our examples have used container-managed transactions. As the name suggests, the transaction lifecycle is controlled by the Web container. Container-managed transactions use the Java Transaction API, or JTA. Container-managed transactions have a default start and end point, however it is possible to configure these as we shall see later in this chapter. Java SE applications, which run outside an container and so do not have JTA available, must use resource-local transactions. Resource-local transactions use the EntityManager interface. The application uses EntityManager methods to explicitly start and end transactions.

7.3 Transactions

A transaction is a sequence of one or more steps that add, modify, or delete persistent data. Typically the data is persisted to a database. All steps must succeed in which case the transaction succeeds or is committed. If any one step fails then the transaction as a whole fails. In the case of failure the transaction is rolled back so that the state of the data reverts to the state at the start of the transaction.

The classic example of a transaction is a transfer of funds from Bank Account A to Bank Account B. The transaction consists of two steps:

1. Subtract R100 from Account A
2. Add R100 to Account B

For the transaction to succeed, both these steps have to succeed. Were step 1 to succeed and step 2 to fail for any reason (for example Account B does not exist or is currently unavailable), the transaction would undo or rollback step 1.

Transactions are said to have ACID properties:

- **Atomicity**Either all steps of a transaction succeed or none succeed. Atomicity specifies that all operations within a single transaction must complete together or not at all. In other words, a transaction allows multiple database operations to be applied together. In the event of an error, the entire set of operations is rolled back.
- **Consistency**A transaction either commits in a valid database state or is rolled back to its original valid stateConsistency refers to the requirement that transactions must transition a database from one consistent state to another consistent state. A successful transaction cannot leave the database in a state that violates the integrity constraints of the database or the schema. In other words, transactions must comply with database constraints and referential integrity rules during every insert, update or delete before a transaction may be committed.
- **Isolation**Transaction changes are not visible to other transactions until the transaction commits. Isolation defines the rules about how one running transaction affects or interacts with other concurrently running transactions.

  The isolation strategy used on a transaction is very important.

  If the chosen isolation level is too loose, hard-to-find bugs can be introduced, which may adversely impact the integrity of your data.

  If your isolation level is too high, however, you run the risk of slowing down your application or deadlocking your database. This setting is both application server and database server dependent. While there are technically eight isolation levels, generally you will only need to concern yourself with the four that are defined by the ANSI/ISO SQL standard.

  You should also note that the default isolation level varies quite a bit amongst DBMS vendors.
- **Durability**Committed data is permanent and survives any system crashes. Durability ensures that once a transaction is committed, the changes will not be lost and should survive database failures.

The atomicity, consistency, and durability properties are desirable in their entirety. However, as we shall see later in this chapter, there are degrees of isolation. The maximum level of isolation may not be the most desirable as there may be a high performance penalty to pay.

7.4 Transaction Demarcation

Transactions define how and when data is committed to a database. They are indispensable in grouping persistence logic together, ensuring that all methods complete successfully or that the database is rolled back to its previous state. For most operations, you also need to be concerned with transactional details, ensuring that transactions are started at the beginning of an operation and are either committed or rolled back when the operation completes. JEE frameworks enable these features through three core concepts:

1. Platform transaction management refers to the platforms abstraction for handling commits and rollbacks.
2. Declarative transaction management allows developers to specify the transactional requirements for a particular method through metadata or configuration.
3. Programmatic transaction management explicitly controls the transaction through code.

7.4.1 Platform Transaction Management

We are using springframework in this course and Spring offers several **TransactionManager** implementations, each of which fills the role of managing transactions. **TransactionManager** instances typically extend the **AbstractPlatformTransactionManager** class, which in turn implements the **PlatformTransactionManager** interface. These classes form the foundation of Springs transactional support, and provide the knowhow to access, initiate, rollback, and commit transactions. The interface looks like this:

```
public interface PlatformTransactionManager {
    TransactionStatus getTransaction(TransactionDefinition definition)
            throws TransactionException;

    void commit(TransactionStatus status) throws TransactionException;

    void rollback(TransactionStatus status) throws TransactionException;
}
```

Listing 7.1: Transaction Interface

7.4.2 Programmatic Transaction Management

To demonstrate how programmatic transactions work, we will rework the **EntityService** example to use programmatic transactions, as follows;

```
 public class EntityServiceImpl implements EntityService {
    private TransactionTemplate transactionTemplate;

    public ArtEntityServiceImpl(PlatformTransactionManager transactionManager) {
        this.transactionTemplate = new TransactionTemplate(transactionManager);
    }

    public Object saveEntity(Entity entity) {
        return transactionTemplate.execute(new TransactionCallback() {
            public Object doInTransactionWithoutResult(TransactionStatus status) {
                    try {
                        this.getEntityDao().saveEntity(entity);
                    } catch (ImageErrorException e) {
                        status.setRollbackOnly();
                    }
                    return;
                }
            });
    }
}
```

Listing 7.2: Transaction Interface

In this snippet, we rely on constructor injection to provide a reference to our JpaTransactionManager (which is an implementation of the PlatformTransactionManager interface). Using transactionManager, we create an instance of TransactionTemplate, which we use to wrap our persistence behavior within the scope of a transaction.

The usage of the TransactionTemplate should look very familiar to you. This is a common Spring idiom, and works in a similar fashion to the HibernateTemplate we use within our DAO classes. The key difference here is that we are using the TransactionTemplate to handle the boilerplate process of transactions, rather than database connection setup and closing.

To wrap our persistence code within a transaction, we call the execute method on our transactionTemplate property, passing in an anonymous implementation of TransactionCallback as a parameter. In the example, our service method does not return a value, so we implement the method doInTransactionWithoutResult. However, if we needed to return a value from our transaction, we would instead use doInTransaction.

7.4.3 Declarative Transaction Management

Declarative programming employs metadata to define the requirements for a particular set of application logic, rather than coding the steps that define this behavior directly. Typically, you use declarative programming within the context of a framework, which is designed to analyze the metadata in order to tailor its behavior accordingly. Using declarative transaction management, therefore, implies that you define the rules or attributes that compose your transactions behavior, rather than interspersing this logic directly in your code.

Using the **@Transactional** annotation, you can set some transactional behavior and attributes. Propagation defines the transactional behavior for the specified method. This setting determines whether a new transaction should always be created, whether a nested transaction should be created, or even if no transaction should be created at all. Here are the Propagation values you can use.

When using Container-managed transactions we need to decide where a transaction starts and where it ends. Container-managed demarcation policies are defined by transaction attributes. There are six transaction attributes:

1. SUPPORTS
2. NOT_SUPPORTED
3. REQUIRED

4. REQUIRES_NEW
5. MANDATORY
6. NEVER

These can be set on the session bean's class or method. A Java application method which is not in a transaction may invoke a method. This method execution may span a transaction, so the method start and end will demarcate the transaction. In another scenario a method which is in a transaction may invoke another method. Do we start a new transaction for this second method, or is the second method run within the first method's transaction? The transaction attributes cater for these scenarios.

SUPPORTS

If the caller is in a transaction the annotated method will execute in that transaction. If the caller is not in a transaction, no transaction is created and the method executes outside a transaction context.

NOT_SUPPORTED

If the caller is in a transaction, then that transaction is suspended. No transaction is created and the annotated method executes outside a transaction context. If the caller is not in a transaction, no transaction is created and the annotated method executes outside a transaction context.

REQUIRED

The annotated method is required to execute in a transaction but not necessarily a new transaction. If a transaction is already active then the method will execute in that transaction. If no transaction is active, a new transaction is started. This transaction attribute is the default.

REQUIRES_NEW

In this case annotated methods must start in a new transaction. If the calling method is in a transaction, then that transaction is suspended. The new transaction is either committed or rolled back; the original transaction is then resumed.

MANDATORY

The annotated method must start in the caller's transaction. If the caller is not in a transaction, an exception is thrown.

NEVER

The annotated method must never be invoked in a transaction context. If the caller is in a transaction, an exception is thrown. If the caller is not in a transaction, no transaction is created and the annotated method executes outside a transaction context.

7.4.3.1 Examples of Transaction Attributes

First we need to configure the XML in spring

```
 <tx:annotation-driven />
    <context:annotation-config />

<!-- Turn on @Autowired, @PostConstruct etc support -->
    <bean class="org.springframework.beans.factory.annotation.↩
     AutowiredAnnotationBeanPostProcessor" />
    <bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor" ↩
     />


    <context:property-placeholder location="classpath:database.properties"/>


    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="↩
     close">
        <property name="driverClassName" value="${database.driverClassName}"/>
        <property name="url" value="${database.url}"/>
        <property name="username" value="${database.username}"/>
        <property name="password" value="${database.password}"/>
        <property name="initialSize" value="5" />
        <property name="maxActive" value="10" />
        <property name="defaultAutoCommit" value="true"/>
        <property name="defaultTransactionIsolation" value="2"/>
    </bean>

    <bean id="entityManagerFactory" class="org.springframework.orm.jpa.↩
     LocalContainerEntityManagerFactoryBean">
        <property name="persistenceUnitName" value="universityPU" />
        <property name="dataSource" ref="dataSource" />
        <property name="jpaVendorAdapter">
            <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
                <property name="databasePlatform" value="${database.platform}" />
                <property name="showSql" value="true" />
                <property name="generateDdl" value="true" />
            </bean>
        </property>
    </bean>

    <bean id="transactionManager" class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
    </bean>
```

Listing 7.3: XML Setup

And the code using this configuration file

```
class EntityServiceImpl implements EntityService {
    @Transactional(rollbackFor = InvalidImageExeption.class,
                   readOnly = false,
                   timeout = 30,
                   propagation = Propagation.SUPPORTS,
                   isolation = Isolation.DEFAULT)
    public void saveEntity(Entity entity) throws InvalidImageException {
        this.getArtEntityDao().saveEntity(entity);
    }
}
```

Listing 7.4: Sample Code

7.5 Concurrency and Database Locking

In this section we cover concurrency and database locking topics. Isolation levels deal with concurrency issues that arise when two or more transactions simultaneously operate on the same data. We discuss isolation levels in the next subsection. The "Lost Update Problem" subsection describes a problem that can occur when transactions concurrently update the same data. The lost update problem is resolved by one of two strategies: pessimistic locking and optimistic locking.

7.5.1 Isolation Levels

We mentioned earlier that isolation was the third of the ACID properties of transactions. Isolation levels deal with concurrency issues that arise when two or more transactions are simultaneously operating on the same data. There are three problems that can occur with concurrent transactions: the dirty read, unrepeatable read, and phantom read problems. Each isolation level successively deals with each of these problems. The strictest isolation level, serializable, deals with all three problems. However, this comes at a high performance cost and a less strict isolation level may be applied in practice.

The four isolation levels that youll encounter in practice, listed from least isolated to most isolated, are **Read Uncommitted**, **Read Committed**, **Repeatable Read**, and **Serializable**. These isolation levels also have an impact on concurrency. The least stringent isolation level allows for the highest number of concurrent database operations, while the most stringent are all but guaranteed to slow down your systems. Table below highlights the ramifications of each isolation level including a demonstration of the correlation between isolation level and concurrency.

In order to explain the side effects, consider the following scenario in in a sample gallery application:

1. Paul opens a database transaction, T1, and SELECTs everything from the Entity table.
2. Brian initiates a separate transaction, T2, to DELETE a piece of data from the Entity table.
3. Brian, still in his same T2 transaction, UPDATEs a record in the Entity table, correcting a typo.
4. Paul, still in his same T1 transaction, SELECTs all pieces of data in the Entity table a second time.
5. Brians transaction, T2, COMMITs.
6. Mary initiates a new transaction, T3, and INSERTs a new piece of data to the Entity table.
7. Paul, still in his same T1 transaction, SELECTs all pieces of data in the Entity table a third time.
8. Marys T3 transaction COMMITs.
9. Paul, still in his same T1 transaction, SELECTs all pieces of data in the Entity table a fourth time.
10. Pauls transaction, T1, finally COMMITs.

What should Paul see in step four? What about steps seven and nine? Your database vendor will have default behaviors defined, but its important to know that you have absolute control over the outcome by choosing the isolation level you prefer for your transactions.

The dirty read problem can be described by the following scenario:

An account entity initially has a balance of 100.

- Transaction 1 updates the balance to 200.
- Transaction 2 reads new balance of 200.
- Transaction 1 rolls back restoring the balance to the original value of 100.
- Transaction 2 now has an incorrect, or dirty, value of 200.

The unrepeatable read problem can be described by the following scenario:

- Transaction 1 reads account id and balance for all accounts with balance less than 150.
- Transaction 2 updates an account entity changing its balance from 100 to 200.
- Transaction 1 reads the data again including detailed information such as account name

An account which appeared in the initial summary read is not present in the second detailed read.

This problem only occurs where there are multiple reads in same transaction. In many cases the reads will be in separate transactions so this problem is fairly rare.

The phantom read problem can be described by the following scenario:

- Transaction 1 reads account ID and balance for all accounts with balance less than 150.
- Transaction 2 adds a new account entity with a balance of 100.
- Transaction 1 reads the data again including detailed information such as account name

A new account now appears in the second detailed read which was not present in the initial summary read. A phantom read is similar to the unrepeatable read, except that data is inserted rather than updated by the second transaction.

There are four transaction isolation levels:

1. **READ UNCOMMITTED**

   This is the lowest level of isolation; it is barely a transaction at all because it allows one transaction to see data modified by other uncommitted transactions.

   If we employ read uncommitted for our scenario, there will be no transactional isolation whatsoever. Consequently, Pauls first three identical queries will all return different results.

If Marys commit in step eight succeeds, Pauls fourth query will return the same results as his third query.

At step four, Paul sees Brians typo correction (SQL UPDATE) as well as the removal he performed (SQL DELETE) before Brians transaction commits.

This third side effect is commonly referred to as a **dirty read** because Paul is reading in tentative data. If Brians commit fails at step five, forcing his transaction to roll back, the data Paul is looking at will be rendered completely inaccurate.

Reading in Marys insert at step seven, prior to her commit, is also representative of a dirty read because that too represents tentative data.

Choosing the read uncommitted isolation level exposes you to all three of the possible side effects. Intuitively, this represents a strategy that is not ideal.

However, there is an advatnage with the read uncommitted isolation level. Because this isolation level offers the highest degree of concurrency, one can expect each of Paul, Brian, and Marys SQL operations to be incredibly fast.

You might adopt this isolation level when you need to emphasize speed and youre confident that your application can cope with the side effects.

2. **READ COMMITTED**

   This is the default isolation level used by most Relational Database vendors, including Oracle and PostgreSQL. It ensures that other transactions are not able to read data that has not been committed by other transactions.A transaction may read only data that has been committed in the database.

   However, the data that was read by one transaction can be updated by other transactions.

   When choosing read committed, Paul will see any changes made by Brian or Mary after their respective transactions have completed and been committed. This provides some data consistency while still delivering high concurrency.

   This isolation level suffers from what is known as a **PHAMTOM READ**. A **Phantom READ** is a side effectwhere newly inserted and committed rows are visible to query that werent visible earlier within a SINGLE transaction.

   For example, in our scenario above, Pauls query at step nine will return a new record that wasnt visible earlier in his transaction.

   When choosing read committed, Paul is also exposed to a second type of side effect; a **NONREPEATABLE READ**.

   A nonrepeatable read occurs when rereads of the same row return different data within the same transaction. This becomes possible after Brians update and delete are committed in step five. These row level modifications become visible to Paul in step seven, even though Paul read these two rows earlier and hes still in the context of his first and only transaction, T1.

   When in doubt, choose the read committed.

3. **REPEATABLE READ**

   This is the isolation level that ensures that once you select data, you can select at least the same set again. However, if other transactions insert new data, you can still select the newly inserted data.

Relaxing isolation a bit by employing the repeatable read isolation level in our scenario would allow Paul to see any inserts that are committed, but not updates or deletes.

In order to guarantee that rereads of the same row stay consistent, the underlying database will ordinarily implement either row-level, shared read locks or multiversioning.

Under this isolation level setting, Paul would not see Brians update or delete at any point in the scenario. However, Paul will see Marys insert at step nine after she has committed her transaction. This side effectwhere newly inserted and committed rows are visible to Pauls query (step nine) that werent visible earlier (steps four and seven) within a single transaction (T1)is known as a phantom read.

4. **SERIALIZABLE**

   The most expensive and reliable isolation level; all transactions are treated as if they were executed one after another.

   Depite that, this is the easiest isolation level to understand is serializable, which mandates complete isolation.

   If we choose serializable as our isolation level, Paul will never see any of Brians or Marys changes until Paul begins a new transaction.

   From Pauls perspective, the database remains completely consistent and there are no side effects; Paul will see the same results for his query all four times because they all take place within a single transaction that is insulated from any other modifications.

   That sounds pretty ideal, right? So what more is there to talk about?

   Unfortunately, there is a lot of overhead associated with this setting. Using serializable vastly reduces the number of concurrent operations that may occur and can result in nasty performance problems involving database locks.

   As such, the serializable isolation level should be used sparingly, when the use case really requires absolute consistency and its acceptable to risk the chance that concurrent transactions may be forced to abort with an error.

The Read Uncommitted level will allow transactions to read uncommitted data written by a concurrent transaction and so allow dirty reads. Unrepeatable reads and phantom reads may also occur with this level.

The Read Committed level will prevent transactions from reading uncommitted data written by a concurrent transaction.

Dirty reads are thus prevented, however unrepeatable reads and phantom reads may still occur.

The Repeatable Read level ensures that whenever committed data is read from the database, subsequent reads of the same data in the same transaction will always have the same values as the first read. So the unrepeatable read problem is prevented. Phantom reads can still occur.

The Serializable level is the strictest isolation level and prevents dirty reads, unrepeatable reads and phantom reads. The table below summarizes the effect of isolation levels.

Isolation Level	**Dirty Read**	**Unrepeatable Read**	**Phantom Read**
Read Uncommitted	YES	YES	YES
Read Committed	NO	YES	YES
Repeatable Read	NO	NO	YES
Serializable	NO	NO	NO

Chapter 8

Messaging and Integration

8.1 Chapter Objectives

1. Explain possible approaches for communicating with an external system from an enterprise system given an outline description of those systems and outline the benefits and drawbacks of each approach.
2. Explain and demonstrate typical uses of Web Services and XML over HTTP as mechanisms to integrate distinct software components.
3. Explain what messaging is and demonstrate how it is used to integrate distinct software components as part of an overall enterprise application.

8.2 Introduction

In this chapter, we address an aspect that is core to almost every application in existence: how to send data to,and receive data from, another system or application. In the real world of designing, building,and managing applications, this is often a task of overriding importance, especially for large businesses with pre-existing systems.

Interesting applications rarely live in isolation. Any application can be made better by integrating it with other applications.

All integration solutions have to deal with a few fundamental challenges:

1. **Networks are unreliable.**

   Integration solutions have to transport data from one computer to another across networks. Compared to a process running on a single computer, distributed computing has to be prepared to deal with a much larger set of possible problems. Often, two systems to be integrated are separated by several kilometres , and data between them has to travel through cables, LAN segments, routers, switches, public networks, and satellite links. Each step can cause delays or interruptions.

2. **Networks are slow.**

   Sending data across a network is multiple orders of magnitude slower than making a local method call. Designing a widely distributed solution the same way you would approach a single application could have disastrous performance implications.

3. **Any two applications are different.**

   Integration solutions need to transmit information between systems that use different programming languages, operating platforms, and data formats. An integration solution must be able to interface with all these different technologies.

4. **Change is inevitable.**

   Applications change over time. An integration solution has to keep pace with changes in the applications it connects. Integration solutions can easily get effected by changes, if one system changes, all other systems may be affected. An integration solution needs to minimize the dependencies from one system to another by using loose coupling between applications.

Over time, developers have overcome these challenges with four main approaches:

1. **File Transfer**

   One application writes a file that another later reads. The applications need to agree on the filename and location, the format of the file, the timing of when it will be written and read, and who will delete the file.

2. **Shared Database**

   Multiple applications share the same database schema, located in a single physical database. Because there is no duplicate data storage, no data has to be transferred from one application to the other.

3. **Remote Procedure Invocation**

   One application exposes some of its functionality so that it can be accessed remotely by other applications as a remote procedure. The communication occurs in real time and synchronously.

4. **Messaging**

   One application publishes a message to a common message channel. Other applications can read the message from the channel at a later time. The applications must agree on a channel as well as on the format of the message. The communication is asynchronous.

8.3 Application Integration Criteria

The fundamental criterion is whether to use application integration at all. If you can develop a single, standalone application that doesn't need to collaborate with any other applications, you can avoid the whole integration business entirely. Realistically, though, even a simple enterprise has multiple applications that need to work together to provide a unified experience.

The following are some other main decision criteria.

- **Application coupling** Integrated applications should minimize their dependencies on each other so that each can evolve without causing problems to the others. Tightly coupled applications make numerous assumptions about how the other applications work; when the applications change and break those assumptions, the integration between them breaks. Therefore, the interfaces for integrating applications should be specific enough to implement useful functionality but general enough to allow the implementation to change as needed.

- **Intrusiveness**

  When integrating an application into an enterprise, developers should strive to minimize both changes to the application and the amount of integration code needed. Yet, changes and new code are often necessary to provide good integration functionality, and the approaches with the least impact on the application may not provide the best integration into the enterprise.

- **Technology selection**

  Different integration techniques require varying amounts of specialized software and hardware. Such tools can be expensive, can lead to vendor lock-in, and can increase the learning curve for developers. On the other hand, creating an integration solution from scratch usually results in more effort than originally intended and can mean reinventing the wheel.

- **Data format**

  Integrated applications must agree on the format of the data they exchange. Changing existing applications to use a unified data format may be difficult or impossible. Alternatively, an intermediate translator can unify applications that insist on different data formats. A related issue is data format evolution and extensibility, how the format can change over time and how that change will affect the applications.

- **Data timeliness**

  Integration should minimize the length of time between when one application decides to share some data and other applications have that data. This can be accomplished by exchanging data frequently and in small chunks. However, chunking a large set of data into small pieces may introduce inefficiencies. Latency in data sharing must be factored into the integration design. Ideally, receiver applications should be informed as soon as shared data is ready for consumption. The longer sharing takes, the greater the opportunity for applications to get out of sync and the more complex integration can become.

- **Data or functionality**

  Many integration solutions allow applications to share not only data but functionality as well, because sharing of functionality can provider better abstraction between the applications. Even though invoking functionality in a remote application may seem the same as invoking local functionality, it works quite differently, with significant consequences for how well the integration works.

- **Remote Communication**

  Computer processing is typically synchronous, that is, a procedure waits while its subprocedure executes. However, calling a remote subprocedure is much slower than a local one so that a procedure may not want to wait for the subprocedure to complete; instead, it may want to invoke the subprocedure asynchronously, that is, starting the subprocedure but continuing with its own processing simultaneously. Asynchronicity can make for a much more efficient solution, but such a solution is also more complex to design, develop, and debug.

- **Reliability**

  Remote connections are not only slow, but they are much less reliable than a local function call. When a procedure calls a subprocedure inside a single application, it's a given that the subprocedure is available. This is not necessarily true when communicating remotely; the remote application may not even be running or the network may be temporarily unavailable. Reliable, asynchronous communication enables the source application to go on to other work, confident that the remote application will act sometime later.

While the four styles or approaches below solve essentially the same problem, each style has its distinct advantages and disadvantages. In fact, applications may integrate using multiple styles such that each point of integration takes advantage of the style that suits it best.

8.3.1 File Transfer

Have each application produce files of shared data for others to consume and consume files that others have produced.

8.3.1.1 Advantages

1. No extra tools or integration packages are needed
2. Integration is simple
3. Integrators need no knowledge of the internals of an application

8.3.1.2 Disadantages

- One of the most obvious issues with File Transfer is that updates tend to occur infrequently, and as a result systems can get out of synchronization
- It can lack timeliness, yet timeliness of integration is often critical.
- Also may not enforce data format sufficiently

8.3.2 Shared Database

Have the applications store the data they wish to share in a common database.

8.3.2.1 Advantages

- If a family of integrated applications all rely on the same database, then you can be pretty sure that they are always consistent all of the time.
- Shared Database is made much easier by the widespread use of SQL-based relational databases.
- Since every application is using the same database, this forces out problems in semantic dissonance.

8.3.2.2 Disadantages

- One of the biggest difficulties with Shared Database is coming up with a suitable design for the shared database.
- Another, harder limit to Shared Database is external packages. Most packaged applications won't work with a schema other than their own.
- Multiple applications using a Shared Database to frequently read and modify the same data can turn the database into a performance bottleneck and can cause deadlocks as each application locks others out of the data.

8.3.3 Remote Procedure Invocation

Have each application expose some of its procedures so that they can be invoked remotely, and have applications invoke those to initiate behavior and exchange data.Develop each application as a large-scale object or component with encapsulated data. Provide an interface to allow other applications to interact with the running application.

8.3.3.1 Advantages

- Remote Procedure Invocation applies the principle of encapsulation to integrating applications.
- The fact that there are methods that wrap the data makes it easier to deal with semantic dissonance.
- Since software developers are used to procedure calls, Remote Procedure Invocation fits in nicely with what they are already used to.

8.3.3.2 Disadantages

- There are big differences in performance and reliability between remote and local procedure calls. If people don't understand these, then Remote Procedure Invocation can lead to slow and unreliable systems
- Although encapsulation helps reduce the coupling of the applications by eliminating a large shared data structure, the applications are still fairly tightly coupled together. The remote calls that each system supports tend to tie the different systems into a growing knot.

8.3.4 Messaging

Use Messaging to transfer packets of data frequently, immediately, reliably, and asynchronously, using customizable formats. Have each application connect to a common messaging system, and exchange data and invoke behavior using messages.

8.3.4.1 Advantages

- Asynchronous messaging is fundamentally a pragmatic reaction to the problems of distributed systems. Sending a message does not require both systems to be up and ready at the same time.
- Furthermore, thinking about the communication in an asynchronous manner forces developers to recognize that working with a remote application is slower, which encourages design of components with high cohesion (lots of work locally) and low adhesion (selective work remotely).

8.3.4.2 Disadantages

- The high frequency of messages in Messaging reduces many of the inconsistency problems that bedevil File Transfer, but it doesn't remove them entirely. There are still going to be some lag problems with systems not being updated quite simultaneously.
- Asynchronous design is not the way most software people are taught, and as a result there are many different rules and techniques in place.

  Complex to set up

8.4 Web Services

Before we look at the concept behind web services we need to understand the core technology standards that make up web services. Covering all the concepts and standards associated with web services is a vast topic in itself. In this section we attempt to cover the relevant web service standards and information used in the context of this course to get you acquainted with the technologies for developing web services . Some of the concepts will be explained in greater detail during the course.

8.4.1 XML

XML stands for Extensible Markup Language. XML is a markup language that specifies or describes the format of the data to be exchanged between two parties. The data is significantly structured as tags or elements in a hierarchical order. A user can create his/her own tag to represent structured information. XML has become the de facto standard for representing structured information

8.4.2 SOAP (Simple Object Access Protocol)

SOAP is a protocol for exchanging XML-based messages over a network, typically using HTTP protocol. The SOAP message format is comprised of a SOAP Envelope which encloses all request information. The SOAP Envelope, in turn, is then made up of optional headers and a body. The headers optionally contain context related information, such as security or transaction, while the body contains actual payload or application data.

8.4.3 WSDL (Web Services Description language)

WSDL is a standard-based XML language used to describe web services. Under WSDL, a web service is described as a set of communication endpoints that are capable of exchanging messages. These communication endpoints are called ports. An endpoint is comprised of two parts:

- The first part is the abstract definitions of operations (similar to methods in Java) provided by the services and messages (input and output parameter types for methods) which are needed to invoke the service. The set of abstract operation definitions is referred to as port type.
- The second part is the concrete binding of those abstract definitions of operations to concrete network protocol, where the service is located, and message format for the service.

The WSDL binding describes how the service is bound to a messaging protocol,particularly the SOAP messaging protocol. Typically, the WSDL files would be created using the tool provided by the web service framework.

8.4.4 REST (Representational State Transfer)

REST (Representational State Transfer) is neither a technology nor a standard; it's an architectural stylea set of guidelines for exposing resources over the Web. The REST architecture style is related to a resource, which is a representation identified by a Uniform Resource Indicator (URI), for example, http://cput.co.za/course. The resource can be any piece of information such as a course, book, order, customer, employee, and so on.

The client queries or updates the resource through the URI and, therefore, influences a state change in its representation. All resources share a uniform interface for the transfer of state between client and resource.

The World Wide Web is a classic example built on the REST architecture style. As implemented on the World Wide Web, URIs identify the resources (http://amazon.com/mybook), and HTTP is the protocol by which resources are accessed. HTTP provides a uniform interface and set of methods to manipulate the resource. A client program, like a web browser, can access, update, add, or remove a Web resource through URI using various HTTP methods, like GET and POST, thereby changing its representational state. More later on this.

8.4.5 Service Registry

Service Registry provides a mechanism to look up web services. Traditionally, there was UDDI specification that defined the standards on registering and discovering a web service, but it lacked enterprise-wide adoption. Enterprises started shipping their own version of Service Registry, providing enterprise capabilities like service versioning, service classifications, and life cycle management.

8.5 Introduction to Web Services

There are many different definitions available for a web service. The World Wide Web Consortium (W3C) defines a web service as follows:

> A Web service is a software system identified by a URI whose public interfaces and bindings are defined and described using XML (specifically WSDL). Its definition can be discovered by other software systems. These systems may then interact with the web service in a manner prescribed by its definition, using XML-based messages conveyed by Internet protocols.

Simply, put web service is a software component that provides a business function as a service over the web that can be accessed through a URL. Web services are next generation web applications, modules, or components that can be thought of as a service provided over the web. Traditionally, we had static HTML pages as web content, which evolved into more dynamic full featured web applications providing business functionality and rich GUI features to the end user. A web service component is one step ahead of this web paradigm and provides only business service, usually in the form of raw XML data that can be digested by virtually all client systems. The GUI and business functionality are well separated. A web service can be thought of as a self contained, self describing, modular application that can be published, located, and invoked across the web.

The greatest benefit that web services provide is interoperability. Web services can be ported on any platform and can be written in different programming languages. Similarly, the client accessing the web service can be an application written in a different language and running on a different platform than that of a service itself.

Web services have become a standard way to achieve interoperability between systems. There are two main approaches for developing web services; one is by using the Simple Object Access Protocol (SOAP) and the other is by using the Representational State Transfer (REST) architecture style.

8.6 Approaches for web service development

Two of the most widely used approaches for developing web services are SOAP (Simple Object Access Protocol) and the REST (Representational State Transfer) architecture style. SOAP uses HTTP as a transport protocol while REST uses HTTP as an application protocol.

A web service involves three types of rolesa service consumer, a service provider, and an optional service registry.

The service providers furnish the services over the web and respond to web service requests. The service consumer consumes the services offered by the service provider. In SOAP-based web services, the service provider publishes the contract (WSDL file) of the service over the web where a consumer can access it directly or by looking up a service registry. The service consumer usually generates a web service client code from a WSDL file using the tools offered by the web service framework to interact with the web service. In the next chapter you will look at how to create web service clients from a WSDL file.

With RESTful Web Services there is no formal contract between the service provider and the service consumer. The service requestor needs to know the format of the message, for instance, XML or JSON (Java Script Object Notation), and operations supported by the service provider. The service provider exposes the set of operations using standard HTTP methods like GET or POST. The service requestor invokes one of the methods defined for the resources using the URI over the HTTP protocol.

The choice of adopting SOAP rather than REST depends on your application's requirements. If your requirement consists of transmitting and receiving simple XML messages, then you would probably go with RESTful Web Services. However, if your requirement consists of various contracts to be defined and negotiated between the provider and consumer such as using a WSDL (Web Service Description Language) file and adhering to various web services specifications (WS Specifications) such as web service security for enterprise adoption, then SOAP-based web services is the right option. If you are developing SOAP-based services, then you also need to be aware of SOAP communication styles.

8.6.1 SOAP Based Web Services

The web service SOAP communication style plays a significant role in communicating SOAP XML messages between the service provider and the service consumer.

There exist two types of SOAP message styles, Document and RPC. The SOAP message styles are defined in a WSDL document as SOAP binding. A SOAP binding can have either an encoded use or a literal use. Encoding as the term implies, the message would be encoded using some format, while literal specifies plain text messages without any encoding logic.

Document style, as the name suggests, deals with XML documents as payloads which adhere to well defined contracts, typically created using XML schema definitions. The XML schema format specifies the contract for business messages being exchanged between web service provider and consumer, which the consumers can call and adhere to. The XML schema defines the request and response message format between the service provider and the service consumer. Document literal style is the preferred way of web service communication for achieving interoperability.

RPC (Remote Procedure Call) style, on the other hand, indicates that the SOAP body contains an XML representation of a method. In order to serialize method parameters into the SOAP message so it can be deserialized back by any web service implementation, the SOAP specification defines a standard set of encoding rules. As RPC is traditionally used in conjunction with SOAP encoding rules, the combination is referred to as RPC/encoded. You also have an RPC/literal communication style model where you don't have any encoding formats, but the messages are still limited to RPC method-based communication, where messages can't be validated as they are not tied to any XML Schema definition. You should probably avoid developing RPC style web services as it has a lot of interoperability issues.

This is only how far in detail the course will go into the details of SAOP based web services. There are tones of tutorials on the internet on how to impliment SOAP based web services using Apache CXF which you already have in your libraries.

8.6.2 REST Based Web Services

REST stands for Representational State Transfer. REST is neither a technology nor a standard; it's an architectural style, a set of guidelines for exposing resources over the web. The REST architecture style is related to a resource, which is a representation identified by a Uniform Resource Indicator (URI). The resource can be any piece of information such as Book, Order, Customer, Employee, and so on. The client queries or updates the resource through the URI by exchanging representations of the resource.

The representations contain actual information in a format such as HTML, XML, or JavaScript Object Notation (JSON) that is accepted by the resource. The client needs to be aware of the representation returned by the client. Usually the client specifies which representations it wants and the server returns the required resource, for instance, the required page with HTML content. All resources share a uniform interface for the transfer of state between client and resource. All the information required to process a request on a resource is contained within the request itself, thereby making the interaction stateless

8.6.2.1 Prnciples of REST

8.6.2.2 Addressable resources

The key abstraction of information and data in REST is a resource, and each resource must be addressable via a URI (Uniform Resource Identifier). Addressability is the idea that every object and resource in your system is reachable through a unique identifier.

In the REST world, addressability is managed through the use of URIs. When you make a request for information in your browser, you are typing in a URI. Each HTTP request must contain the URI of the object you are requesting information from or posting information to. The format of a URI is standardized as follows:

scheme:host:portpath?queryString#fragment

The scheme is the protocol you are using to communicate with. For RESTful web services, it is usually http or https. The host is a DNS name or IP address. It is followed by an optional port, which is numeric. The host and port represent the location of your resource on the network. Following host and port is a path expression. This path expression is a set of text segments delimited by the character. Think of the path expression as a directory list of a file on your machine. Following the path expression is an optional query string. The ? character separates the path from the query string. The query string is a list of parameters represented as name/value pairs.

Each pair is delimited with the & character. Heres an example query string within a URI:

http:/example.com/customers?lastName=Burke&zipcode=02115

8.6.2.3 A uniform, constrained interface

Use a small set of well-defined methods to manipulate your resources. The idea behind it is that you stick to the finite set of operations of the application protocol youre distributing your services upon. This means that you dont have an action parameter in your URI and use only the methods of HTTP for your web services. HTTP has a small, fixed set of operational methods. Each method has a specific purpose and meaning. Lets review them:

- **GET**

  GET is a read-only operation. It is used to query the server for specific information. It is both an idempotent and safe operation. Idempotent means that no matter how many times you apply the operation, the result is always the same. The act of reading an HTML document shouldnt change the document. Safe means that invoking a GET does not change the state of the server at all. This means that, other than request load, the operation will not affect the server.

- **PUT**

  PUT requests that the server store the message body sent with the request under the location provided in the HTTP message. It is usually modeled as an insert or update. It is also idempotent. When using PUT, the client knows the identity of the resource it is creating or updating. It is idempotent because sending the same PUT message more than once has no effect on the underlying service. An analogy is an MS Word document that you are editing. No matter how many times you click the Save button, the file that stores your document will logically be the same document.

- **DELETE**

  DELETE is used to remove resources. It is idempotent as well.

- **POST**

  POST is the only nonidempotent and unsafe operation of HTTP. Each POST method is allowed to modify the service in a unique way. You may or may not send information with the request. You may or may not receive information from the response.

- **HEAD**

  HEAD is exactly like GET except that instead of returning a response body, it returns only a response code and any headers associated with the request.

- **OPTIONS**

  OPTIONS is used to request information about the communication options of the resource you are interested in. It allows the client to determine the capabilities of a server and a resource without triggering any resource action or retrieval.

There are other HTTP methods (like TRACE and CONNECT), but they are unimportant when designing and implementing RESTful web services.

8.6.2.4 Representation-oriented

The third architectural principle of REST is that your services should be representation oriented. Each service is addressable through a specific URI and representations are exchanged between the client and service. With a GET operation, you are receiving a representation of the current state of that resource. A PUT or POST passes a representation of the resource to the server so that the underlying resources state can change.

In a RESTful system, the complexity of the client-server interaction is within the representations being passed back and forth. These representations could be XML, JSON, YAML, or really any format you can come up with.

With HTTP, the representation is the message body of your request or response. An HTTP message body may be in any format the server and client want to exchange. HTTP uses the Content-Type header to tell the client or server what data format it is receiving. The Content-Type header value string is in the Multipurpose Internet Mail Extension (MIME) format.

You interact with services using representations of that service. A resource referenced by one URI can have different formats. Different platforms need different formats. For example, browsers need HTML, JavaScript needs JSON (JavaScript Object Notation), and a Java application may need XML.

8.6.2.5 Communicate statelessly

The fourth RESTful principle I will discuss is the idea of statelessness. Stateless applications are easier to scale. When we talk about statelessness, though, w dont mean that your applications cant have state. In REST, stateless means that there is no client session data stored on the server. The server only records and manages the state of the resources it exposes. If there needs to be session-specific data, it should be held and maintained by the client and transferred to the server with each request as needed. A service layer that does not have to maintain client sessions is a lot easier to scale, as it has to do a lot fewer expensive replications in a clustered environment. Its a lot easier to scale up, because all you have to do is add machines.

8.6.2.6 Hypermedia As The Engine Of Application State (HATEOAS)

The final principle of REST is the idea of using Hypermedia As The Engine Of Application State (HATEOAS). Hypermedia is a document-centric approach with the added support for embedding links to other services and information within that document format. Let your data formats drive state transitions in your applications.

8.7 REST Configuration and Worked Example

In this section we are going to apply RESTful capabilities to the university application that was used in the exam. The complete file will be later loaded on WEBCT.

There are about four solid APIs that have a solid implementation of the REST Specification namely Apache CXF, Jersey, REST Easy and Restlet. This Course will use Apache CXF and you are advised to read the online resource for Apache CXF before going through the worked example. The resource is on this link

http://cxf.apache.org/docs/jax-rs.html

The First Step is to fix the Domain Model by adding annotation by adding the annotations shown in the code in line 2 and 3

```
@Entity
@XmlRootElement(name = "university")
@XmlAccessorType(XmlAccessType.FIELD)
public class University implements Serializable {
    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String universityName;
  .......
```

Next we need to update the web.xml file in the web folder. You modified web.xm fil should look as shown below

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee" xmlns:xsi="http://www.w3.↩
     org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-↩
     app_2_5.xsd">

    <display-name>University</display-name>
    <description>University</description>

 <!-- =========== Loading Log 4J configuration Files ================-->
    <context-param>
        <param-name>log4jConfigLocation</param-name>
        <param-value>/WEB-INF/classes/log4j.properties</param-value>
    </context-param>

     <!-- ===========Loading Application Context Files ================-->
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/classes/com/cput/infrastructure/conf/applicationContext-*.xml↩
     </param-value>
    </context-param>

 <!-- =========== Filter and Mapping  to Enable Lazy Loading of Entity Beans↩
     ================-->
    <filter>
```

```
        <filter-name>Spring OpenEntityManagerInViewFilter</filter-name>
        <filter-class>org.springframework.orm.jpa.support.OpenEntityManagerInViewFilter</↩
     filter-class>
    </filter>

    <filter-mapping>
        <filter-name>Spring OpenEntityManagerInViewFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>


  <!-- =========== Spring COntext Listener Activation ================-->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-↩
     class>
    </listener>

    <!-- ===========  Apache CXF Configuration================-->
    <servlet>
        <servlet-name>CXFServlet</servlet-name>
        <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>CXFServlet</servlet-name>
        <url-pattern>/ws/*</url-pattern>
    </servlet-mapping>

 <!-- =========== Duration  Before Session Times out================-->
    <session-config>
        <session-timeout>30</session-timeout>
    </session-config>

</web-app>
```

Next we create our resources in the files in the web services folder. Please not the package path

The Interface

```
package com.cput.application.webservices.rest.resources.university;

import com.cput.application.webservices.rest.resources.university.util.RequestForm;
import com.cput.application.webservices.rest.resources.university.util.UniversityForm;
import com.cput.model.University;
import java.util.Collection;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.Request;
import javax.ws.rs.core.Response;

/**
 *
 * @author boniface
 */
public interface UniversityResource {

    public abstract Collection<University> getUniversities();

    public abstract University getUniversity(Long id);

    public abstract University getUniversity(@Context Request request,RequestForm form);

    public abstract Response createUniversities(@Context Request request,UniversityForm form↩
     );

     public abstract Response updateUniversity(@Context Request request,University ↩
     university);
}
```

The Implementation

```
package com.cput.application.webservices.rest.resources.university.Impl;

import com.cput.application.exceptions.UniversityNotFoundException;
import com.cput.application.webservices.rest.resources.university.UniversityResource;
import com.cput.application.webservices.rest.resources.university.util.RequestForm;
import com.cput.application.webservices.rest.resources.university.util.UniversityForm;
import com.cput.model.University;
import com.cput.services.UniversityService;
import java.net.URI;
import java.net.URISyntaxException;
import java.util.Collection;
```

```java
import java.util.logging.Level;
import java.util.logging.Logger;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.WebApplicationException;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.Request;
import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.ResponseBuilder;
import javax.ws.rs.core.Response.Status;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

/**
 *
 * @author boniface
 */
@Path("/universityservice")
@Produces("application/xml")
@Service("universityResource")
public class UniversityResourceImpl implements UniversityResource {

    @Autowired
    private UniversityService service;

    /**
     * @return the service
     */
    public UniversityService getService() {
        return service;
    }

    /**
     * @param service the service to set
     */
    public void setService(UniversityService service) {
        this.service = service;
    }

    @GET
    @Path("/universities")
    @Override
    public Collection<University> getUniversities(){
        return service.findAll();
    }

    @GET
    @Path("/university/{id}")
    @Override
    public University getUniversity(@PathParam("id") Long id) {
       University university = service.find(id);
        if (university == null) {
            ResponseBuilder builder = Response.status(Status.BAD_REQUEST);
            builder.type("application/xml");
            builder.entity("<error>University Not Found</error>");
            throw new WebApplicationException(builder.build());
        } else {
            return university;
        }
    }

    @POST
    @Path("universities")
    public Response createUniversities(@Context Request request, UniversityForm form) {
             University u = new University();
             u.setNumberOfCampuses(form.getNumberOfCampuses());
             u.setUniversityName(form.getUniversityName());

        try {
            service.persist(u);
            ResponseBuilder builder = Response.created(new
URI("http://localhost:8084/university/ws/universityservice/universities"));
            return builder.build();
        } catch (URISyntaxException e) {
            throw new RuntimeException(e);
        }

    }

    @POST
    @Path("university")
    public University getUniversity(@Context Request request, RequestForm form) {
        try {
            University university = service.getByPropertyName(form.getPropertyName(), form.↩
     getPropertyValue());
            return university;
        } catch (UniversityNotFoundException ex) {
            ResponseBuilder builder = Response.status(Status.BAD_REQUEST);
            builder.type("application/xml");
            builder.entity("<error>Invalid Code</error>");
            Logger.getLogger(UniversityResourceImpl.class.getName()).log(Level.SEVERE, null,↩
      ex);
            throw new WebApplicationException(builder.build());
        }
    }

    @PUT
    @Path("/university/update")
    public Response updateUniversity(@Context Request request, University university) {

        if (university == null) {
            return Response.status(Status.BAD_REQUEST).build();
        } else {
            service.merge(university);
            return Response.ok(university).build();
        }

    }
}
```

The Utility files

```
package com.cput.application.webservices.rest.resources.university.util;

import java.io.Serializable;
import javax.xml.bind.annotation.XmlRootElement;

/**
 *
 * @author boniface
 */
@XmlRootElement
public class RequestForm implements Serializable{

    private String propertyName;
    private String propertyValue;

    /**
     * @return the propertyName
     */
    public String getPropertyName() {
        return propertyName;
    }

    /**
     * @param propertyName the propertyName to set
     */
    public void setPropertyName(String propertyName) {
        this.propertyName = propertyName;
    }

    /**
     * @return the propertyValue
     */
    public String getPropertyValue() {
        return propertyValue;
    }

    /**
     * @param propertyValue the propertyValue to set
     */
    public void setPropertyValue(String propertyValue) {
        this.propertyValue = propertyValue;
    }
}
```

And

```
package com.cput.application.webservices.rest.resources.university.util;

import java.io.Serializable;
import javax.xml.bind.annotation.XmlRootElement;

/**
 *
 * @author boniface
 */
@XmlRootElement
public class UniversityForm implements Serializable{

    private String universityName;
    private int numberOfCampuses;

    /**
     * @return the universityName
     */
    public String getUniversityName() {
        return universityName;
    }
    /**
     * @param universityName the universityName to set
     */
    public void setUniversityName(String universityName) {
        this.universityName = universityName;
    }

    /**
     * @return the numberOfCampuses
     */
    public int getNumberOfCampuses() {
        return numberOfCampuses;
    }

    /**
     * @param numberOfCampuses the numberOfCampuses to set
     */
    public void setNumberOfCampuses(int numberOfCampuses) {
        this.numberOfCampuses = numberOfCampuses;
    }
}
```

Next we create an xml file that will enable the Spring Framework to load the configurations for Apache CXF. Create a file called **applicationContext-webservices.xml** and put in the contents show below.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:cxf="http://cxf.apache.org/core"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xmlns:jaxrs="http://cxf.apache.org/jaxrs"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans http://www.springframework.org/schema/↩
     beans/spring-beans-2.0.xsd
       http://cxf.apache.org/core
       http://cxf.apache.org/schemas/core.xsd
       http://cxf.apache.org/jaxws
       http://cxf.apache.org/schemas/jaxws.xsd
       http://cxf.apache.org/jaxrs
       http://cxf.apache.org/schemas/jaxrs.xsd"
       default-autowire="byName">

    <!-- Load CXF modules from cxf.jar -->
    <import resource="classpath:META-INF/cxf/cxf.xml"/>
    <import resource="classpath:META-INF/cxf/cxf-servlet.xml"/>
    <import resource="classpath:META-INF/cxf/cxf-extension-jaxrs-binding.xml"/>


    <cxf:bus>
        <cxf:features>
            <cxf:logging/>
        </cxf:features>
    </cxf:bus>
    <jaxrs:server id="resources" address="/">
        <jaxrs:serviceBeans>
            <ref bean="universityResource"/>
        </jaxrs:serviceBeans>
        <jaxrs:extensionMappings>
            <entry key="xml" value="application/xml"/>
        </jaxrs:extensionMappings>
    </jaxrs:server>


</beans>
```

And finally the Sample Test Case File for Testing

```



package com.cput.test.intergrations.webservice.rest;


import com.cput.application.webservices.rest.resources.university.util.UniversityForm;
import com.cput.model.University;
import org.apache.cxf.jaxrs.client.WebClient;
import org.junit.After;
import org.junit.AfterClass;
import org.junit.Assert;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

/**
 *
 * @author boniface
 */
public class UniversityRESTServiceTest {

    public UniversityRESTServiceTest() {
    }

    @BeforeClass
    public static void setUpClass() throws Exception {
        WebClient client = WebClient.create("http://localhost:8084/university/ws/↩
universityservice/");
    }

    @AfterClass
    public static void tearDownClass() throws Exception {
    }

    @Before
    public void setUp() {
    }

    @After
    public void tearDown() {
    }

    //Test requires the Application to be running in the Container
    //
    @Test
    public void readUniversity() {
        WebClient client = WebClient.create("http://localhost:8084/university/ws/↩
universityservice/");
        // You have to Supply the ID that is in the DB /voucher/{ID} for the Test to pass
        University university = client.path("university/1770").accept("application/xml").get↩
(University.class);
        Assert.assertNotNull(university);
    }

    @Test
    public void updateUniversity() {
        WebClient client = WebClient.create("http://localhost:8084/university/ws/↩
universityservice/");
        // You have to Supply the ID that is in the DB /voucher/{ID} for the Test to pass
        University voucher = client.path("PATH/1770").accept("application/xml").get(↩
University.class);

        //Now Update the University
        client.back(true);
        client.path("/PATH/update");
        client.put(voucher);
        client.back(true);
        //Retrieve the Updated Voucher
        University newuniversity = client.path("PATH/1770").accept("application/xml").get(↩
University.class);

    }


    public void printUniversities(){

        //Code Invetory

    }

    // @Test
    public void testCreateUniversities() {
        WebClient client = WebClient.create("http://localhost:8084/hashpay/voucherservice/")↩
;
        client.path("PATH");
        UniversityForm f = new  UniversityForm();
        //Initialise Here
        client.post(f);
        client.back(true);
    }
}
```

There are a number of things worth noting. First this is a web service and it has to run in a web container like Tomcat. To Test it, you need to run your container and trying to access the results through a browser to see the xml output.

As you can see the Test client acceses the application throug a URI. This means that your webserver should be running when you run the clients test. The Test and implementation of the methods is incomplete. Your task is to complete the CRUD implemetation in your application.

8.8 Messaging

As stated at the beginning of this chapter, messaging solves many architectural challenges such as heterogeneous integration, scalability, system bottlenecks, concurrent processing, and overall architecture flexibility and agility.

8.8.0.7 Heterogeneous Integration

The communication and integration of heterogeneous platforms is perhaps the most common use case for messaging. Using messaging you can invoke services from applications and systems that are implemented in completely different platforms. Many open source and commercial messaging systems provide seamless connectivity between Java and other languages and platforms by leveraging an integrated message bridge that converts a message sent using one form to a common internal message format.

Historically, there have been many ways of tackling the issue of heterogeneous systems integration. Some earlier solutions involved the transfer of information through FTP or some other file transfer means, including the classic sneakernet method of carrying a diskette or tape from one machine to another. Using a database to share information between two heterogeneous systems or applications is another common approach that is still widely used today. Remote Procedure Call, or RPC, is yet another way of sharing both data and functionality between disparate systems. While each of these solutions have their advantages and disadvantages, only messaging provides a truly decoupled solution allowing both data and functionality to be shared across applications or subsystems.

More recently, Web Services has emerged as another possible solution for integrating heterogeneous systems. However, lack of reliability for web services make messaging a better integration choice.

8.8.0.8 Reduce System Bottlenecks

System and application bottlenecks occur whenever you have a process that cannot keep up with the rate of requests made to that process. A good example of a system bottleneck is a poorly tuned database where applications and processes wait until database connections are available or database locks free up. At some point the system backs up, response time gets worse, and eventually requests start timing out.

A good analogy of a system bottleneck is pouring water into a funnel. The funnel becomes a bottleneck because it can only allow a certain amount of water to pass through. As the amount of water entering the funnel increases, the funnel eventually overflows because water cannot exit the funnel fast enough to handle the increased flow.

IT systems work in much the same way: some components can only handle a limited number of requests and can quickly become bottlenecks.

Going back to our example, if a single funnel can process one liter of water per minute, but three liters of water are entering the funnel, the funnel will eventually back up and overflow. However, by adding two more funnels to the process, we can now theoretically process three liters of water per minute, thereby keeping up with the demand.

Similarly, within IT systems messaging can be used to reduce or even eliminate system bottlenecks. Rather than have requests backing up one behind the other while a synchronous component is processing them, the requests are sent to a messaging system that distributes the requests to multiple message listener components. In this manner the bottlenecks experienced with a single synchronous point-to-point connection are reduced or in some cases completely eliminated.

8.8.0.9 Increase Scalability

Much in the same way that messaging reduces system bottlenecks, it can also be used to increase the overall scalability and throughput of a system, effectively reducing the response time as well. Scalability in messaging systems is achieved by introducing multiple message receivers that can process different messages concurrently. As messages stack up waiting to be processed, the number of messages in the queue, or what is otherwise known as the queue depth, starts to increase. As the queue depth increases, system response time increases and throughput decreases.

One way to increase the scalability of a system is to add multiple concurrent message listeners to the queue (similar to what we did in the funnel example previously) to process more requests concurrently.

Another way to increase the overall scalability of a system is to make as much of the system asynchronous as possible. Decoupling components in this manner allows for systems to grow horizontally, with hardware resources being the main limiting factor.

However, while this may seem like a silver bullet, the middleware can only be horizontally scaled within practical limits of another major system bottleneckthe database. You can have hundreds or even thousands of message listeners on a single queue providing the ability to process many messages at the same time, but the database may only be able to process a limited number of concurrent requests.

Although there are complicated techniques for addressing the database bottleneck issue, the reality is that there will always be practical limits to how far you can scale the middleware layer.

8.8.0.10 Increase End User Productivity

The use of asynchronous messaging can also increase end user productivity. Consider the case where an end user makes a request to the system from a web-based or desktop user interface that takes several minutes to run. During that time the end user is waiting for the results, unable to do any additional work. By using asynchronous messaging, the end user can make a request to the system and get an immediate response back indicating that the request was accepted.

The end user now continues to do other work on the system while the long running request is executing. Once the request has completed, the end user is notified that the request has been processed and the results are delivered to the end user. By using messaging, the end user is able to get more work done with less wait time, making that end user more productive.

Many front-office systems use this sort of messaging strategy between the front application and the backend systems. This type of messaging-based architecture allows the user to perform other work without having to wait for a response from the system.

The trade-off for this increased flexibility and productivity, however, is added complexity. A good architect will always look for opportunities to make various aspects of a system asynchronous, whether it be between a user interface and a system or between internal components within the system.

8.8.0.11 Architecture Flexibility and Agility

The use of messaging as part of an overall enterprise architecture solution allows for greater architectural flexibility and agility. These qualities are achieved through the use of abstraction and decoupling. With messaging, subsystems, components, and even services can be abstracted to the point where they can be replaced with little or no knowledge by the client components.

Architectural agility is the ability to respond quickly to a constantly changing environment. By using messaging to abstract and decouple components, one can quickly respond to changes in software, hardware, and even business changes.

The ability to swap out one system for another, change a technology platform, or even change a vendor solution without affecting the client applications can be achieved through abstraction using messaging. Through messaging, the message producer, or client component, does not know which programming language or platform the receiving component is written in, where the component or service is located, what the component or service implementation name is, or even the protocol used to access that component or service.

It is by means of these levels of abstraction that we are able to more easily replace components and subsystems, thereby increasing architectural agility.

8.8.1 Messaging Lab Practice

1. Demostrate the setup of the Messaging Middleware
2. Demontrate the sending of a Message to a Queue by a Producer
3. Demonstrate the Consumption of a Message from a Queue by a Consumer

8.8.2 Introduction

Messaging Middleware is the software or system that provides an interface between applications, allowing them to send data back and forth to each other asynchronously.

Data sent by one program can be stored in a queue and then forwarded to the receiving program when it becomes available to process it. Without using a common message transport and queueing system such as this, each application must be responsible for ensuring that the data sent is received properly.

Maintaining communications between different types of applications as they are revised and eventually replaced with newer architectures creates an enormous programming burden in the large enterprise.

In this lab, we are going to use Active MQ as the messaging middleware

8.8.3 Active MQ

ActiveMQ is an open-source enterprise-level messaging server, in the category of message-oriented middleware (MOM) software. It is developed as part of the Apache Software Foundation, and it is JMS (Java Message Service) 1.1 compliant. In short, if you are looking for a JMS-compliant server to integrate into your existing application server or if you're looking for a highly available and highly scalable standalone service bus, then ActiveMQ is a great option.

Before getting into the details of the lab using ActiveMQ, let's first review what constitutes enterprise messaging, what business value it provides to us, and how JMS works.

8.8.4 The Java Message Service API

JMS lets you send messages containing for example a String, array of bytes or a serializable Java object, from one program to another. It doesnt however use a direct connection from program A to program B, instead the message is sent to a JMS provider and put there in a Queue where it waits until the other program receives it.

Producer is a Java program sending a JMS message to a Queue on the JMS Provider. Consumer is another program which receives that message. These two programs can run on separate machines and all they have to know to communicate is the URL of the JMS Provider. The Provider can be for example a Java EE server, like JBoss or Glassfish. In this lab we will use ActiveMQ which is lightweight and easy to use.

The details of JMS API and tutorial can be found on the link below

urlhttp://docs.oracle.com/javaee/1.4/tutorial/doc/JMS.html

8.8.5 The JMS Provider

The provider acts like a broker service to which clients connect to exchange messages. The code listing below will help you start the broker.

```
package nl.hhs.mw.messaging;


import java.net.URI;
import org.apache.activemq.broker.BrokerService;
import org.apache.activemq.broker.TransportConnector;

/**
 *
 * @author boniface
 */
public class MessagingMiddleware {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) throws Exception {
        BrokerService broker = new BrokerService();

        TransportConnector connector = new TransportConnector();
        connector.setUri(new URI("tcp://localhost:61616"));
        //broker.
        broker.addConnector(connector);
        broker.start();
        System.out.println("Messaging Middleware Started");
    }
}
```

The above code will run the broker or MOM server. In your lab project downloaded from blackboard, you can find the file name **MessagingMiddleware** and run it in your netbeans application and observe the messages in the console to make sure the broker started. The messages will be similar to what is seen below

```
JMX consoles can connect to service:jmx:rmi:///jndi/rmi://localhost:1099/jmxrmi
PListStore:activemq-data/localhost/tmp_storage started
Using Persistence Adapter:
KahaDBPersistenceAdapter[
/home/boniface/NetBeansProjects/middlewaremessaging/activemq-data/localhost/Kaha
DB]
KahaDB is version 3
Recovering from the journal ...
Recovery replayed 1 operations from the journal in 0.021 seconds.
ActiveMQ 5.5.1 JMS Message Broker (localhost) is starting
For help or more information please see: http://activemq.apache.org/
Listening for connections at: tcp://localhost.localdomain:61616
Connector tcp://localhost:61616 Started
ActiveMQ JMS Message Broker (localhost, ID:dev-48218-1325075557003-0:1) started
Messaging Middleware Started
```

8.8.6 The JMS Producer

Below is the code producing the message and acting as a producer.

```
 package nl.hhs.mw.messaging;

import javax.jms.*;
import org.apache.activemq.ActiveMQConnection;
import org.apache.activemq.ActiveMQConnectionFactory;


/**
 *
 * @author boniface
 */
public class Producer {

    // URL of the JMS server. DEFAULT_BROKER_URL will just mean
    // that JMS server is on localhost
    private static String url = ActiveMQConnection.DEFAULT_BROKER_URL;

    // Name of the queue we will be sending messages to
    private static String subject = "TESTQUEUE";

    public static void main(String[] args) throws JMSException {
        // Getting JMS connection from the server and starting it
        ConnectionFactory connectionFactory = new
ActiveMQConnectionFactory(url);
        Connection connection = connectionFactory.createConnection();
        connection.start();

        // JMS messages are sent and received using a Session. We will
        // create here a non-transactional session object. If you want
        // to use transactions you should set the first parameter to 'true'
        Session session = connection.createSession(false,
Session.AUTO_ACKNOWLEDGE);

        // Destination represents here our queue 'TESTQUEUE' on the
        // JMS server. You don't have to do anything special on the
        // server to create it, it will be created automatically.
        Destination destination = session.createQueue(subject);

        // MessageProducer is used for sending messages (as opposed
        // to MessageConsumer which is used for receiving them)
        MessageProducer producer = session.createProducer(destination);

        // We will send a small text message
        TextMessage message = session.createTextMessage(" Message 1");

        // Here we are sending the message!
        producer.send(message);
        System.out.println("Sent message '" + message.getText() + "'");

        connection.close();
    }
}
```

Run this file to send a message to the Queue and make sure that the Broker is running.

In the above code, the Connection represents our connection with the JMS Provider ActiveMQ. Destination represents the Queue on the JMS Provider that we will be sending messages to. In this case, we will send it to Queue called TESTQUEUE (it will be automatically created if it didnt exist yet).

What you should note is that there is no mention of who will finally read the message. Actually, the Producer does not know where or who the consumer is! We are just sending messages into queue TESTQUEUE and what happens from there to the sent messages is not of Producers interest any more.

Now lets see how to receive (consume) the sent message. Here is the code for the Consumer class:

8.8.7 The JMS Consumer

Now lets see how to receive (consume) the sent message. Here is the code for the Consumer class:

```
package nl.hhs.mw.messaging;

import javax.jms.*;
import org.apache.activemq.ActiveMQConnection;
import org.apache.activemq.ActiveMQConnectionFactory;

/**
 *
 * @author boniface
 */
public class Consumer {

// URL of the JMS server
    private static String url = ActiveMQConnection.DEFAULT_BROKER_URL;

    // Name of the queue we will receive messages from
    private static String subject = "TESTQUEUE";

    public static void main(String[] args) throws JMSException {
        // Getting JMS connection from the server
        ConnectionFactory connectionFactory = new ActiveMQConnectionFactory(url);
        Connection connection = connectionFactory.createConnection();
        connection.start();

        // Creating session for seding messages
        Session session = connection.createSession(false,Session.AUTO_ACKNOWLEDGE);

        // Getting the queue 'TESTQUEUE'
        Destination destination = session.createQueue(subject);

        // MessageConsumer is used for receiving (consuming) messages
        MessageConsumer consumer = session.createConsumer(destination);

        // Here we receive the message.
        // By default this call is blocking, which means it will wait
        // for a message to arrive on the queue.
```

```
            Message message = consumer.receive();

            // There are many types of Message and TextMessage
            // is just one of them. Producer sent us a TextMessage
            // so we must cast to it to get access to its .getText()
            // method.
            if (message instanceof TextMessage) {
                TextMessage textMessage = (TextMessage) message;
                System.out.println("Received message '" + textMessage.getText() +
"'");
            }
            connection.close();
      }
}
```

Code looks similar to the Producers code before. We have there a MessageConsumer instead of MessageReceiver and then use its .receive() method instead of .send(). You can see also a cast from Message to TextMessage because .receive() method just returns interface Message (TextMessage interface extends Message) and there are no separate methods for receiving just TextMessages.

8.8.8 Chapter Exercise 1

1. Download the Messaging Application from Blackboard and open it in Netbeans. The following is the order in which you run the programs.

   - First Run the MessagingMiddleware and ensure that is is running without an error.
   - Second Run the Producer and see the sent message in the console
   - Next Run the Consumer file and see the message in the console

2. Run the Consumer anagain and record what you see and offer an explanation

3. Then Run the Producer and see what happened to the Consumer that you run in question 2

4. Shutdowm the Middleware Service and try to run the Consumer and Producer and record the results you see.

5. Modify the Consumer and Produce to send different messages of your choice.

8.9 Chapter Exercise 2

Using the application of the domain you have been working on, add REST webservices capability to 15 of your domain classes. All the CRUD methods should be implemented using TEST Cases. Once done, submit your work through a WEBCT link that would be provided.

Chapter 9

Enterprise Web Framework Technologies

9.1 Chapter Objectives

1. State the benefits and drawbacks of adopting a web framework in designing enterprise applications.
2. Explain and demonstrate request/response and event based web frameworks
3. Given a system requirements definition, explain and justify your rationale for choosing request/response based or an event based framework

9.2 MVC Pattern

Most software rendering is based on the famous Modelviewcontroller (MVC) pattern.The MVC pattern is an architectural pattern used in software engineering for user interface interactions. Successful use of the pattern isolates business logic from user interface considerations, resulting in an application where it is easier to modify either the visual appearance of the application or the underlying business rules without affecting the other.

In MVC, the model represents the information (the data) of the application; the view corresponds to elements of the user interface such as text, checkbox items, and so forth; and the controller manages the communication of data and the business rules used to manipulate the data to and from the model.

The MVC pattern is all about separation of concerns. As we mentioned previously, each component has its own role . Separation of concerns is important in the presentation layer because it helps us keep the different components clean. This way, we dont burden the actual view with business logic, navigation logic, and model data. Following this approach keeps everything nicely separated, which makes it easier to maintain and test our application.

The classic implementation of the MVC pattern involves the user triggering an action. This prompts the controller to update the model, which in turn pushes the changes

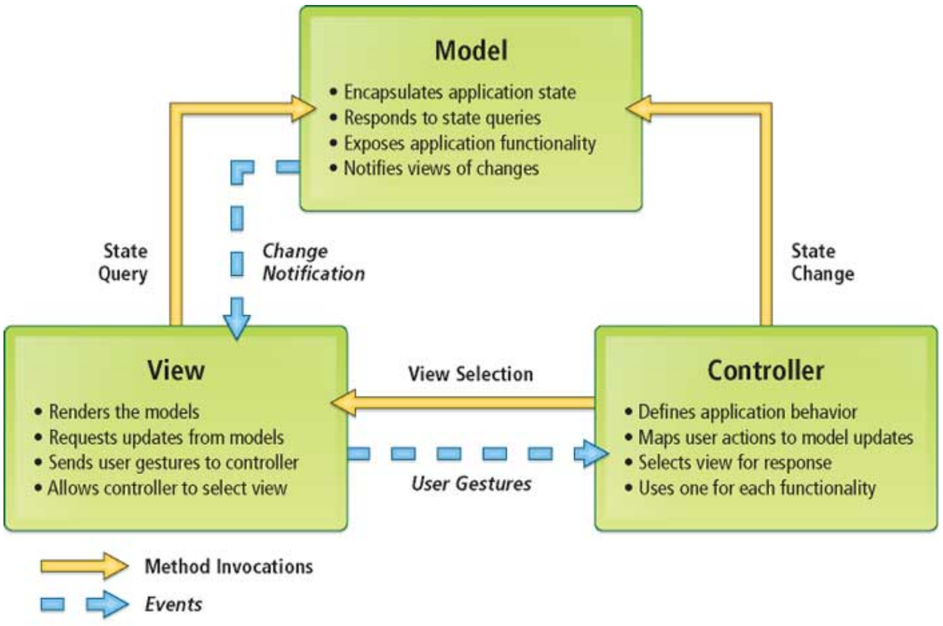

Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• Uses one for each functionality
Method Invocations
Events

FIGURE 9.1: MVC Pattern Model 1

back to the view. The view then updates itself with the updated data from the model. This is the ideal implementation of an MVC pattern, and it works very well in desktop applications based on Swing, for example.

However, this approach is not feasible in a web environment due to the nature of the HTTP protocol. For a web application, the user typically initiates action by issuing a request. This prompts the app to update and render the view, which is sent back to the user.

This means that we need a slightly different approach in a web environment. Instead of pushing the changes to the view, we need to pull the changes from the server. This approach works, but it isnt as straightforward to apply in a web application. The Web (or HTTP) is stateless by design, so keeping a model around can be quite difficult.

For the Web, the MVC pattern is implemented as a Model 2 architecture . The difference between the original pattern Model 1 and the modified pattern is that it incorporates a Front Controller that dispatches the incoming requests to other controllers. These controllers handle the incoming request, return the model, and select the view.

9.3 Web Applications

Web applications today use this MVC design pattern as well. Most web ap- plications are divided into two camps. These are request/response (Push- based) Frameworks and Event based (Pull-based) Frameworks.A web applica- tion Framework is a software Framework that is designed to support the develop- ment of dynamic websites, Web applications and Web services. The framework aims to alleviate the overhead associated with common activities used in Web development.

Most MVC Frameworks follow a push-based architecture. In request/re- sponse Frameworks MVC Frameworks work in terms of whole requests and whole pages. In each

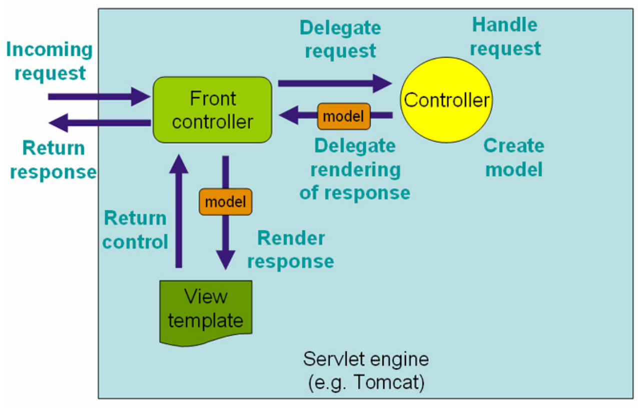


FIGURE 9.2: MVC pattern Model 2

request cycle, the incoming request is mapped to a method on a controller object, which then generates the outgoing response in its entirety, usually by pulling data out of a model to populate a view written in specialized template markup.These frameworks use actions that do the required processing, and then push the data to the view layer to render the results. This keeps the applications flow-of-control simple and clear, but can make code reuse in the controller difficult. Struts, Django, Ruby on Rails and Spring MVC are good examples of this architecture.

An alternative to this is pull-based architecture, sometimes also called component- based or Event based Frameworks. These frameworks start with the view layer, which can then pull results from multiple controllers as needed. In this archi- tecture, multiple controllers can be involved with a single view. In Events based frameworks , the framework is closely patterned after stateful GUI frameworks such as Swing. Event based frameworks applications are trees of components, which use listener delegates to react to HTTP requests against links and forms in the same way that Swing components react to mouse and keystroke events. Each component is backed by its own model, which represents the state of the component.

The framework does not have knowledge of how components interact with their models, which are reated as opaque objects automatically serialized and persisted between requests. More complex models, however, may be made de- tachable and provide hooks to arrange their own storage and restoration at the beginning and end of each request cycle.Struts2, Tapestry, JBoss Seam and Wicket are examples of pull-based architectures.

Both camps have a lot of framweorks to choose from. In this Course we shall use Vaadin Framework, a component based. We need to decide one of the framework based up on our business requirements and it is possible to use both in the same application.

9.3.1 Request-Response Frameworks

Springs web MVC framework is, like many other web MVC frameworks, request- driven, designed around a central servlet that dispatches requests to controllers and offers other functionality facilitating the development of web applications. Springs DispatcherServlet however, does more than just that. It is completely integrated with the Spring IoC container and as such allows you to use every other feature that Spring has.

The request processing workflow of the Spring Web MVC DispatcherServlet is illustrated in the following diagram. The pattern-savvy reader will recognize that the DispatcherServlet is an expression of the Front Controller design pat- tern (this is a pattern that Spring Web MVC shares with many other leading web frameworks).

Based on the HTTP request URL, the DispatcherServlet calls the corre- sponding Controller. The Controllers sets the model and the view to be used for rendering the response and sends back that information to the Dispatch- Servlet(Front Controller -FP). The Front Controller selects the view and pulls

data from the models and populates the templates. A View is rendered and sent as HTTP response.

9.3.1.1 Spring MVC Worked Example

Springs web MVC framework is, like many other web MVC frameworks, request-driven, designed around a central servlet that dispatches requests to controllers and offers other functionality that facilitates the development of web applications. Springs DispatcherServlet is completely integrated with Spring IoC container and allows us to use every other feature of Spring.

Following is the Request process lifecycle of Spring 3.0 MVC:

1. The client sends a request to web container in the form of http request.
2. This incoming request is intercepted by Front controller (DispatcherServlet) and it will then tries to find out appropriate Handler Mappings.
3. With the help of Handler Mappings, the DispatcherServlet will dispatch the request to appropriate Controller.
4. The Controller tries to process the request and returns the Model and View object in form of ModelAndView instance to the Front Controller.
5. The Front Controller then tries to resolve the View (which can be JSP, Freemarker, Velocity etc) by consulting the View Resolver object.
6. The selected view is then rendered back to client.

The entry point of Spring 3.0 MVC is the DispatcherServlet. DispatcherServlet is a normal servlet class which implements HttpServlet base class. Thus we need to configure it in web.xml.

```
<web-app>

    <servlet>
        <servlet-name>spring</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>spring</servlet-name>
        <url-pattern>*.html</url-pattern>
    </servlet-mapping>
</web-app>
```

In above code snippet, we have configure DispatcherServlet in web.xml. Note that we have mapped *.html url pattern with example DispatcherServlet. Thus any url with *.html pattern will call Spring MVC Front controller.

Once the DispatcherServlet is initialized, it will looks for a file names **[servlet-name]-servlet.xml** in WEB-INF folder of web application. In above example, the framework will look for file called **example-servlet.xml**.

Our goal is to create a basic Spring MVC application using latest 3.0 version. There will be an index page which will display a link Say Hello to user. On clicking this link, user will be redirected to another page hello which will display a message Hello World!.

We will need a spring mvc controller class that will process the request and display a Hello World message. For this we will create a package xxx.xxx.xxx.web.controller in the source folder. This package will contain the Controller file.

Create a class called HelloWorldController in the package and copy following content into it.

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class HelloWorldController {

    @RequestMapping("/hello")
    public ModelAndView helloWorld() {

        String message = "Hello World!";
        return new ModelAndView("hello", "message", message);
    }
}
```

Note that we have annotated the HelloWorldController class with **@Controller** and **@RequestMapping("/hello")**. When Spring scans our package, it will recognize this bean as being a Controller bean for processing requests. The **@RequestMapping** annotation tells Spring that this Controller should process all requests beginning with **/hello** in the URL path. That includes **/hello/*** and **/hello.html.**

The helloWorld() method returns ModelAndView object. The ModelAndView object tries to resolve to a view named hello and the data model is being passed back to the browser so we can access the data within the JSP. The logical view name will resolve to "**/WEB-INF/jsp/hello.jsp**". We will discuss this shortly how the logical name hello which is return in ModelAndView object is mapped to path **/WEB-INF/jsp/hello.jsp.**

The ModelAndView object also contains a message with key message and value Hello World!. This is the data that we are passing to our view. Normally this will be a value object in form of java bean that will contain the data to be displayed on our view. Here we are simply passing a string

To display the hello world message we will create a JSP. Note that this JSP is created in folder /WEB-INF/jsp. Create hello.jsp under WEB-INF/jsp directory and copy following content into it.

```
 <html>
<head>
    <title> Hello World</title>
```

```
</head>
<body>
    ${message}
</body>
</html>
```

The above JSP simply display a message using expression $message. Note that the name message is the one which we have set in ModelAndView object with the message string.

Also we will need an index.jsp file which will be the entry point of our application. Create a file index.jsp under WebContent folder in your project and copy following content into it.

```
 <html>
<head>
    <title>Spring 3.0 MVC Series:</title>
</head>
<body>
    <a href="hello.html">Say Hello</a>
</body>
</html>
```

One thing to note here is the name of servlet in ¡servlet-name¿ tag in web.xml. Once the DispatcherServlet is initialized, it will looks for a file name [servlet-name]-servlet.xml in WEB-INF folder of web application. In this example, the framework will look for file called **spring-servlet.xml**.

Create a file spring-servlet.xml in WEB-INF folder and copy following content into it.

File: WEB-INF/spring-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-3.0.xsd">

    <context:component-scan base-package="xxx.xxxx.xxx.controller" />

    <bean id="viewResolver"
        class="org.springframework.web.servlet.view.UrlBasedViewResolver">
        <property name="viewClass" value="org.springframework.web.servlet.view.JstlView" />
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

In the above xml configuration file, we have defined a tag ¡context:component-scan¿. This will allow Spring to load all the components from package net.viralpatel.spring3.controller and all its child packages. This will load our HelloWorldController class. Also we have defined a bean viewResolver. This bean will resolve the view and add prefix string /WEB-INF/jsp/ and suffix .jsp to the view in ModelAndView. Note that in our HelloWorldController class, we have return a ModelAndView object with view name hello. This will be resolved to path **/WEB-INF/jsp/hello.jsp**.

Handling Forms in Spring 3.0 MVC

Our goal is to create basic Contact Manager application. This app will have a form to take contact details from user. For now we will just print the details in logs. We will learn how to capture the form data in Spring 3 MVC

Let us add the contact form to our Spring 3 MVC Hello World application. Open the index.jsp file and change it to following:

```
<jsp:forward page="contacts.html"></jsp:forward>
```

The above code will just redirect the user to contacts.html page.

Create a JSP file that will display Contact form to our users. File: WEB-INF/jsp/contact.jsp

```
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring 3 MVC Series - Contact Manager</title>
</head>
<body>
<h2>Contact Manager</h2>
<form:form method="post" action="addContact.html">

    <table>
    <tr>
        <td><form:label path="firstname">First Name</form:label></td>
        <td><form:input path="firstname" /></td>
    </tr>
    <tr>
        <td><form:label path="lastname">Last Name</form:label></td>
        <td><form:input path="lastname" /></td>
    </tr>
    <tr>
        <td><form:label path="lastname">Email</form:label></td>
        <td><form:input path="email" /></td>
    </tr>
    <tr>
        <td><form:label path="lastname">Telephone</form:label></td>
        <td><form:input path="telephone" /></td>
    </tr>
    <tr>
        <td colspan="2">
            <input type="submit" value="Add Contact"/>
        </td>
    </tr>
</table>

</form:form>
</body>
</html>
```

Here in above JSP, we have displayed a form. Note that the form is getting submitted to addContact.html page.

We will now add the logic in Spring 3 to display the form and fetch the values from it. For that we will create two java files. First the Contact.java which is nothing but the form to display/retrieve data from screen and second the **ContactController.java** which is the spring controller class.

```
public class Contact {
    private String firstname;
    private String lastname;
    private String email;
    private String telephone;

    //.. getter and setter for all above fields.

}
```

The above file is the contact form which holds the data from screen. Note that I havent showed the getter and setter methods.

The the contact Controller

```
import org.springframework.stereotype.Controller;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.servlet.ModelAndView;

@Controller
@SessionAttributes
public class ContactController {

    @RequestMapping(value = "/addContact", method = RequestMethod.POST)
    public String addContact(@ModelAttribute("contact")
                            Contact contact, BindingResult result) {

        System.out.println("First Name:" + contact.getFirstname() +
                    "Last Name:" + contact.getLastname());

        return "redirect:contacts.html";
    }

    @RequestMapping("/contacts")
    public ModelAndView showContacts() {

        return new ModelAndView("contact", "command", new Contact());
    }
}
```

In above controller class, note that we have created two methods with Request Mapping /contacts and /addContact. The method showContacts() will be called when user request for a url contacts.html. This method will render a model with name contact. Note that in the ModelAndView object we have passed a blank Contact object with name command. The spring framework expects an object with name command if you are using in your JSP file.

Also note that in method **addContact()** we have annotated this method with RequestMapping and passed an attribute **method=RequestMethod.POST**. Thus the method will be called only when user generates a POST method request to the url **/addContact.html**. We have annotated the argument Contact with annotation **@ModelAttribute**. This will binds the data from request to the object Contact. In this method we just have printed values of Firstname and Lastname and redirected the view to **cotnacts.html.**

What is i18n and L10n?

In computing, internationalization and localization are means of adapting computer software to different languages and regional differences. Internationalization is the process of designing a software application so that it can be adapted to various languages and regions without engineering changes. Localization is the process of adapting internationalized software for a specific region or language by adding locale-specific components and translating text.

The terms are frequently abbreviated to the numeronyms i18n (where 18 stands for the number of letters between the first i and last n in internationalization) and L10n respectively, due to the length of the words. The capital L in L10n helps to distinguish it from the lowercase i in i18n.

We will add two languages support to our application: English and German. Depending on the locale setting of users browser, the appropriate language will be selected. Also user will be able to select the language from top-right corner of the application.

Create two files messages_en.properties and messages_de.properties in this folder and copy following content into it.

```
label.firstname=First Name
label.lastname=Last Name
label.email=Email
label.telephone=Telephone
label.addcontact=Add Contact

label.menu=Menu
label.title=Contact Manager

label.footer=&copy; hashcode.zm
```

```
label.firstname=Vorname
label.lastname=Familiename
label.email=Email
label.telephone=Telefon
label.addcontact=Addieren Kontakt

label.title=Kontakt Manager
label.menu=Men&#252;

label.footer=&copy; Hashcode.zm
```

Configuring Internationalization (i18n) / Localization (L10n) in Spring MVC

Now we have created message resource properties for our application. We need to declare these files in spring configuration file. We will use class org.springframework.context.support.Reloadable
to define the message resources.

Also, note that we will provide a feature where user will be able to select language for the application. This is implemented by using **org.springframework.web.servlet.i18n.LocaleChangeI**
class. The **LocaleChangeInterceptor** class will intercept any changes in the locale. These changes are then saved in cookies for future request. **org.springframework.web.servlet.i18n.C**
class will be used to store the locale changes in cookies.

Add following code in the **spring-servlet.xml** file.

```
<bean id="messageSource" class="org.springframework.context.support.↩
    ReloadableResourceBundleMessageSource">
    <property name="basename" value="classpath:messages" />
    <property name="defaultEncoding" value="UTF-8"/>
</bean>

<bean id="localeChangeInterceptor" class="org.springframework.web.servlet.i18n.↩
    LocaleChangeInterceptor">
    <property name="paramName" value="lang" />
</bean>

<bean id="localeResolver" class="org.springframework.web.servlet.i18n.CookieLocaleResolver">
    <property name="defaultLocale" value="en"/>
</bean>

<bean id="handlerMapping" class="org.springframework.web.servlet.mvc.annotation.↩
    DefaultAnnotationHandlerMapping">
        <property name="interceptors">
        <ref bean="localeChangeInterceptor" />
    </property>
</bean>
```

Note that in above configuration we have defined basename property in messageSource bean to classpath:messages. By this, spring will identify that the message resource message_ will be used in this application.

```
<%@taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<html>
<head>
    <title>Spring 3 MVC Series - Contact Manager</title>
```

```
</head>
<body>

<form:form method="post" action="addContact.html">

    <table>
    <tr>
        <td><form:label path="firstname"><spring:message code="label.firstname"/></form:↩
     label></td>
        <td><form:input path="firstname" /></td>
    </tr>
    <tr>
        <td><form:label path="lastname"><spring:message code="label.lastname"/></form:label↩
     ></td>
        <td><form:input path="lastname" /></td>
    </tr>
    <tr>
        <td><form:label path="lastname"><spring:message code="label.email"/></form:label></↩
     td>
        <td><form:input path="email" /></td>
    </tr>
    <tr>
        <td><form:label path="lastname"><spring:message code="label.telephone"/></form:label↩
     ></td>
        <td><form:input path="telephone" /></td>
    </tr>
    <tr>
        <td colspan="2">
            <input type="submit" value="<spring:message code="label.addcontact"/>"/>
        </td>
    </tr>
</table>

</form:form>
</body>
</html>
```

Note that in above JSP, we used **¡spring:message¿** tag to display the message from resource bundle.

One thing that we must note here is that in header.jsp file, we have specified two links to select language. The link sets a request parameter ?lang= when user click on this link. Note that spring identifies this request parameter by using LocaleChangeInterceptor interceptor and change the local accordingly. Also note that while configuring LocaleChangeInterceptor in **spring-servlet.xml** file, we have specified property paramName with value lang

```
<property name="paramName" value="lang" />
```

Thus the Spring framework will look for a parameter called lang from request.

9.3.2 Event or Component Based Frameworks

There are many component based Frameworks on the market. We are going to use JSF in this course.

9.3.2.1 JSF Web Framework

JavaServer Faces (JSF) is a component-based framework. For example, if you want to display a table with rows and columns, you do not generate HTML tags for rows and cells in a loop, but you add a table component to a page. By using components, you can think about your user interface at a higher level than raw HTML. You can reuse your own components and use third-party component sets. And you have the option of using a visual development environment in which you can drag and drop components onto a form.

JSF has these parts:

1. A set of prefabricated UI (user interface) components
2. An event-driven programming model
3. A component model that enables third-party developers to supply additional components

For the worked examples che the ebook on WebCT titled The Java EE 6 Tutorial in Chapter 3 -16 covering the Web Tier

9.4 Chapter Exercise

For your application add a request Response and a Event based presetation for one use case to demstrate a CRUD function.

Chapter 10

Security

10.1 Chapter Objectives

1. Explain the client-side security model for enterprise applications
2. Given an architectural system specification, select appropriate locations for implementation of specified security features, and select suitable technologies for implementation of those features.
3. Identify and classify potential threats to a system and describe how a given architecture will address the threats.
4. Describe and apply the commonly used declarative and programmatic methods used to secure applications.

10.2 Introduction

This chapter introduces basic security concepts and security implementation mechanisms.

10.2.1 Security Functions

A properly implemented security mechanism will provide the following functionality:

1. Prevent unauthorized access to application functions and business or personal data (authentication)
2. Hold system users accountable for operations they perform (non-repudiation)
3. Protect a system from service interruptions and other breaches that affect quality of service (QoS)

Ideally, properly implemented security mechanisms will also provide the following functionality:

1. Easy to administer
2. Transparent to system users
3. Interoperable across application and enterprise boundaries

10.2.2 Characteristics of Application Security

Java EE applications consist of components that can contain both protected and unprotected resources. Often, you need to protect resources to ensure that only authorized users have access. Authorization provides controlled access to protected resources. Authorization is based on identification and authentication. Identification is a process that enables recognition of an entity by a system, and authentication is a process that verifies the identity of a user, device, or other entity in a computer system, usually as a prerequisite to allowing access to resources in a system.

Authorization and authentication are not required for an entity to access unprotected resources. Accessing a resource without authentication is referred to as unauthenticated or anonymous access.

The characteristics of application security that, when properly addressed, help to minimize the security threats faced by an enterprise, include the following:

1. **Authentication:** The means by which communicating entities (for example, client and server) prove to one another that they are acting on behalf of specific identities that are authorized for access. This ensures that users are who they say they are.
2. **Authorization, or Access Control:** The means by which interactions with resources are limited to collections of users or programs for the purpose of enforcing integrity, confidentiality, or availability constraints. This ensures that users have permission to perform operations or access data.
3. **Data integrity:** The means used to prove that information has not been modified by a third party (some entity other than the source of the information). For example, a recipient of data sent over an open network must be able to detect and discard messages that were modified after they were sent. This ensures that only authorized users can modify data.
4. **Confidentiality or Data Privacy:** The means used to ensure that information is made available only to users who are authorized to access it. This ensures that only authorized users can view sensitive data.
5. **Non-repudiation:** The means used to prove that a user performed some action such that the user cannot reasonably deny having done so. This ensures that transactions can be proven to have happened.
6. **Quality of Service (QoS):** The means used to provide better service to selected network traffic over various technologies.
7. **Auditing:** The means used to capture a tamper-resistant record of security-related events for the purpose of being able to evaluate the effectiveness of security policies and mechanisms. To enable this, the system maintains a record of transactions and security information.

10.3 Working with Realms, Users, Groups, and Roles

You often need to protect resources to ensure that only authorized users have access. Authorization provides controlled access to protected resources. Authorization is based on identification and authentication. Identification is a process that enables recognition of an entity by a system, and authentication is a process that verifies the identity of a user, device, or other entity in a computer system, usually as a prerequisite to allowing access to resources in a system. These concepts are discussed in more detail in Characteristics of Application Security.

This section discusses setting up users so that they can be correctly identified and either given access to protected resources, or denied access if the user is not authorized to access the protected resources. To authenticate a user, you need to follow these basic steps:

1. The Application Developer writes code to prompt the user for their user name and password.
2. The Application Developer communicates how to set up security for the deployed application by use of a deployment descriptor or metadata annotation.
3. The Server Administrator sets up authorized users and groups on the Enterprise Server.
4. The Application Deployer maps the applications security roles to users, groups, and principals defined on the Enterprise Server.

10.3.1 What Are Realms, Users, Groups, and Roles?

A realm is a security policy domain defined for a web or application server. It is also a string, passed as part of an HTTP request during basic authentication, that defines a protection space. The protected resources on a server can be partitioned into a set of protection spaces, each with its own authentication scheme and/or authorization database containing a collection of users, which may or may not be assigned to a group.

An application will often prompt a user for their user name and password before allowing access to a protected resource. After the user has entered their user name and password, that information is passed to the server, which either authenticates the user and sends the protected resource, or does not authenticate the user, in which case access to the protected resource is denied.

In some applications, authorized users are assigned to roles. In this situation, the role assigned to the user in the application must be mapped to a principal or group defined on the application server.

The following sections provide more information on realms, users, groups, and roles.

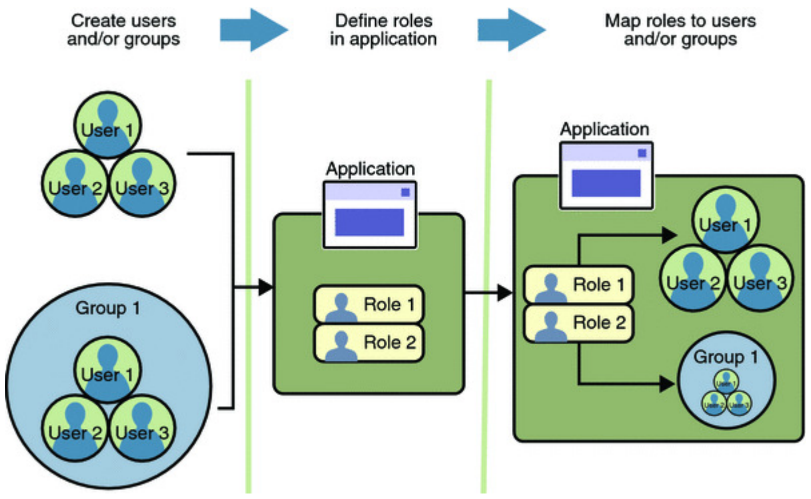

Create users and/or groups
Define roles in application
Map roles to users and/or groups
User 1
User 2
User 3
Group 1
Application
Role 1
Role 2

FIGURE 10.1: Security Role Mapping

10.3.1.1 What Is a Realm?

A realm is a security policy domain defined for a web or application server. It is also a string, passed as part of an HTTP request during basic authentication, that defines a protection space. The protected resources on a server can be partitioned into a set of protection spaces, each with its own authentication scheme and/or authorization database containing a collection of users, which may or may not be assigned to a group.

For a web application, a realm is a complete database of users and groups that identify valid users of a web application (or a set of web applications) and are controlled by the same authentication policy.

The authentication service can govern users in multiple realms.

10.3.1.2 What Is a User?

A user is an individual (or application program) identity that has been defined in the Enterprise Server. In a web application, a user can have a set of roles associated with that identity, which entitles them to access all resources protected by those roles. Users can be associated with a group.

10.3.1.3 What Is a Group?

A group is a set of authenticated users, classified by common traits, defined in the Enterprise Server.

10.3.1.4 What Is a Role?

A role is an abstract name for the permission to access a particular set of resources in an application. A role can be compared to a key that can open a lock. Many people might have a copy of the key. The lock doesnt care who you are, only that you have the right key.

10.3.1.5 Some Other Terminology

The following terminology is also used to describe the security requirements of the Enterprise Applications:

-
- **Principal:** A principal is an entity that can be authenticated by an authentication protocol in a security service that is deployed in an enterprise. A principal is identified using a principal name and authenticated using authentication data.
- **Security policy domain (also known as security domain or realm):** A security policy domain is a scope over which a common security policy is defined and enforced by the security administrator of the security service.
- **Security attributes:** A set of security attributes is associated with every principal. The security attributes have many uses, for example, access to protected resources and auditing of users. Security attributes can be associated with a principal by an authentication protocol.
- **Credential:** A credential contains or references information (security attributes) used to authenticate a principal for the product services. A principal acquires a credential upon authentication, or from another principal that allows its credential to be used.

10.4 Establishing a Secure Connection Using SSL

Secure Socket Layer (SSL) technology is security that is implemented at the transport layer. SSL allows web browsers and web servers to communicate over a secure connection. In this secure connection, the data that is being sent is encrypted before being sent and then is decrypted upon receipt and before processing. Both the browser and the server encrypt all traffic before sending any data. SSL addresses the following important security considerations.

1. **Authentication:** During your initial attempt to communicate with a web server over a secure connection, that server will present your web browser with a set of credentials in the form of a server certificate. The purpose of the certificate is to verify that the site is who and what it claims to be. In some cases, the server may request a certificate that the client is who and what it claims to be (which is known as client authentication).
2. **Confidentiality:** When data is being passed between the client and the server on a network, third parties can view and intercept this data. SSL responses are encrypted so that the data cannot be deciphered by the third party and the data remains confidential.
3. **Integrity:** When data is being passed between the client and the server on a network, third parties can view and intercept this data. SSL helps guarantee that the data will not be modified in transit by that third party.

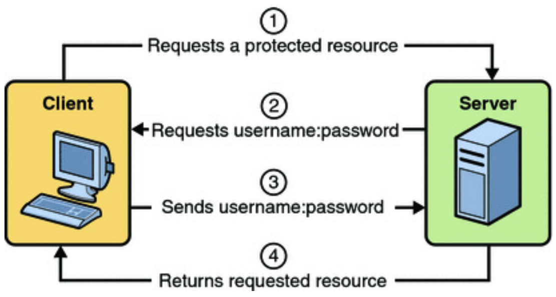

① Requests a protected resource
Client
Server
② Requests username:password
③ Sends username:password
④ Returns requested resource

FIGURE 10.2: Basic Security Authentication Mechanism

10.5 Authentication Mechanisms

The choices for authentication mechanisms are available to help one secure web applications:

1. HTTP Basic Authentication
2. Form-Based Authentication
3. HTTPS Client Authentication
4. Digest Authentication

10.5.1 HTTP Basic Authentication

Specifying HTTP Basic Authentication requires that the server request a user name and password from the web client and verify that the user name and password are valid by comparing them against a database of authorized users in the specified or default realm.

When basic authentication is declared, the following actions occur:

1. A client requests access to a protected resource.
2. The web server returns a dialog box that requests the user name and password.
3. The client submits the user name and password to the server.
4. The server authenticates the user in the specified realm and, if successful, returns the requested resource.

HTTP basic authentication is not a secure authentication mechanism. Basic authentication sends user names and passwords over the Internet as text that is Base64 encoded, and the target server is not authenticated. This form of authentication can expose user names and passwords. If someone can intercept the transmission, the user name and password information can easily be decoded. However, when a secure transport mechanism, such as SSL, or security at the network level, such as the IPSEC protocol or VPN strategies, is used in conjunction with basic authentication, some of these concerns can be alleviated.

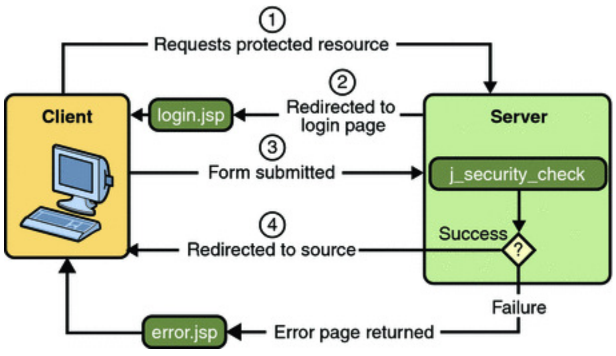


FIGURE 10.3: Security Form Based Authetication

10.5.2 Form-Based Authentication

Form-based authentication allows the developer to control the look and feel of the login authentication screens by customizing the login screen and error pages that an HTTP browser presents to the end user. When form-based authentication is declared, the following actions occur:

1. A client requests access to a protected resource.
2. If the client is unauthenticated, the server redirects the client to a login page.
3. The client submits the login form to the server.
4. The server attempts to authenticate the user.
   (a) If authentication succeeds, the authenticated users principal is checked to ensure it is in a role that is authorized to access the resource. If the user is authorized, the server redirects the client to the resource using the stored URL path.
   (b) If authentication fails, the client is forwarded or redirected to an error page.

Form-based authentication is not particularly secure. In form-based authentication, the content of the user dialog box is sent as plain text, and the target server is not authenticated. This form of authentication can expose your user names and passwords unless all connections are over SSL. If someone can intercept the transmission, the user name and password information can easily be decoded. However, when a secure transport mechanism, such as SSL, or security at the network level, such as the IPSEC protocol or VPN strategies, is used in conjunction with form-based authentication, some of these concerns can be alleviated. To add a protected transport in your application, use the elements described in Specifying a Secure Connection.

10.5.2.1 Using Login Forms

When creating a form-based login, be sure to maintain sessions using cookies or SSL session information.

As shown in Form-Based Authentication, for authentication to proceed appropriately, the action of the login form must always be **j_security_check**. This restriction is made so that the login form will work no matter which resource it is for, and to avoid requiring the server to specify the action field of the outbound form. The following code snippet shows how the form should be coded into the HTML page:

```
<form method="POST" action="j_security_check">
<input type="text" name="j_username">
<input type="password" name="j_password">
</form>
```

10.5.3 HTTPS Client Authentication

HTTPS Client Authentication requires the client to possess a Public Key Certificate (PKC). If you specify client authentication, the web server will authenticate the client using the clients public key certificate.

HTTPS Client Authentication is a more secure method of authentication than either basic or form-based authentication. It uses HTTP over SSL (HTTPS), in which the server authenticates the client using the clients Public Key Certificate (PKC). Secure Sockets Layer (SSL) technology provides data encryption, server authentication, message integrity, and optional client authentication for a TCP/IP connection. You can think of a public key certificate as the digital equivalent of a passport. It is issued by a trusted organization, which is called a certificate authority (CA), and provides identification for the bearer.

10.5.3.1 Mutual Authentication

With mutual authentication, the server and the client authenticate one another. There are two types of mutual authentication:

1. Certificate-based mutual authentication
2. User name- and password-based mutual authentication

10.5.3.2 Certificate-based mutual authentication

When using certificate-based mutual authentication, the following actions occur:

1. A client requests access to a protected resource.
2. The web server presents its certificate to the client.
3. The client verifies the servers certificate.
4. If successful, the client sends its certificate to the server.
5. The server verifies the clients credentials.

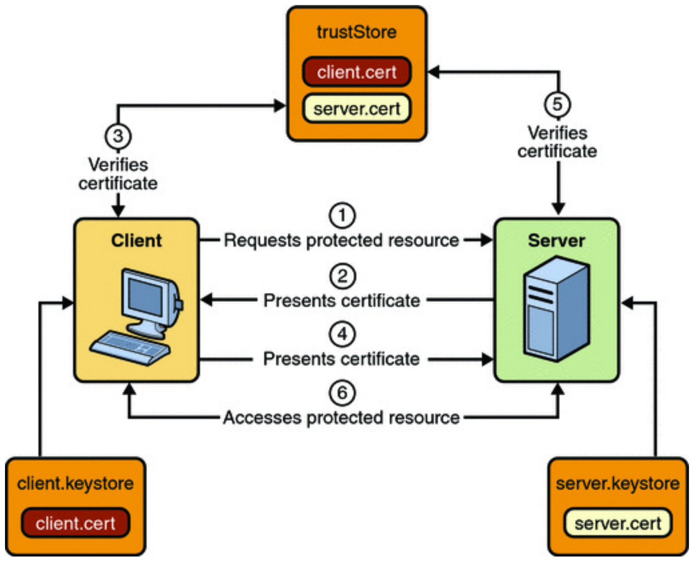


FIGURE 10.4: Certificate-Based Mutual Authentication

6. If successful, the server grants access to the protected resource requested by the client.

10.5.3.3 User name- and password-based mutual authentication

In user name- and password-based mutual authentication, the following actions occur:

1. A client requests access to a protected resource.
2. The web server presents its certificate to the client.
3. The client verifies the servers certificate.
4. If successful, the client sends its user name and password to the server, which verifies the clients credentials.
5. If the verification is successful, the server grants access to the protected resource requested by the client.

10.5.4 Digest Authentication

Like HTTP Basic Authentication, HTTP Digest Authentication authenticates a user based on a username and a password. However, unlike HTTP Basic Authentication, HTTP Digest Authentication does not send user passwords over the network. In HTTP Digest authentication, the client sends a one-way cryptographic hash of the password (and additional data). Although passwords are not sent on the wire, HTTP Digest authentication requires that clear text password equivalents be available to the authenticating container so that it can validate received authenticators by calculating the expected digest.

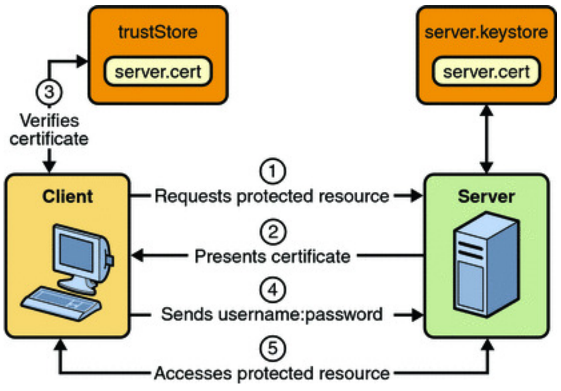


FIGURE 10.5: User Name- and Password-Based Mutual Authentication

10.6 Security With SpringFramework

The application we are using in the lab uses SpringFramework. To configure security you first need to setup two entities one with username and passowrd and the the with ROLES that a user will be assigned once loged in. Below are excepts of the two entities. In this class were are going to use a FORM BASED authetication mechanism

For the user:

```
public class Users implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(unique = true)
    private String username;
    private String passwd;
    private boolean enabled;
    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name = "role_id")
    private List<Roles> roles;
```

For the Roles:

```
public class Roles implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String roleName;
    private String username;
    @ManyToOne(cascade = CascadeType.ALL)
    private Users user;
```

Please note the relation ship between the entinties

Next we need to modify our application to add the Spring Security Filter as shown below in the web.xml

```
...........
 <servlet-mapping>
        <servlet-name>CXFServlet</servlet-name>
        <url-pattern>/ws/*</url-pattern>
    </servlet-mapping>
    <filter>
        <filter-name>springSecurityFilterChain</filter-name>

<filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <servlet>
        <servlet-name>hashpayservlet</servlet-name>
        <servlet-class>
        com.vaadin.terminal.gwt.server.ApplicationServlet
        </servlet-class>
        <init-param>
            <param-name>application</param-name>
            <param-value>com.hashpay.clients.web.HashPayMain</param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>hashpayservlet</servlet-name>
        <url-pattern>/app/*</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>hashpayservlet</servlet-name>
        <url-pattern>/VAADIN/*</url-pattern>
    </servlet-mapping>
..........
```

Not also the the change has been made to the application context so that the user can not access it directly, but through a redirect call in the index file. Note that the app contect has been added to the mapping

Next we create the index file with the redirect call as show below in the root folder of the web folder for the web resourses.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<% response.sendRedirect("/university/app/"); %>
```

After the redirect, we create two files in the web resources folder, the login.jsp file, which will have the form and the accessDenied.jsp

```
<center>
    <form action="j_spring_security_check" method="POST">
        <label for="j_username">Username</label>
        <input type="text" name="j_username" id="j_username" />
        <br/>
        <label for="j_password">Password</label>
        <input type="password" name="j_password" id="j_password"/>
        <br/>
        <input type='checkbox' name='_spring_security_remember_me'/> Remember me
on this computer.
        <br/>
        <input type="submit" value="Login"/>
    </form>
</center>
```

accessDenied.jsp

```
 <%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>Foo</title>
</head>
<body>
    <p>You have tried to access a protected area of this application where
you don't have rights.</p>
</body>
```

```
</html>
```

Lastly we create a the ApplicationContextsecurty.xml in the conf folder of our application. This is the file that forms the heart oof the security mechanism. A sample is shown below.

```
<!--SECTION A -->
<beans:beans xmlns="http://www.springframework.org/schema/security"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
          http://www.springframework.org/schema/security

http://www.springframework.org/schema/security/spring-security-3.0.xsd">

<!-- SECTION B --->

    <global-method-security secured-annotations="enabled">

    </global-method-security>

<--! SECITION C -->

    <http auto-config="true">
        <intercept-url pattern="/login.jsp*" access="IS_AUTHENTICATED_ANONYMOUSLY"/>
        <intercept-url pattern="/VAADIN/**" filters="none" />
        <intercept-url pattern="/*.jsp" filters="none" />
        <intercept-url pattern="/app/**" access="ROLE_CUSTOMER" />
        <form-login login-page='/login.jsp'/>
    </http>

<-- SECTION D -->

    <authentication-manager alias="authenticationManager">
        <authentication-provider>
            <password-encoder hash="md5"/>
            <jdbc-user-service data-source-ref="dataSource"
        users-by-username-query="SELECT username, passwd, enabled FROM Users u where u.↩
    username=?"
        authorities-by-username-query="SELECT u.username, r.rolename FROM Users u, Roles r
        WHERE u.id = r.role_id AND u.username=?"/>
        </authentication-provider>
    </authentication-manager>
</beans:beans>
```

The configuration file is divided into 4 Sections A, B, C, A

Section A deals with the namespace configuration. Section B is used to provide security annotations on the application classes. Section C is used to protect resources and set the form configurations and type of authentication mechanism. Section D define where the users and roles are stored in the database.

Lastly Make sure you have a Data Source in your ApplicationContext-connection.xml as shown below

```
 <context:property-placeholder location="classpath:database.properties"/>
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="↩
    close">
        <property name="driverClassName" value="${database.driverClassName}"/>
        <property name="url" value="${database.url}"/>
        <property name="username" value="${database.username}"/>
        <property name="password" value="${database.password}"/>
        <property name="maxActive" value="10" />
        <property name="maxIdle" value="3" />
        <property name="maxWait" value="5000" />
        <property name="defaultAutoCommit" value="true" />
    </bean>
```

Notice that the database variables are stored in a properties file called database.properties. So you need to creeate this file in your root folder as shown below

```
 # To change this template, choose Tools | Templates
```

```
# and open the template in the editor.
database.url=jdbc:derby://localhost:1527/sample
database.generateDdl=true
database.username=app
database.password=app
database.showSql=true
database.platform=org.hibernate.dialect.DerbyDialect
database.driverClassName=org.apache.derby.jdbc.ClientDriver
```

10.7 Spring Security Configurations

10.7.1 Security Namespace Configuration

A namespace element can be used simply to allow a more concise way of configuring an individual bean or, more powerfully, to define an alternative configuration syntax which more closely matches the problem domain and hides the underlying complexity from the user.

```
 <beans:beans xmlns="http://www.springframework.org/schema/security"
   xmlns:beans="http://www.springframework.org/schema/beans"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://www.springframework.org/schema/beans
             http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
             http://www.springframework.org/schema/security

http://www.springframework.org/schema/security/spring-security-3.0.xsd">
    ...
</beans:beans>
```

10.7.2 Adding a Password Encoder

Often your password data will be encoded using a hashing algorithm. This is supported by the ¡password-encoder¿ element. With SHA encoded passwords, the original authentication provider configuration would look like this:

```
  <authentication-manager>
    <authentication-provider>
      <password-encoder hash="sha"/>
      <user-service>
        <user name="jimi" password="d7e6351eaa13189a5a3641bab846c8e8c69ba39f"
              authorities="ROLE_USER, ROLE_ADMIN" />
        <user name="bob" password="4e7421b1b8765d8f9406d87e7cc6aa784c4ab97f"
              authorities="ROLE_USER" />
      </user-service>
    </authentication-provider>
  </authentication-manager>
```

Please Note that you have to save your passoword in the encoded format when you create a user. The code listing below shows you how to encode a passowrd in MD5 in Java

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.NoSuchProviderException;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 *
 * @author boniface
 */
public class PasswordEncrypt {

    public static String encrypt(String freeText) {
        StringBuffer getString = new StringBuffer();
        try {
            MessageDigest msg = MessageDigest.getInstance("MD5", "SUN");
            byte bs[] = freeText.getBytes();
            byte digest[] = msg.digest(bs);
            for (int i = 0; i < digest.length; ++i) {
                getString.append(Integer.toHexString(0x0100 + (digest[i] & 0x00FF)).↩
    substring(1));
            }
        } catch (NoSuchAlgorithmException ex) {
            Logger.getLogger(PasswordEncrypt.class.getName()).log(Level.SEVERE, null, ex);
        } catch (NoSuchProviderException ex) {
            Logger.getLogger(PasswordEncrypt.class.getName()).log(Level.SEVERE, null, ex);
        }
        return getString.toString();
    }
}
```

and the you would set the passoword in your class as

```
....
Users user = new Users();
user.setPassword(new PasswordEncrypt.encrypt("test123"));
.....
```

10.7.3 Adding HTTP/HTTPS Channel Security

If your application supports both HTTP and HTTPS, and you require that particular URLs can only be accessed over HTTPS, then this is directly supported using the requires-channel attribute on

```
<intercept-url>:
    <http>
      <intercept-url pattern="/secure/**" access="ROLE_USER" requires-channel="https"/>
      <intercept-url pattern="/**" access="ROLE_USER" requires-channel="any"/>
      ...
    </http>
```

With this configuration in place, if a user attempts to access anything matching the "secure**" pattern using HTTP, they will first be redirected to an HTTPS URL. The available options are "http", "https" or "any". Using the value "any" means that either HTTP or HTTPS can be used. If your application uses non-standard ports for HTTP andor HTTPS, you can specify a list of port mappings as follows:

```
    <http>
      ...
      <port-mappings>
        <port-mapping http="9080" https="9443"/>
      </port-mappings>
    </http>
```

10.7.4 Concurrent Session Control

If you wish to place constraints on a single user's ability to log in to your application, Spring Security supports this out of the box with the following simple additions. First you need to add the following listener to your web.xml file to keep Spring Security updated about session lifecycle events:

```
     <listener>
       <listener-class>
          org.springframework.security.web.session.HttpSessionEventPublisher
```

```
    </listener-class>
  </listener>
```

Then add the following lines to your application context:

```
  <http>
    ...
    <session-management>
         <concurrency-control max-sessions="1" />
    </session-management>
  </http>
```

This will prevent a user from logging in multiple times - a second login will cause the first to be invalidated. Often you would prefer to prevent a second login, in which case you can use

```
  <http>
    ...
    <session-management>
         <concurrency-control max-sessions="1"
       error-if-maximum-exceeded="true" />
    </session-management>
  </http>
```

The second login will then be rejected. By rejected, we mean that the user will be sent to the authentication-failure-url if form-based login is being used. If the second authentication takes place through another non-interactive mechanism, such as remember-me, an unauthorized (402) error will be sent to the client. If instead you want to use an error page, you can add the attribute session-authentication-error-url to the session-management element.

.

10.7.5 Session Fixation Attack Protection

Session fixation attacks are a potential risk where it is possible for a malicious attacker to create a session by accessing a site, then persuade another user to log in with the same session (by sending them a link containing the session identifier as a parameter, for example). Spring Security protects against this automatically by creating a new session when a user logs in. If you don't require this protection, or it conflicts with some other requirement, you

can control the behaviour using the session-fixation-protection attribute on ¡session-management¿, which has three options

- migrateSession - creates a new session and copies the existing session attributes to the new session. This is the default.
- none - Don't do anything. The original session will be retained.
- newSession - Create a new "clean" session, without copying the existing session data.

10.7.6 Method Security

The ¡global-method-security¿ Element is used to enable annotation-based security in your application (by setting the appropriate attributes on the element), and also to group together security pointcut declarations which will be applied across your entire application context. You should only declare one ¡global-method-security¿ element. The following declaration would enable support for Spring Security's @Secured:

```
     <global-method-security secured-annotations="enabled" />
```

Adding an annotation to a method (on an class or interface) would then limit the access to that method accordingly. Spring Security's native annotation support defines a set of attributes for the method. These will be passed to the AccessDecisionManager for it to make the actual decision:

```
     public interface BankService {
       @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
       public Account readAccount(Long id);
       @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
       public Account[] findAccounts();
       @Secured("ROLE_TELLER")
       public Account post(Account account, double amount);
     }
```

Support for JSR-250 annotations can be enabled using

```
   <global-method-security jsr250-annotations="enabled" />
```

These are standards-based and allow simple role-based constraints to be applied but do not have the power Spring Security's native annotations. To use the new expression-based syntax, you would use

```
    <global-method-security pre-post-annotations="enabled" />

and the equivalent Java code would be
      public interface BankService {
        @PreAuthorize("isAnonymous()")
        public Account readAccount(Long id);
        @PreAuthorize("isAnonymous()")
        public Account[] findAccounts();
        @PreAuthorize("hasAuthority('ROLE_TELLER')")
        public Account post(Account account, double amount);
      }
```

Expression-based annotations are a good choice if you need to define simple rules that go beyond checking the role names against the user's list of authorities. You can enable more than one type of annotation in the same application, but you should avoid mixing annotations types in the same interface or class to avoid confusion.

10.7.7 Obtaining information about the current user

Inside the SecurityContextHolder we store details of the principal currently interacting with the application. Spring Security uses an Authentication object to represent this information. You won't normally need to create an Authentication object yourself, but it is fairly common for users to query the Authentication object. You can use the following code block - from anywhere in your application - to obtain the name of the currently authenticated user, for example:

```
  Object principal =
SecurityContextHolder.getContext().getAuthentication().getPrincipal();
  if (principal instanceof UserDetails) {
     String username = ((UserDetails)principal).getUsername();
  } else {
    String username = principal.toString();
  }
```

The object returned by the call to getContext() is an instance of the SecurityContext interface. This is the object that is kept in thread-local storage. As we'll see below, most authentication mechanisms withing Spring Security return an instance of UserDetails as the principal.

10.8 Chapter Exercise

Apply the Securty to your application with users with different roles and athorities on which methods to call and display appropritate messages if errors occur.

Chapter 11

Mobile Device Application Development

11.1 Chapter Objectives

1. Create, test and debug applications for the Mobile Information Device Profile (MIDP) 1.0, 2.0, 2.1 (MSA), the Connected Limited Device Configuration (CLDC) 1.0 and 1.1, and the Connected Device Configuration (CDC).

2. Write applications that access web services directly from Symbian or Android powered mobile phones. Use the Wireless Connection Bridge to access web services (or other server-side data) via appropriate technology from any device using MIDlets

3. Write applications for other the Smart Phones like the Blackberry and Android powered Phones

11.2 Introduction

Computing continues to become more personal increasingly accessible anytime,anywhere. At the forefront of this development are handheld devices that are transforming into computing platforms. Mobile phones are no longer just for talking they have been capable of carrying data and video for some time. Significantly, the mobile device is becoming so capable of general-purpose computing that its destined to become the next PC (Personal Computer).

The battle lines between operating systems, computing platforms, programming languages, and development frameworks are being shifted and reapplied to mobile devices. We are also expecting a surge in mobile programming in the IT industry as more and more IT applications start to offer mobile counterparts.

To help you profit from this trend,this chapter will teach your about the technologies that are being used out there for mobile devices. Traditional there is a pattern emerging in the mobile platform that this course will address if time permits. The majors platforms out there are:

1. **Java 2 Micro Edition (J2ME)** - the is the technology supported by an average phone on the market today. J2ME works with practically any phone on the market today. Ig you can run MIXIT on your phone, then you have support for J2ME.
2. **Android**-This is an Android Mobile Software Development Kit (SDK) and platform from Google, and its hardware partners in the Open Handset Alliance. Android works with the HTC G1 and G2 phones, the DROID, and the Nexus One. Android is also available on other mobile phones and devices, including netbooks.
3. **Blackberry**- This is the technology that has been around for a while and with the prices of the blackberry dropping, it penetration could increase. This technology just runs on the Blackberry Phone.
4. **IPhone Ipad**-Undoubtedly this is the phone that has had the biggest impact on the smart phones space. Technology is tightly controlled by Apple and requires Apple licence to get the application Development Kit. We are unlikely to cover the development of this platform because of licensing issues.

11.3 Java ME Platform

Java ME technology was originally created in order to deal with the constraints associated with building applications for small devices. For this purpose Sun defined the basics for Java ME technology to fit such a limited environment and make it possible to create Java applications running on small devices with limited memory, display and power capacity.

Java ME platform is a collection of technologies and specifications that can be combined to construct a complete Java runtime environment specifically to fit the requirements of a particular device or market. This offers a flexibility and co-existence for all the players in the eco-system to seamlessly cooperate to offer the most appealing experience for the end-user.

The Java ME technology is based on three elements;

1. a configuration provides the most basic set of libraries and virtual machine capabilities for a broad range of devices,
2. a profile is a set of APIs that support a narrower range of devices, and
3. an optional package is a set of technology-specific APIs.

Over time the Java ME platform has been divided into two base configurations, one to fit small mobile devices and one to be targeted towards more capable mobile devices like smart-phones and set top boxes.

The configuration for small devices is called the Connected Limited Device Configuration (CLDC) and the more capable configuration is called the Connected Device Configuration (CDC).

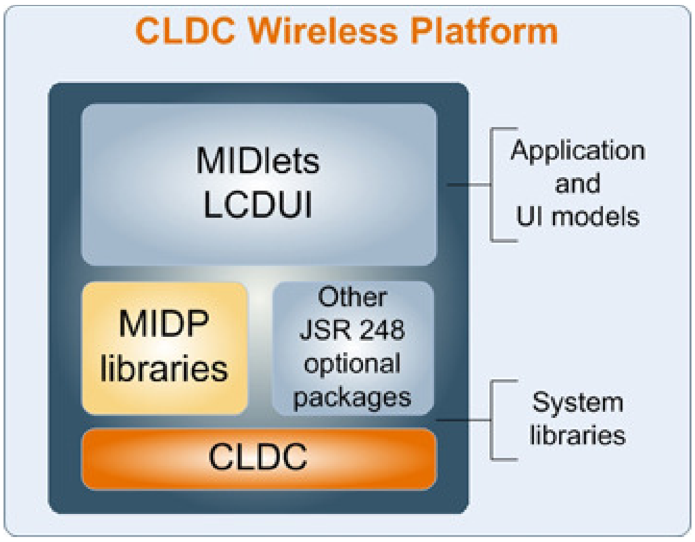


FIGURE 11.1: The Connected Limited Device Configuration

11.3.1 Configuration for Small Devices - The Connected Limited Device Configuration (CLDC)

The configuration targeting resource-constraint devices like mobile phones is called the Connected Limited Device Configuration (CLDC). It is specifically designed to meet the needs for a Java platform to run on devices with limited memory, processing power and graphical capabilities. On top of the different configurations Java ME platform also specifies a number of profiles defining a set of higher-level APIs that further define the application. A widely adopted example is to combine the CLDC with the Mobile Information Device Profile (MIDP) to provide a complete Java application environment for mobile phones and other devices with similar capabilities.

With the configuration and profiles the actual application then resides, using the different available APIs in the profile. For a CLDC and MIDP environment, which is typically what most mobile devices today are implemented with, a MIDlet is then created. A MIDlet is the application created by a Java ME software developer, such as a game, a business application or other mobile features. These MIDlets can be written once and run on every available device conforming with the specifications for Java ME technology. The MIDlet can reside on a repository somewhere in the ecosystem and the end user can search for a specific type of application and having it downloaded over the air to his/her device.

11.3.2 Configuration for More Capable Devices and SmartPhones - The Connected Device Configuration (CDC)

The configuration targeted larger devices with more capacity and with a network-connection, like high-end personal digital assistants, and set-top boxes, is called the Connected Device Profile (CDC). The goals of the CDC configuration is to leverage technology skills and developer tools based on the Java Platform Standard Edition (SE), and to support the feature sets of a broad range of connected devices while fitting within their resource constraints.

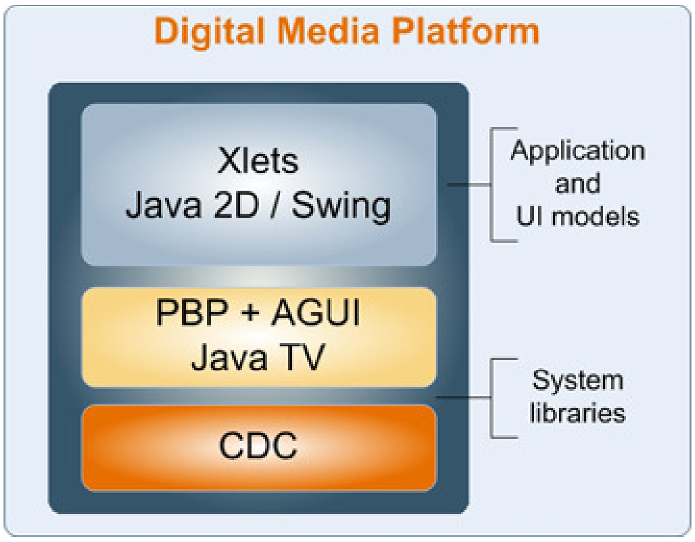


FIGURE 11.2: The Connected Device Configuration

Looking at the benefits the CDC configuration brings to the different groups in the value-chain the following can be said:

- Enterprises benefit from using network-based applications that extend the reach of business logic to mobile customers, partners and workers.
- Users will benefit from the compatibility and security of Java technology.
- Developers benefit from the safety and productivity of the Java programming language and the rich APIs in the Java platform.

On the CDC configuration there are three different defined profiles:

- The Foundation Profile (JSR 219)
- The Personal Basis Profile (JSR 217) and
- The Personal Profile (JSR 216)

11.3.2.1 Worked Examples

TextFieldCheck

Our first example of an MIDP application is a simple MIDlet to demonstrate the use of a TextField. You can use it in several classes and methods that are required in most MIDlets. These include, especially, the class MIDlet (which has to be used in every MIDlet) and the methods of this class for managing the life cycle of a MIDlet (such as startApp, pauseApp, and destroyApp). You can run this application in J2ME Wireless Toolkit embeded in Netbeans. On starting the MIDlet, the user sees a screen displaying the name of the MIDlet. On selecting it, the user is shown a TextField in which he can enter any text. After he has done this, he can press the OK button. Upon this, a message is displayed that the user has successfully entered whatever was required refer to view the Output. Of course, the button is just a label above the actual button of the mobile phone. The top-right button is used as an OK button, and the top-left button is used as an Exit button.

```
 /*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package com.cput.mobile.texttieldcheck;

import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.StringItem;
import javax.microedition.lcdui.TextField;
import javax.microedition.midlet.MIDlet;

/**
 *
 * @author boniface
 */
public class TextFieldCheck extends MIDlet implements CommandListener {
    // Declaring variable for Display class.

    private Display display = null;
    // Declaring variable for StringItem class.
    StringItem string_item = null;
    // Declaring variable for Form class.
    Form ui_holder = null;
    // Declaring variables for Buttons in the UI.
    Command ok = null;
    Command quit = null;
    // Declaring variable for the TextField.
    TextField textcheck = null;

    public TextFieldCheck() {
        // Initializing the Display
        display = Display.getDisplay(this);
        // Initializing the Buttons
        ok = new Command("Ok", Command.SCREEN, 3);
        quit = new Command("Quit", Command.SCREEN, 2);
    }

    public void startApp() {
        ui_holder = new Form("User Interface - TextField");
        // Initializing the Form
        textcheck = new TextField("Enter the Text Here..", "", 50, 0);
        string_item = new StringItem("User Entered ..", "");
        // Adding TextField to the form (place holder)
        ui_holder.append(textcheck);
// Adding Stringitem to the form (place holder)
        ui_holder.append(string_item);
// Adding Command Button to the Form
        ui_holder.addCommand(ok);
        // Adding Command Button to the Form
        ui_holder.addCommand(quit);
        // Invoking Action Listener
        ui_holder.setCommandListener(this);
        /*
         * Making the Display Current so that it can show
         * the Form.
         */
        display.setCurrent(ui_holder);
    }

    public void pauseApp() {
        string_item = null;
        ui_holder = null;
        textcheck = null;
    }

    public void destroyApp(boolean condition) {
        string_item = null;
        ui_holder = null;
        textcheck = null;
        // Destroy the form...
        notifyDestroyed();
    }

    public void commandAction(Command c, Displayable d) {
// Event handling for the Button
        if (c == ok) {
            string_item.setText(textcheck.getString());
        }
        if (c == quit) {
            destroyApp(true);
        }
    }
} // End of TextFieldCheck class.
```

LabelUI

The next MIDlet is about putting some text on the screen to serve as a label. This is done here in two ways. First, a string User Interface Label is passed as parameter to a Form named ui_holder. Then another string A simple Label... is added to the Form by calling the append method. The first string can be considered the title and the second can be the description

In the second way of doing a similar thing, two strings are passed as parameters to a StringItem, the first of which serves as a label for the second. Then this StringItem is added to the Form, again by calling the append method. There is an OK command so that the user can indicate that he has completed viewing the text.

Again note that all items have to be assigned null values in pauseApp and destroyApp methods.

```
 /*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package com.cput.mobile.labelui;

import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.StringItem;
import javax.microedition.midlet.*;

/**
 * @author boniface
 */
public class LabelUI extends MIDlet implements
        CommandListener {
    // Declaring variables for Display class.

    private Display display = null;
    // Declaring variables for StringItem class.
    StringItem string_item = null;
    // Declaring variables for Form class.
    Form ui_holder = null;
    // Declaring variables for Buttons.
    Command ok = null;

    public LabelUI() {
        // Initializing the Display
        display = Display.getDisplay(this);
        ok = new Command("Ok", Command.SCREEN, 3);
    }

    public void startApp() {
// Initializing the Form.
        ui_holder = new Form("User Interface - Label");
// Adding a simple label to the Form.
        ui_holder.append("A simple Label...");
        string_item = new StringItem("Example of a String Item....", "Hello World Label");
// Adding StringItem to the form (place holder).
        ui_holder.append(string_item);
// Adding Command Button to the Form.
        ui_holder.addCommand(ok);
// Invoking Action Listener..
        ui_holder.setCommandListener(this);
        /* Making the Display Current so that it can show
         * the Form.
         */
        display.setCurrent(ui_holder);
    }

    public void pauseApp() {
        string_item = null;
        ui_holder = null;
    }

    public void destroyApp(boolean condition) {
        string_item = null;
        ui_holder = null;
// Destroy the form.
        notifyDestroyed();
    }

    public void commandAction(Command c, Displayable d) {
        // Event handling for the Button.
        if (c == ok) {
            destroyApp(true);
        }
    }
} // End of LabelUI class.
```

ChoiceGroupUI

This MIDP application shows how you can put the equivalent of radio buttons and check boxes on the UI of a MIDlet. The class responsible for both these in MIDP is ChoiceGroup. It is only the value of the parameter choiceType that determines whether the ChoiceGroup elements will act as radio buttons or check boxes. The three types of ChoiceGroups are EXCLUSIVE, MULTIPLE, or IMPLICIT.

The label for the whole ChoiceGroup is set by passing the value of the label parameter as Choices. Text labels for the elements of a ChoiceGroup are added by passing the stringElements parameter with a value equal to a string array containing the labels. Since in this case we wanted no images, we have set the value of the imageElements as null. All the elements are added to the Form by calling the append method as before.

When the user makes his choices and presses the OK button, the commandAction is used for event handling. For this event handler, if blocks and a for loop are used. As a result of this, the user is shown a message reporting the choices made by him. He can get out of the application by pressing the Quit button.

```
 /*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package com.cput.mobile.choicegroupui;

import javax.microedition.lcdui.ChoiceGroup;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.Image;
import javax.microedition.lcdui.StringItem;
import javax.microedition.midlet.MIDlet;

/**
 *
 * @author boniface
 */
public class ChoiceGroupUI extends MIDlet
        implements CommandListener {
    // Declaring variable for the Display class.

    private Display display = null;
    // Declaring variables for the ChoiceGroup.
    ChoiceGroup radiobutton_type = null;
    ChoiceGroup checkbox_type = null;
    // Declaring variables for the StringItem class.
    StringItem string_item1 = null;
    StringItem string_item2 = null;
    // Declaring variables for the Form class.
    Form ui_holder = null;
    // Declaring variables for Buttons.
    Command ok = null;
    Command quit = null;
    String[] name = {"a", "b", "c"};
    String[] name1 = {"d", "e", "f"};
    Image[] img = null;
    boolean[] values = null;

    public ChoiceGroupUI() {
// Initializing the Display.
        display = Display.getDisplay(this);
// Initializing the Button.
        ok = new Command("Ok", Command.SCREEN, 3);
        quit = new Command("Quit", Command.SCREEN, 2);
    }
    public void startApp() {
// Initializing the Form.
        ui_holder = new Form("User Interface - ChoiceGroups");
        // Initializing the Choice groups.
        radiobutton_type = new ChoiceGroup("Choices..",
                ChoiceGroup.EXCLUSIVE, name, img);
        string_item1 = new StringItem("User Clicked ", "");
        checkbox_type = new ChoiceGroup("Choices..",
                ChoiceGroup.MULTIPLE, name1, img);
// Getting the values the user clicked.
        string_item2 = new StringItem("User Clicked  ", "");
// Adding StringItem to the form (place holder).
        ui_holder.append(radiobutton_type);
        ui_holder.append(string_item1);
        ui_holder.append(checkbox_type);
        ui_holder.append(string_item2);
// Adding Command Button to the Form.
        ui_holder.addCommand(ok);
        ui_holder.addCommand(quit);
        // Invoking Action Listener.
        ui_holder.setCommandListener(this);
        /* Making the Display Current so that it can show
         * the Form.
         */
        display.setCurrent(ui_holder);
    }

    public void pauseApp() {
        string_item1 = null;
        string_item2 = null;
        ui_holder = null;
        radiobutton_type = null;
        checkbox_type = null;
    }

    public void destroyApp(boolean condition) {
        string_item1 = null;
        string_item2 = null;
        ui_holder = null;
        radiobutton_type = null;
        checkbox_type = null;
// Destroy the form.
        notifyDestroyed();
    }

    public void commandAction(Command c, Displayable d) {
// Event handling for the Button.
        if (c == ok) {
            String temp = "";
            string_item1.setText(
                    radiobutton_type.getString(
                    radiobutton_type.getSelectedIndex()));
            for (int i = 0; i < 3; i++) {
                boolean val1 = checkbox_type.isSelected(i);
                if (val1) {
                    temp = temp + name1[i];
                }
            }
            string_item2.setText(temp);
        }
        if (c == quit) {
            destroyApp(true);
        }
    }
} // End of TextFieldCheck class.
```

TickerUI

This is a simple MIDlet, showing the use of the Ticker class. It basically shows the user some scrolling text somewhat like a marquee in HTML. The Ticker has to be passed some string as the parameter. This string will be shown scrolling horizontally on the screen . But before it happens, you have to call the setTicker method and pass the name of the Ticker object as an argument to this method. Note that we are not using the append method as we did in previous examples. There is an OK button for the user to indicate that the user has had enough of the Ticker.

```
 /*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package com.cput.mobile.tickerui;

import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.Ticker;
import javax.microedition.midlet.MIDlet;

/**
 *
 * @author boniface
 */
public class TickerUI extends MIDlet
        implements CommandListener {
// Declaring variables for Display class.

    private Display display = null;
// Declaring variables for Form class.
    Form ui_holder = null;
// Declaring variables for Buttons.
    Command ok = null;
// Declaring variables for Ticker in the UI.
    Ticker ui_ticker = null;

    public TickerUI() {
// Initializing the Display.
        display = Display.getDisplay(this);
// Initializing the Command Button.
        ok = new Command("Ok", Command.SCREEN, 3);
    }

    public void startApp() {
// Initializing the Form.
        ui_holder = new Form("User Interface - Ticker");
// Initializing the Ticker.
        ui_ticker = new Ticker("..This is an Example of a Ticker User Interface..");
// Adding Ticker to the Form.
        ui_holder.setTicker(ui_ticker);
// Adding Command Button to the Form.
        ui_holder.addCommand(ok);
// Invoking Action Listener.
        ui_holder.setCommandListener(this);
        /* Making the Display Current so that it can show
         * the Form.
         */
        display.setCurrent(ui_holder);
    }

    public void pauseApp() {
        ui_holder = null;
        ui_ticker = null;
    }

    public void destroyApp(boolean condition) {
        ui_holder = null;
        ui_ticker = null;
// Destroy the form.
        notifyDestroyed();
    }

    public void commandAction(Command c, Displayable d) {
// Event handling for the Button.
        if (c == ok) {
            destroyApp(true);
        }
    }
} // End of TickerUI.
```

MenuCheck

Now we present a more advanced application. This is a very useful MIDlet, since it shows how a menu can be created with MIDP. The class used for this is List. Just as in the case of ChoiceGroup, there can be three types of Lists: IMPLICIT, EXCLUSIVE, and MULTIPLE. We use the first variety here, as this will perhaps be the most used in applications. For this variety, we dont have to attach application- defined commands it is enough to register CommandListener that is called when the user makes a selection.

We begin building a menu by adding elements to a List by calling the append method. The other important thing to do is to implement the commandAction method of the CommandListener class.

We do this by using the switch block to determine what the user will get when he selects an entry on the menu. In our case, the user sees a message describing the selection he made. Needless to say, there is a Quit button here as well, since we dont want to lock in the user.

```
/*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */

package com.cput.mobile.menucheck;

import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.List;
import javax.microedition.lcdui.StringItem;
import javax.microedition.midlet.MIDlet;

/**
 *
 * @author boniface
 */
public class MenuCheck extends MIDlet implements CommandListener {
 private Display display = null;
 Command ok = null;
 Command quit = null;
 List menu = null;
 Form ui_form = null;
 StringItem si = null;
 public MenuCheck() {
             // Initializing the Display...
   display = Display.getDisplay(this);
   quit = new Command("Quit",Command.SCREEN,2);
 }
 public void startApp() {
   menu = new List("Various Options..",List.IMPLICIT);
   menu.append("TextField",null);
   menu.append("Ticker",null);
   menu.append("Alert",null);
   menu.setCommandListener(this);
   display.setCurrent(menu);
   ui_form = new Form("User's Choice...");
   si = new StringItem("User Entered ..", "");
   ui_form.append(si);
   // Adding Command Button to the ui_form.
   ui_form.addCommand(quit);
   // Invoking Action Listener
   ui_form.setCommandListener(this);
 }
 public void pauseApp() {
   menu = null;
 }
 public void destroyApp(boolean unconditional) {
   menu = null;
   notifyDestroyed();
 }
 public void commandAction(Command c, Displayable d) {
             // Event handling for the Button
   if ( c == quit ) {
     destroyApp(true);
   }
   else {
     List down = (List)display.getCurrent();
switch(down.getSelectedIndex()) {
case 0: si.setText("Text Field...");
break;
case 1: si.setText("Ticker...");
break;
case 2: si.setText("Alert...");
break;
     }
 display.setCurrent(ui_form);
   }
 }
} // End of MenuCheck.
```

AddressBook

This is perhaps the most advanced application in this chapter. In a way, this MIDlet is an extension of the MenuCheck MIDlet. It doesnt just show some message when the user makes a selection. Rather, it goes on to add some real functionality by allowing the user to add records to an address book, search them, delete them, and moreover, to quit the application.

The menu is created in the same way as previously by using the List class and adding elements to it by calling the append method see Figure 4-16. This List is again of IMPLICIT type. For adding entries to an address book (or for searching for and deleting them), you need to use database capability. This is where MIPDs persistence package and its RecordStore class are relevant. A RecordStore is opened by calling its open-RecordStore method. Records are added to it by calling the addRecord method. When the user selects Add Address on the menu, he gets two TextFields in which he can type his name and telephone number, respectively. The getString method is used on these TextFields to get the text user types. You can also see the use of ByteArrayOuput-Stream, DataOutputStream, and the writeUTF method.

For searching records, enumerateRecords method is used, along with while and if blocks. For deleting records, deleteRecord method is used, in addition to the enumerateRecords method. Dont forget that nextRecord method makes you capable of going to the next record while enumerating.

Finally, calling the closeRecordStore method of the RecordStore class closes the database. But before proceeding to the next MIDlet, take note of the exceptions and try...catch blocks, as well as the switch block that is used for event handling in case of the main menu.

```
 /*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package com.cput.mobile.addressbook;

import java.io.ByteArrayOutputStream;
import java.io.DataOutputStream;
import java.io.IOException;
import javax.microedition.lcdui.Command;
import javax.microedition.lcdui.CommandListener;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Displayable;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.List;
import javax.microedition.lcdui.StringItem;
import javax.microedition.lcdui.TextField;
import javax.microedition.midlet.MIDlet;
import javax.microedition.rms.InvalidRecordIDException;
import javax.microedition.rms.RecordEnumeration;
import javax.microedition.rms.RecordStore;
import javax.microedition.rms.RecordStoreException;
import javax.microedition.rms.RecordStoreNotOpenException;

/**
 *
 * @author boniface
 */
public class AddressBook extends MIDlet
        implements CommandListener {

    private Display display = null;
    Command search = null;
    Command quit = null;
    Command delete = null;
    Command addnow = null;
    Command mainmenu = null;
    // Declaring a Menu(List)
    List menu = null;
    Form ui_form = null;
    StringItem si = null;
    // TextField for storing the name...
    TextField name = null;
    TextField phone = null;
    // Declaring a RecordStore (Database)...
    RecordStore recordStore = null;

    public AddressBook() {
        display = Display.getDisplay(this);
        quit = new Command("Quit", Command.SCREEN, 3);
        search = new Command("Search", Command.SCREEN, 2);
        delete = new Command("Delete", Command.SCREEN, 2);
        addnow = new Command("Add", Command.SCREEN, 2);
        mainmenu = new Command("Main", Command.SCREEN, 2);
// Initializing the Record Store
        try {
            recordStore = RecordStore.openRecordStore(
                    "addresses", true);
        } catch (RecordStoreException rse) {
            rse.printStackTrace();
        }
    }
    public void startApp() {
        menu = new List("Address Book...", List.IMPLICIT);
        menu.append("1. Search Address", null);
        menu.append("2. Add Address", null);
        menu.append("3. Delete Address", null);
        menu.append("4. Quit", null);
        menu.setCommandListener(this);
        display.setCurrent(menu);
    }
    // GUI for the Search Screen...

    void searchScreen() {
        ui_form = new Form("Search An Address");
        name = new TextField("Name to Search..", "", 50, 0);
        ui_form.append(name);
        ui_form.addCommand(search);
        ui_form.addCommand(quit);
        // Invoking Action Listener...
        ui_form.setCommandListener(this);
        display.setCurrent(ui_form);
    }
    // GUI for the Addition Screen...

    void addScreen() {
        ui_form = new Form("Add an Entry..");
        name = new TextField("Name ..", "", 50, 0);
        ui_form.append(name);
        phone = new TextField("Phone No.. ", "", 50, 0);
        ui_form.append(phone);
        ui_form.addCommand(addnow);
        ui_form.addCommand(quit);
        // Invoking Action Listener...
        ui_form.setCommandListener(this);
        display.setCurrent(ui_form);
    }
    // GUI for the Delete Screen...

    void deleteScreen() {
        ui_form = new Form("Delete An Address");
        name = new TextField("Name to Delete..", "", 50, 0);
        ui_form.append(name);
        ui_form.addCommand(delete);
        ui_form.addCommand(quit);
        // Invoking Action Listener...
        ui_form.setCommandListener(this);
        display.setCurrent(ui_form);
    }

    public void pauseApp() {
        menu = null;
    }

    public void destroyApp(boolean unconditional) {
        menu = null;
        notifyDestroyed();
    }

    public void commandAction(Command c, Displayable d) {
// Event handling for the Button...
        if (c == quit) {
            try {
                close();
            } catch (RecordStoreException rse) {
                rse.printStackTrace();
            }
            destroyApp(true);
        } else if (c == search) {
            // When Search button is pressed (search_add is called)
            String temp_search = name.getString();
            search_add(temp_search);
        } else if (c == mainmenu) {
            // To return to Main Menu...
            startApp();
        } else if (c == delete) {
            // When delete button is pressed (delete_add) is called
            String temp_delete = name.getString();
            delete_add(temp_delete);
        } else if (c == addnow) {
            // When add button is pressed (address_add) is called
            String temp_name = name.getString();
            String temp_phone = phone.getString();
            address_add(temp_name, temp_phone);
        } else {
            List down = (List) display.getCurrent();
            switch (down.getSelectedIndex()) {
                case 0:
                    searchScreen();
                    break;
                case 1:
                    addScreen();
                    break;
                case 2:
                    deleteScreen();
                    break;
                case 3:
                    destroyApp(true);
                    break;
            }
        }
    }

    void search_add(String address) {
        // Function for searching...
        String temp = " ";
        String phone_number;
        String person_name;
        int size = address.length();
        try {
            RecordEnumeration re =
                    recordStore.enumerateRecords(
                    null, null, false);
            ui_form = new Form("Search Result.");
            while (re.hasNextElement()) {
                String name1 = new String(re.nextRecord());
                try {


                    person_name = name1.substring(
                            2, name1.indexOf("?"));
                } catch (Exception ef) {
                    person_name = "check";
                }
                String check_name =
                        person_name.substring(0, size);
                if (check_name.equals(address)) {
                    try {
                        phone_number = name1.substring(
                                name1.indexOf("?") + 1);
                    } catch (Exception e) {
                        phone_number = "";
                    }
                    temp = temp + "\nName.." +
                            person_name + "\nPhone.." +
                            phone_number;
                }
            }
            if (temp.equals(" ")) {
                temp = "The required address not found...";
            }
            ui_form.append(temp);
            ui_form.addCommand(quit);
            ui_form.addCommand(mainmenu);
            // Invoking Action Listener...
            ui_form.setCommandListener(this);
            display.setCurrent(ui_form);
        } catch (RecordStoreNotOpenException rsnoe) {
            rsnoe.printStackTrace();
        } catch (InvalidRecordIDException irid) {
            irid.printStackTrace();
        } catch (RecordStoreException rse) {
            rse.printStackTrace();
        }
    }

    void delete_add(String address) {
        // Function for deletion....
        String temp = " ";
        String phone_number;
        String person_name;
        int i = 1;
        int del_id = 0;
        try {
            RecordEnumeration re =
                    recordStore.enumerateRecords(
                    null, null, false);
            ui_form = new Form("Delete Result..");
            while (re.hasNextElement()) {
                String name1 = new String(re.nextRecord());
                try {
                    person_name = name1.substring(
                            2, name1.indexOf("?"));
                } catch (Exception ef) {
                    person_name = "check";
                }
                if (person_name.equals(address)) {
                    del_id = i;
                }
                i++;
            }
            if (del_id != 0) {
                recordStore.deleteRecord(del_id);
                temp = "One Record Deleted Successfully...";
            } else {
                temp = "The Record is not Listed...";
            }
        } catch (Exception e) {
        }
        ui_form.append(temp);
        ui_form.addCommand(quit);
        ui_form.addCommand(mainmenu);
        // Invoking Action Listener...
        ui_form.setCommandListener(this);
        display.setCurrent(ui_form);
    }
    void address_add(String address, String phone) {
        // Function for address addtion...
        String data = address + "?" + phone;
// ? (to distinguish between name and phone number
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        DataOutputStream outputStream =
                new DataOutputStream(baos);
        try {
            outputStream.writeUTF(data);
            byte[] b = baos.toByteArray();
            recordStore.addRecord(b, 0, b.length);
            ui_form = new Form("Success.....");
            ui_form.append("One Record Entered Successfully... ");
            ui_form.addCommand(quit);
            ui_form.addCommand(mainmenu);
            // Invoking Action Listener...
            ui_form.setCommandListener(this);
            display.setCurrent(ui_form);
        } catch (IOException ioe) {
            ioe.printStackTrace();
        } catch (RecordStoreException rse) {
            rse.printStackTrace();
        }
    }

    public void close() throws RecordStoreNotOpenException,
            RecordStoreException {
        recordStore.closeRecordStore();
    }
} // End of AddressBook.
```

TestHTTP

We end the J2ME worked examples by showing how connectivity can be established with a MIDlet. It is done by using the HttpConnection interface and Connector class of javax.io package. We first open an HTTP connection by using the open(url) method and then calling the openInputStream method on this connection in order to allow reading data from it by using the read(data) method. The file to be read is passed as a URL to the connection. Both the connection and the input stream are closed by calling the close method. This final part is done inside a finally block to release all the resources. It is a simple MIDlet, but a useful one. Expect an exception ot error and by reading the errors fix the problem.

```
 /*
 * To change this template, choose Tools | Templates
 * and open the template in the editor.
 */
package com.cput.mobile.testhttp;

import java.io.IOException;
import java.io.InputStream;
import javax.microedition.io.Connector;
import javax.microedition.io.HttpConnection;
import javax.microedition.lcdui.Display;
import javax.microedition.lcdui.Form;
import javax.microedition.lcdui.TextField;
import javax.microedition.midlet.MIDlet;

/**
 *
 * @author boniface
 */
public class TestHTTP extends MIDlet {

    String url = "http://yash/hello.txt";
    HttpConnection con = null;
    InputStream ins = null;
    String str = null;
    TextField tx = null;
    Form myform = null;
    private Display show = null;

    public TestHTTP() {
        try {
            try {
                con = (HttpConnection) Connector.open(url);
                ins = con.openInputStream();
            } catch (IOException ex) {
                ex.printStackTrace();
            }
            int i = (int) con.getLength();
            byte[] data = new byte[i];
            if (i > 0) {
                try {
                    int actual = ins.read(data);
                } catch (IOException ex) {
                    ex.printStackTrace();
                }
                str = new String(data);
            }
            show = Display.getDisplay(this);
            myform = new Form("User Interface − TextField");
            tx = new TextField(url, str, 70, 0);
            myform.append(tx);
            show.setCurrent(myform);
            System.out.println(str + "hello");
        } finally {
            try {
                if (con != null) {
                    con.close();
                }
                if (ins != null) {
                    ins.close();
                }
            } catch (IOException ex) {
                ex.printStackTrace();
            }
            str = null;
            tx = null;
        }
    }
    public void startApp() {
        new TestHTTP();
    }
    public void pauseApp() {
    }

    public void destroyApp(boolean b) {
    }
}
```

11.4 Android Platform

11.4.1 Introduction

Android is the first complete, open, and free mobile platform. Developers enjoy a compre- hensive software development kit, with ample tools for developing powerful, feature-rich applications. The platform is open source, relying on tried-and-true open standards devel- opers will be familiar with. And best of all, there are no costly barriers to entry for devel- opers: no required fees. (A modest fee is required to publish on third-party distribution mechanisms such as the Android Market.) Android developers have numerous options for distributing and commercializing their applications.

11.4.2 Concepts

The Concepts of the Android Development Platform are in the Power Point Slides shown in the class. You can get the slides from WEBCT

11.4.3 Android Development Environment

11.4.3.1 Android Operation System

Android is an operating system based on Linux with a Java programming interface. It provides tools, e.g. a compiler, debugger and a device emulator as well as its own Java Virtual machine (Dalvik Virtual Machine - DVM). Android is created by the Open Handset Alliance which is lead by Google.

Android uses a special Java virtual machine (Dalvik) which is based on the Apache Harmony Java implementation. Dalvik uses special bytecode. Therefore you cannot run standard Java bytecode on Android. Android provides a tool "dx" which allows to convert Java Class files into "dex" (Dalvik Executable) files. Android applications are then packed into an .apk (Android Package) file.

Android supports 2-D and 3-D graphics using the OpenGL libraries and supports data storage in a SQLLite database.

For development Google provides the Android Development Tools (ADT) for Eclipse to develop Android applications.

Every Android applications runs in its own process and it is isolated from other running applications. Therefore on misbehaving application cannot harm other Android applications.

11.4.3.2 Important Android terms

An Android application consists out of the following parts:

1. Activity - A screen in the Android application
2. Services - Background activities without UI
3. Content Provider - provides data to applications, Android contains a SQLLite DB which can serve as data provider

11.4.3.3 Activities

The building block of the user interface is the activity. Activity is the Android analogue for the window or dialog in a desktop application, or the page in a classic Web app. Android is designed to support lots of cheap activities, so you can allow users to keep clicking to bring up new activities and tapping the BACK button to back up, just like they do in a Web browser.

Intends allow the application to request and / or provide services . For example the application call ask via an intent for a contact application. Application register themself via an IntentFilter. Intends are a powerful concept as they allow to create loosely coupled applications.

An Android application is described the file "AndroidManifest.xml". This files contains all activities application and the required permissions for the application. For example if the application requires network access it must be specified here. "AndroidManifest.xml" can be thought as the deployment descriptor for an Android application.

11.4.3.4 Activities

The user interface for Activities is defined via layouts. The layout defines the UI elements, their properties and their arragement. A layout can be defined via XML and via code at runtime. The XML way is usually preferred for a fixed layout while defining the layout via code is more flexible. You can also mix both approaches.

11.4.3.5 Services

Activities are short-lived and can be shut down at any time. Services, on the other hand, are designed to keep running, if needed, independent of any activity. You might use a service for checking for updates to an RSS feed, or to play back music even if the controlling activity is no longer operating. You will also use services for scheduled tasks (”cron jobs”) and for exposing custom APIs to other applications on the device, though those are relatively advanced capabilities.

11.4.3.6 Content Providers

Content providers provide a level of abstraction for any data stored on the device that is accessible by multiple applications. The Android development model encourages you to make your own data available to other applications, as well as your own building a content provider lets you do that, while maintaining complete control over how your data gets accessed.

11.4.3.7 Intents

Intents are system messages, running around the inside of the device, notifying applications of various events, from hardware state changes (e.g., an SD card was inserted), to incoming data (e.g., an SMS message arrived), to application events (e.g., your activity was launched from the device’s main menu). Not only can you respond to an Intent, but you can create your own, to launch other activities, or to let you know when specific situations arise (e.g., raise such-and-so Intent when the user gets within 100 meters of this-and-such location).

11.4.4 Storage

You can package data files with your application, for things that do not change, such as icons or help files. You also can carve out a small bit of space on the device itself, for databases or files containing user-entered or retrieved data needed by your application. And, if the user supplies bulk storage, like an SD card, you can read and write files on there as needed.

11.4.5 Network

Android devices will generally be Internet-ready, through one communications medium or another. You can take advantage of the Internet access at any level you wish, from raw Java sockets all the way up to a built-in WebKit-based Web browser widget you can embed in your application.

11.4.6 Multimedia

Android devices have the ability to play back and record audio and video. While the specifics may vary from device to device, you can query the device to learn its capabilities and then take advantage of the multimedia capabilities as you see fit, whether that is to play back music, take pictures with the camera, or use the microphone for audio note-taking.

11.4.7 GPS

Android devices will frequently have access to location providers, such as GPS, that can tell your applications where the device is on the face of the Earth. In turn, you can display maps or otherwise take advantage of the location data, such as tracking a device's movements if the device has been stolen.

11.4.8 Phone Services

Lastly, Android devices are typically phones, they allow your software to initiate calls, send and receive SMS messages, and everything else you expect from a modern bit of telephony technology.

11.4.9 Worked Example

We are going to create an application called Sudoku. If you don't know how Sudoku works , just google. Create the Project and make sure and see the screen shot show below.

[DIAG]

11.4.9.1 Quick Start

Well create a detailed example: a Sudoku game. By gradually adding features to the game, youll learn about many aspects of Android programming. Well start with the user interface.

11.4.9.2 Introducing the Sudoku Example

Sudoku makes a great sample program for Android because the game itself is so simple. You have a grid of eighty-one tiles (nine across and nine down), and you try to fill them in with numbers so that each column, each row, and each of the three-by-three boxes contains the numbers 1 through 9 only once. When the game starts, some of the numbers (the givens) are already filled in. All the player has to do is supply the rest. A true Sudoku puzzle has only one unique solution.

Sudoku is usually played with pencil and paper, but computerized versions are quite popular too. With the paper version, its easy to make a mistake early on, and when that happens, you have to go back and erase most of your work.

11.4.9.3 Designing by Declaration

User interfaces can be designed using one of two methods: procedural and declarative.

Procedural simply means programmiinng in Java code. For example, when youre programming a Swing application, you write Java code to create and manipulate all the user interface objects such as JFrame and JButton. Thus, Swing is procedural.

Declarative design, on the other hand, does not involve any code. When youre designing a simple web page, you use HTML, a markup language similar to XML that describes what you want to see on the page, not how you want to do it. HTML is declarative.

Android tries to straddle the gap between the procedural and declarative worlds by letting you create user interfaces in either style. You can stay almost entirely in Java code, or you can stay almost entirely in XML descriptors.

If you lookup the documentation for any Android user interface component, youll see both the Java APIs and the corresponding declarative XML attributes that do the same thing. Which should you use? Either way is valid, but Googles advice is to use declarative XML as much as possible. The XML code is often shorter and easier to understand than the corresponding Java code, and its less likely to change in future versions.

11.4.9.4 Creating the Start Up Screen

Well start with a skeleton Android program created by the Eclipse plugin. Create a new new project, with following values:

Project name: Sudoku Build Target: Android 2.1 Application name: Sudoku Package name: za.ac.cput.sudoku Create Activity: Sudoku Min SDK Version: 7

The package name is particularly important. Each application in the system must have a unique package name. Once you choose a package name, its a little tricky to change it because its used in so many places.

Keep the Android emulator window up all the time and run the program after every change, since it takes only a few seconds.

The first order of business is to create an opening screen for the game, with buttons to let the player start a new game, continue a previous one, get information about the game, and exit.

Note, Android applications are a loose collection of activities, each of which define a user interface screen. When you create the Sudoku project, the Android plugin makes a single activity for you in Sudoku.java:

Android calls the onCreate( ) method of your activity to initialize it. The call to setContentView( ) fills in the contents of the activitys screen with an Android view widget.

We could have used several lines of Java code, and possibly another class or two, to define the user interface procedurally. But instead, we shall use the declarative method. The **R.layout.main** is a resource identifier that refers to the **main.xml** file in the **res/layout** directory.

main.xml declares the user interface in XML, so thats the file we need to modify. At runtime, Android parses and instantiates the resource defined there and sets it as the view for the current activity

Its important to note that the R class is managed automatically by the Android Eclipse plugin. When you put a file anywhere in the res directory, the plug-in notices the change and adds resource IDs in R.java in the gen directory for you. If you remove or change a resource file, R.java is kept in sync.

Note that almost every Android program, including the base Android framework itself, has an R class.

To modify main.xml open the file and you will see the listing below

```
<LinearLayout
 xmlns:android="http://schemas.android.com/apk/res/android"
 android:orientation="vertical"
 android:layout_width="fill_parent"
 android:layout_height="fill_parent" >
<!-- ... -->
</LinearLayout>
```

A layout is a container for one or more child objects and a behavior to position them on the screen within the rectangle of the parent object. Here is a list of the most common layouts provided by Android:

1. FrameLayout: Arranges its children so they all start at the top left of the screen. This is used for tabbed views and image switchers.
2. LinearLayout: Arranges its children in a single column or row. This is the most common layout you will use.
3. RelativeLayout: Arranges its children in relation to each other or to the parent. This is often used in forms.
4. TableLayout: Arranges its children in rows and columns, similar to an HTML table.

Some parameters are common to all layouts:

```
 xmlns:android="http://schemas.android.com/apk/res/android"
```

Defines the XML namespace for Android. You should define this once, on the first XML tag in the file.

```
Takes up the entire width and height of the parent (in this case,the window). Possible ↩
     values are fill_parent and
wrap_content.

Inside the <LinearLayout> tag you ll find one child widget:
\begin{lstlisting}

<TextView
android:layout_width="fill_parent"
android:layout_height="wrap_content"
android:text="@string/hello" />
```

This defines a simple text label. Lets replace that with some different text and a few buttons.

```
<?xml version="1.0" encoding="utf-8"?>
<!--

-->
<LinearLayout
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:orientation="vertical"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent">
   <TextView
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="@string/main_title" />
   <Button
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="@string/continue_label" />
   <Button
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="@string/new_game_label" />
   <Button
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="@string/about_label" />
   <Button
      android:layout_width="fill_parent"
      android:layout_height="wrap_content"
      android:text="@string/exit_label" />
</LinearLayout>
```

If you see warnings in the editor about missing grammar constraints (DTD or XML schema), just ignore them. Instead of hard-coding English text into the layout file, we use the @string/resid syntax to refer to strings in the res/values/strings.xml file. You can have different versions of this and other resource files based on the locale or other parameters such as screen resolution and orientation.

Open that file now, switch to the strings.xml tab at the bottom if neces- sary, and enter this:

```
 <?xml version="1.0" encoding="utf-8"?>
<!--

-->
<resources>
   <string name="app_name">Sudoku</string>
   <string name="main_title">Android Sudoku</string>
   <string name="continue_label">Continue</string>
   <string name="new_game_label">New Game</string>
   <string name="about_label">About</string>
   <string name="exit_label">Exit</string>

   <string name="settings_label">Settings...</string>
   <string name="settings_title">Sudoku settings</string>
   <string name="settings_shortcut">s</string>
   <string name="music_title">Music</string>
   <string name="music_summary">Play background music</string>
   <string name="hints_title">Hints</string>
   <string name="hints_summary">Show hints during play</string>
   <string name="new_game_title">Difficulty</string>
   <string name="easy_label">Easy</string>
   <string name="medium_label">Medium</string>
   <string name="hard_label">Hard</string>

   <string name="about_title">About Android Sudoku</string>
   <string name="about_text">\
Sudoku is a logic-based number placement puzzle.
Starting with a partially completed 9x9 grid, the
objective is to fill the grid so that each
row, each column, and each of the 3x3 boxes
(also called <i>blocks</i>) contains the digits
1 to 9 exactly once.
</string>
</resources>
```

Save strings.xml so Eclipse will rebuild the project. Run you project and see the output.

The current screen is readable, but it could use some cosmetic changes. Lets make the title text larger and centered, make the buttons smaller, and use a different background color. Heres the color definition, which you should put in **res/values/colors.xml:**

```
 <?xml version="1.0" encoding="utf-8"?>
<!--
 !
-->
<resources>
   <color name="background">#3500ffff</color>
</resources>
```

The Updated main.xml file

```
 <?xml version="1.0" encoding="utf-8"?>

<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:background="@color/background"
    android:layout_height="fill_parent"
    android:layout_width="fill_parent"
    android:padding="30dip"
    android:orientation="horizontal">
   <LinearLayout
       android:orientation="vertical"
       android:layout_height="wrap_content"
       android:layout_width="fill_parent"
       android:layout_gravity="center">
      <TextView
          android:text="@string/main_title"
          android:layout_height="wrap_content"
          android:layout_width="wrap_content"
          android:layout_gravity="center"
          android:layout_marginBottom="25dip"
          android:textSize="24.5sp" />
      <Button
          android:id="@+id/continue_button"
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="@string/continue_label" />
      <Button
          android:id="@+id/new_button"
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="@string/new_game_label" />
      <Button
          android:id="@+id/about_button"
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="@string/about_label" />
      <Button
          android:id="@+id/exit_button"
          android:layout_width="fill_parent"
          android:layout_height="wrap_content"
          android:text="@string/exit_label" />
   </LinearLayout>
</LinearLayout>
```

In this version, we introduce a new syntax, @+id/resid. Instead of referring to a resource ID defined somewhere else, this is how you create a new resource ID to which others can refer. For example,

```
@+id/about_button
```

defines the ID for the About button, which well use later to make something happen when the user presses that button.

This new screen looks good in portrait mode , but how about landscape mode? The user can switch modes at any time, for example, by flipping out the keyboard or turning the phone on its side, so you need to handle that.

As a test, try switching the emulator to landscape mode (Ctrl+F11) or (the 7 or 9 key on the keypad). You will noticThee that the Exit button runs off the bottom of the screen

To fix this we use different layout for Landscape mode

Create a file called **res/layout-land/main.xml** (note the -land suffix) that contains the following layout:

```
 <?xml version="1.0" encoding="utf-8"?>

<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:background="@color/background"
    android:layout_height="fill_parent"
    android:layout_width="fill_parent"
    android:padding="15dip"
    android:orientation="horizontal">
   <LinearLayout
       android:orientation="vertical"
       android:layout_height="wrap_content"
       android:layout_width="fill_parent"
       android:layout_gravity="center"
       android:paddingLeft="20dip"
       android:paddingRight="20dip">
      <TextView
          android:text="@string/main_title"
          android:layout_height="wrap_content"
          android:layout_width="wrap_content"
          android:layout_gravity="center"
          android:layout_marginBottom="20dip"
          android:textSize="24.5sp" />
      <TableLayout
          android:layout_height="wrap_content"
          android:layout_width="wrap_content"
          android:layout_gravity="center"
          android:stretchColumns="*">
         <TableRow>
            <Button
               android:id="@+id/continue_button"
               android:text="@string/continue_label" />
            <Button
               android:id="@+id/new_button"
               android:text="@string/new_game_label" />
         </TableRow>
         <TableRow>
            <Button
               android:id="@+id/about_button"
               android:text="@string/about_label" />
            <Button
               android:id="@+id/exit_button"
               android:text="@string/exit_label" />
         </TableRow>
      </TableLayout>
   </LinearLayout>
</LinearLayout>
```

This uses a TableLayout to create two columns of buttons. Now run the program again Even in landscape mode, all the buttons are visible. You can use resource suffixes to specify alternate versions of any resources, not just the layout. For example, you can use them to provide localized text strings in different languages. Androids screen density support depends heavily on these resource suffixes All Screens Great and Small

11.4.9.5 Events

Implementing an About Box When the user selects the About button, meaning that either they touch it (if they have a touch screen) or they navigate to it with the D-pad (directional pad) or trackball and press the selection button, we want to pop up a window with some information about Sudoku.

After scrolling through the text, the user can press the Back button to dismiss the window. We can accomplish this in several ways:

1. Define a new Activity, and start it.
2. Use the AlertDialog class, and show it.
3. Subclass Androids Dialog class, and show that.

For this example, lets define a new activity. Like the main Sudoku activity, the About activity will need a layout file. We will name it **res/layout/about.xml:**

```
 <?xml version="1.0" encoding="utf-8"?>

<ScrollView
   xmlns:android="http://schemas.android.com/apk/res/android"
   android:layout_width="fill_parent"
   android:layout_height="fill_parent"
   android:padding="10dip">
   <TextView
      android:id="@+id/about_content"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@string/about_text" />
</ScrollView>
```

We need only one version of this layout because it will look fine in both portrait and landscape modes.

The About activity should be defined in About.java. All it needs to do is override onCreate( ) and call setContentView( ). To create a new class in Eclipse, use File ¿ New ¿ Class. Specify the following:

```
Source␣folder:␣Sudoku/src
Package:␣za.ac.cput.sudoku
Name:␣About
```

Edit the class so it looks like this:

```
package za.ac.cput.sudoku;

import android.app.Activity;
import android.os.Bundle;

public class About extends Activity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.about);
    }
}
```

Next we need to wire all this up to the About button in the Sudoku class. Start by adding a few imports that well need to Sudoku.java:

```
import android.content.Intent;
import android.view.View;
import android.view.View.OnClickListener;
```

In the onCreate( ) method, add code to call findViewById( ) to look up an Android view given its resource ID, and add code to call setOnClickListener( ) to tell Android which object to fire when the user touches or clicks the view:

```
@Override
public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.main);
      // Set up click listeners for all the buttons
      View continueButton = findViewById(R.id.continue_button);
      continueButton.setOnClickListener(this);
      View newButton = findViewById(R.id.new_button);
      newButton.setOnClickListener(this);
      View aboutButton = findViewById(R.id.about_button);
      aboutButton.setOnClickListener(this);
      View exitButton = findViewById(R.id.exit_button);
      exitButton.setOnClickListener(this);
}
```

The setOnClickListener( ) method needs to be passed an object that implements the OnClickListener Java interface. Were passing it the this variable, so we had better make the current class (Sudoku) implement that interface, or well get a compiler error. OnClickListener has one method in it called onClick( ), so we have to add that method to our class as well

Below is the full listing of Soduko

```

package za.ac.cput.sudoku;

import android.app.Activity;
import android.app.AlertDialog;
import android.content.DialogInterface;
import android.content.Intent;
import android.os.Bundle;
import android.util.Log;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;
import android.view.View;
import android.view.View.OnClickListener;


public class Sudoku extends Activity implements OnClickListener {
       private static final String TAG = "Sudoku";

       /** Called when the activity is first created. */
       @Override
       public void onCreate(Bundle savedInstanceState) {
          super.onCreate(savedInstanceState);
          setContentView(R.layout.main);

          // Set up click listeners for all the buttons
          View continueButton = findViewById(R.id.continue_button);
          continueButton.setOnClickListener(this);
          View newButton = findViewById(R.id.new_button);
          newButton.setOnClickListener(this);
          View aboutButton = findViewById(R.id.about_button);
          aboutButton.setOnClickListener(this);
          View exitButton = findViewById(R.id.exit_button);
          exitButton.setOnClickListener(this);
       }

       // ...
       public void onClick(View v) {
          switch (v.getId()) {
          case R.id.about_button:
             Intent i = new Intent(this, About.class);
             startActivity(i);
             break;
```

```
         // More buttons go here (if any) ...
         case R.id.new_button:
            openNewGameDialog();
            break;
         case R.id.exit_button:
            finish();
            break;
         }
      }

      @Override
      public boolean onCreateOptionsMenu(Menu menu) {
         super.onCreateOptionsMenu(menu);
         MenuInflater inflater = getMenuInflater();
         inflater.inflate(R.menu.menu, menu);
         return true;
      }

      @Override
      public boolean onOptionsItemSelected(MenuItem item) {
         switch (item.getItemId()) {
         case R.id.settings:
            startActivity(new Intent(this, Prefs.class));
            return true;
         // More items go here (if any) ...
         }
         return false;
      }

      /** Ask the user what difficulty level they want */
      private void openNewGameDialog() {
         new AlertDialog.Builder(this)
              .setTitle(R.string.new_game_title)
              .setItems(R.array.difficulty,
               new DialogInterface.OnClickListener() {
                  public void onClick(DialogInterface dialoginterface,
                        int i) {
                     startGame(i);
                  }
               })
              .show();
      }

      /** Start a new game with the given difficulty level */
      private void startGame(int i) {
         Log.d(TAG, "clicked on " + i);
         // Start game here...
      }
   }
```

To start an activity in Android, we first need to create an instance of the Intent class. There are two kinds of intents: public (named) intents that are registered with the system and can be called from any appli- cation and private (anonymous) intents that are used within a single application. Every activity needs to be declared in **AndroidManifest.xml**. To do that, double-click the file to open it, switch to XML mode if necessary by selecting the AndroidManifest.xml tab at the bottom, and add a new ¡activity¿ tag after the closing tag of the first one:

```
 <activity android:name=".About"
android:label="@string/about_title" >
</activity>
```

Now run your application.

11.4.9.6 Code Explanation

Applying theme

A theme is a collection of styles that override the look and feel of Android widgets. Themes were inspired by Cascading Style Sheets (CSS) used for web pagesthey separate the content of a screen and its presentation or style. Android is packaged with several themes that you can reference by name or you can make up your own theme by subclassing existing ones and overriding their default values.

We could define our own custom theme in **res/values/styles.xml**, but for this example well just take advantage of a predefined one. To use it, we opened the **AndroidManifest.xml** editor , and changed the definition of the About activity so it has a theme property. as shown below

```
<activity android:name=".About"
android:label="@string/about_title"
android:theme="@android:style/Theme.Dialog" >
</activity>
```

The @android: prefix in front of the style name means this is a reference to a resource defined by Android, not one that is defined in your program.

Adding a Menu

Android supports two kinds of menus. First, there is the menu you get when you press the physical Menu button. Second, there is a context menu that pops up when you press and hold your finger on the screen (or press and hold the trackball or the D-pad center button).

In this tutorial we did the first one and we first we defined a few strings in **/res/values/strings.xml**

```
    <string name="settings_label">Settings...</string>
    <string name="settings_title">Sudoku settings</string>
    <string name="settings_shortcut">s</string>
    <string name="music_title">Music</string>
    <string name="music_summary">Play background music</string>
    <string name="hints_title">Hints</string>
    <string name="hints_summary">Show hints during play</string>
```

Then we defined the menu using XML in **res/menu/menu.xml:**

```
 <?xml version="1.0" encoding="utf-8"?>

<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/settings"
        android:title="@string/settings_label"
        android:alphabeticShortcut="@string/settings_shortcut" />
</menu>
```

Next we need to modified the Sudoku class to bring up the menu we just defined by overriding the **Sudoku.onCreateOptionsMenu( )** method:

```
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;

......

 @Override
public boolean onCreateOptionsMenu(Menu menu) {
super.onCreateOptionsMenu(menu);
MenuInflater inflater = getMenuInflater();
inflater.inflate(R.menu.menu, menu);
return true;
}
```

getMenuInflater( ) returns an instance of MenuInflater that we use to read the menu definition from XML and turns it into a real view. When the user selects any menu item, **onOptionsItemSelected( )** will be called. Heres the definition for that method:

```
 @Override
public boolean onOptionsItemSelected(MenuItem item) {
switch (item.getItemId()) {
case R.id.settings:
startActivity(new Intent(this, Prefs.class));
return true;
// More items go here (if any) ...
}
return false;
}
```

Prefs is a class that we defined that displays all our preferences and allows the user to change them.

Adding Settings

Android provides a nice facility for defining what all your program preferences are and how to display them using almost no code. You define the preferences in a resource file called **res/xml/settings.xml** shown below

```
 <?xml version="1.0" encoding="utf-8"?>

<PreferenceScreen
    xmlns:android="http://schemas.android.com/apk/res/android">
   <CheckBoxPreference
       android:key="music"
       android:title="@string/music_title"
       android:summary="@string/music_summary"
       android:defaultValue="true" />
   <CheckBoxPreference
       android:key="hints"
       android:title="@string/hints_title"
       android:summary="@string/hints_summary"
       android:defaultValue="true" />
</PreferenceScreen>
```

The Sudoku program has two settings: one for background music and one for displaying hints. The keys are constant strings that will be used under the covers in Androids preferences database. Next defined the **Prefs** class, and made it extend **Preference-Activity** class as shown below:

```
 package za.ac.cput.sudoku;

import android.os.Bundle;
import android.preference.PreferenceActivity;

public class Prefs extends PreferenceActivity {
   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      addPreferencesFromResource(R.xml.settings);
   }
}
```

The **addPreferencesFromResource( )** method reads the settings definition from XML and inflates it into views in the current activity. All the heavy lifting takes place in the **PreferenceActivity class**.

We made sure we registered the **Prefs** activity in **AndroidManifest.xml:** as shown below

```
 <activity android:name=".Prefs"
android:label="@string/settings_title" >
</activity>
```

To see the effects of settings rerun Sudoku, press the Menu key, select the Settings... item, and the Sudoku settings page appears . Try changing the values there and exiting the program, and then come back in and make sure theyre all still set.

Starting a New Game

We added a start menu so that when the user selects New Game, we want to pop up a dialog box asking them to select between three difficulty levels. Selecting from a list of things is easy to do in Android.

We added a few more strings in **res/values/strings.xml**:

```
  <string name="new_game_title">Difficulty</string>
   <string name="easy_label">Easy</string>
   <string name="medium_label">Medium</string>
   <string name="hard_label">Hard</string>
```

Created the list of difficulties as an array resource in **res/values/arrays.xml**:

```
 <?xml version="1.0" encoding="utf-8"?>
<resources>
   <array name="difficulty">
      <item>@string/easy_label</item>
      <item>@string/medium_label</item>
      <item>@string/hard_label</item>
   </array>
</resources>
```

Then added a few more imports in the Sudoku class:

```
import android.app.AlertDialog;
import android.content.DialogInterface;
import android.util.Log;
```

And then added code in the switch statement of the **onClick( )** method to handle clicking the New Game button:

```
 case R.id.new_button:
openNewGameDialog();
break;
```

The **openNewGameDialog( )** method takes care of creating the user interface for the difficulty list as show below.

```
  private static final String TAG = "Sudoku";

...
```

```
/** Ask the user what difficulty level they want */
       private void openNewGameDialog() {
          new AlertDialog.Builder(this)
               .setTitle(R.string.new_game_title)
               .setItems(R.array.difficulty,
                new DialogInterface.OnClickListener() {
                   public void onClick(DialogInterface dialoginterface,
                         int i) {
                      startGame(i);
                   }
                })
               .show();
       }

       /** Start a new game with the given difficulty level */
       private void startGame(int i) {
          Log.d(TAG, "clicked on " + i);
          // Start game here...
       }
```

The setItems( ) method takes two parameters: the resource ID of the item list and a listener that will be called when one of the items is selected.

Exiting the Game

This game doesnt really need an Exit button, because the user can just press the Back key or the Home key to do something else. But I wanted to add one to show you how to terminate an activity. Add this to the switch statement in the **onClick( )** method:

```
 case R.id.exit_button:
  finish();
  break;
```

When the Exit button is selected, we call the finish( ) method. This shuts down the activity and returns control to the next activity on the Android application stack (usually the Home screen).

11.4.9.7 Adding 2D Graphics to Sudoku

Starting the Game

First we need to fill in the code that starts the game. **startGame( )** takes one parameter, the index of the difficulty name selected from the list. Heres the new definition:

```
 private void startGame(int i) {
Log.d(TAG, "clicked on " + i);
Intent intent = new Intent(Sudoku.this, Game.class);
intent.putExtra(Game.KEY_DIFFICULTY, i);
startActivity(intent);
}
```

The game part of Sudoku will be another activity called **Game**, so we create a new intent to kick it off. We place the difficulty number in an extraData area provided in the intent, and then we call the startActivity( ) method to launch the new activity.

The extraData area is a map of key/value pairs that will be passed along to the intent. The keys are strings, and the values can be any primitive type, array of primitives, Bundle, or a subclass of Serializable or Parcelable.

```
 import android.app.Activity;
import android.app.Dialog;
import android.os.Bundle;
import android.util.Log;
import android.view.Gravity;
import android.widget.Toast;

public class Game extends Activity {
   private static final String TAG = "Sudoku";

   public static final String KEY_DIFFICULTY =
      "org.example.sudoku.difficulty";
   public static final int DIFFICULTY_EASY = 0;
   public static final int DIFFICULTY_MEDIUM = 1;
   public static final int DIFFICULTY_HARD = 2;

   private int puzzle[] = new int[9 * 9];

 private PuzzleView puzzleView;


 @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      Log.d(TAG, "onCreate");

      int diff = getIntent().getIntExtra(KEY_DIFFICULTY,
            DIFFICULTY_EASY);
      puzzle = getPuzzle(diff);
      calculateUsedTiles();

      puzzleView = new PuzzleView(this);
      setContentView(puzzleView);
      puzzleView.requestFocus();
   }
   // ...
```

The **onCreate( )** method fetches the difficulty number from the intent and selects a puzzle to play. Then it creates an instance of the **PuzzleView** class, setting the **PuzzleView** as the new contents of the view. Since this is a fully customized view, it was easier to do this in code than in XML.

This is an activity, so we need to register it in AndroidManifest.xml:

```
 <activity android:name=".Game"
android:label="@string/game_title" />
```

We also need to add a few more string resources to **res/values/strings.xml**:

```
 <string name="game_title">Game</string>
<string name="no_moves_label">No moves</string>
<string name="keypad_title">Keypad</string>
```

View Class Using Java and NOT XML

Next we need to define the PuzzleView class. Instead of using an XML layout, this time lets do it entirely in Java.

```
import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Paint;
import android.graphics.Rect;
import android.graphics.Paint.FontMetrics;
import android.graphics.Paint.Style;
import android.util.Log;
import android.view.KeyEvent;
import android.view.MotionEvent;
import android.view.View;
import android.view.animation.AnimationUtils;
```

```
public class PuzzleView extends View {
   private static final String TAG = "Sudoku";


   private float width;     // width of one tile
   private float height;    // height of one tile
   private int selX;        // X index of selection
   private int selY;        // Y index of selection
   private final Rect selRect = new Rect();



   private final Game game;
   public PuzzleView(Context context) {
      super(context);
      this.game = (Game) context;
      setFocusable(true);
      setFocusableInTouchMode(true);
   }
   // ...

@Override
   protected void onSizeChanged(int w, int h, int oldw, int oldh) {
      width = w / 9f;
      height = h / 9f;
      getRect(selX, selY, selRect);
      Log.d(TAG, "onSizeChanged: width " + width + ", height "
            + height);
      super.onSizeChanged(w, h, oldw, oldh);
   }

 private void getRect(int x, int y, Rect rect) {
      rect.set((int) (x * width), (int) (y * height), (int) (x
            * width + width), (int) (y * height + height));
   }

}
```

In the constructor we keep a reference to the Game class and set the option to allow user input in the view. Inside PuzzleView, we need to implement the **onSizeChanged( )** method. This is called after the view is created

We use **onSizeChanged( )** to calculate the size of each tile on the screen (1/9th of the total view width and height). Note this is a floating-point number, so its possible that we could end up with a fractional number of pixels. **selRect** is a rectangle well use later to keep track of the selection cursor.

At this point weve created a view for the puzzle, and we know how big it is. The next step is to draw the grid lines that separate the tiles on the board.

Drawing the Board

Android calls a views onDraw( ) method every time any part of the view needs to be updated. To simplify things, onDraw( ) pretends that youre re-creating the entire screen from scratch. In reality, you may be drawing only a small portion of the view as defined by the canvas clip rectangle. Android takes care of doing the clipping for you. Start by defining a few new colors to play with in **res/values/colors.xml**:

```
 <resources>
   <color name="background">#3500ffff</color>
   <color name="puzzle_background">#ffe6f0ff</color>
   <color name="puzzle_hilite">#ffffffff</color>
   <color name="puzzle_light">#64c6d4ef</color>
   <color name="puzzle_dark">#6456648f</color>
   <color name="puzzle_foreground">#ff000000</color>
   <color name="puzzle_hint_0">#64ff0000</color>
   <color name="puzzle_hint_1">#6400ff80</color>
   <color name="puzzle_hint_2">#2000ff80</color>
   <color name="puzzle_selected">#64ff8000</color>
</resources>
```

Heres the basic outline for onDraw( ):

```
@Override
protected void onDraw(Canvas canvas) {
   // Draw the background...
   Paint background = new Paint();
   background.setColor(getResources().getColor(
         R.color.puzzle_background));
   canvas.drawRect(0, 0, getWidth(), getHeight(), background);


   // Draw the board...

   // Define colors for the grid lines
   Paint dark = new Paint();
   dark.setColor(getResources().getColor(R.color.puzzle_dark));

   Paint hilite = new Paint();
   hilite.setColor(getResources().getColor(R.color.puzzle_hilite));

   Paint light = new Paint();
   light.setColor(getResources().getColor(R.color.puzzle_light));

   // Draw the minor grid lines
   for (int i = 0; i < 9; i++) {
      canvas.drawLine(0, i * height, getWidth(), i * height,
            light);
      canvas.drawLine(0, i * height + 1, getWidth(), i * height
           + 1, hilite);
      canvas.drawLine(i * width, 0, i * width, getHeight(),
            light);
      canvas.drawLine(i * width + 1, 0, i * width + 1,
            getHeight(), hilite);
   }

   // Draw the major grid lines
   for (int i = 0; i < 9; i++) {
      if (i % 3 != 0)
         continue;
      canvas.drawLine(0, i * height, getWidth(), i * height,
            dark);
      canvas.drawLine(0, i * height + 1, getWidth(), i * height
           + 1, hilite);
      canvas.drawLine(i * width, 0, i * width, getHeight(), dark);
      canvas.drawLine(i * width + 1, 0, i * width + 1,
            getHeight(), hilite);
   }



   // Draw the numbers...

   // Define color and style for numbers
   Paint foreground = new Paint(Paint.ANTI_ALIAS_FLAG);
   foreground.setColor(getResources().getColor(
         R.color.puzzle_foreground));
   foreground.setStyle(Style.FILL);
   foreground.setTextSize(height * 0.75f);
   foreground.setTextScaleX(width / height);
   foreground.setTextAlign(Paint.Align.CENTER);

   // Draw the number in the center of the tile
   FontMetrics fm = foreground.getFontMetrics();
   // Centering in X: use alignment (and X at midpoint)
   float x = width / 2;
   // Centering in Y: measure ascent/descent first
   float y = height / 2 - (fm.ascent + fm.descent) / 2;
   for (int i = 0; i < 9; i++) {
      for (int j = 0; j < 9; j++) {
         canvas.drawText(this.game.getTileString(i, j), i
               * width + x, j * height + y, foreground);
      }
   }



   // Draw the hints...

   // Pick a hint color based on #moves left
   Paint hint = new Paint();
   int c[] = { getResources().getColor(R.color.puzzle_hint_0),
         getResources().getColor(R.color.puzzle_hint_1),
         getResources().getColor(R.color.puzzle_hint_2), };
   Rect r = new Rect();
   for (int i = 0; i < 9; i++) {
      for (int j = 0; j < 9; j++) {
         int movesleft = 9 - game.getUsedTiles(i, j).length;
         if (movesleft < c.length) {
            getRect(i, j, r);
            hint.setColor(c[movesleft]);
            canvas.drawRect(r, hint);
         }
```

```
        }
    }



    // Draw the selection...

    Log.d(TAG, "selRect=" + selRect);
    Paint selected = new Paint();
    selected.setColor(getResources().getColor(
          R.color.puzzle_selected));
    canvas.drawRect(selRect, selected);

  }
```

The first parameter is the Canvas on which to draw. In this code, were just drawing a background for the puzzle using the puzzle background color.

The code uses three different colors for the grid lines: a light color between each tile, a dark color between the three-by-three blocks, and a highlight color drawn on the edge of each tile to make them look like they have a little depth. The order in which the lines are drawn is important, since lines drawn later will be drawn over the top of earlier lines.

Next, we need some numbers to go inside those lines.

Drawing the Numbers

The following code draws the puzzle numbers on top of the tiles. The tricky part here is getting each number positioned and sized so it goes in the exact center of its tile

```
// Draw the numbers...

    // Define color and style for numbers
    Paint foreground = new Paint(Paint.ANTI_ALIAS_FLAG);
    foreground.setColor(getResources().getColor(
          R.color.puzzle_foreground));
    foreground.setStyle(Style.FILL);
    foreground.setTextSize(height * 0.75f);
    foreground.setTextScaleX(width / height);
    foreground.setTextAlign(Paint.Align.CENTER);

    // Draw the number in the center of the tile
    FontMetrics fm = foreground.getFontMetrics();
    // Centering in X: use alignment (and X at midpoint)
    float x = width / 2;
    // Centering in Y: measure ascent/descent first
    float y = height / 2 - (fm.ascent + fm.descent) / 2;
    for (int i = 0; i < 9; i++) {
       for (int j = 0; j < 9; j++) {
          canvas.drawText(this.game.getTileString(i, j), i
                * width + x, j * height + y, foreground);
       }
    }
```

We call the **getTileString( )** method to find out what numbers to display. To calculate the size of the numbers, we set the font height to three-fourths the height of the tile, and we set the aspect ratio to be the same as the tiles aspect ratio. We cant use absolute pixel or point sizes because we want the program to work at any resolution.

To determine the position of each number, we center it in both the x and y dimensions. The x direction is easyjust divide the tile width by 2. But for the y direction, we have to adjust the starting position downward a little so that the midpoint of the tile will be the midpoint of the number instead of its baseline. We use the graphics librarys

FontMetrics class to tell how much vertical space the letter will take in total, and then we divide that in half to get the adjustment.

Handling Input

One difference in Android programmingas opposed to, say, iPhone programmingis that Android phones come in many shapes and sizes and have a variety of input methods. They might have a keyboard, a D-pad, a touch screen, a trackball, or some combination of these. A good Android program, therefore, needs to be ready to support whatever input hardware is available, just like it needs to be ready to support any screen resolution.

First were going to implement a little cursor that shows the player which tile is currently selected. The selected tile is the one that will be modified when the player enters a number. This code will draw the selection in **onDraw( ):**

```
// Draw the selection...

     Log.d(TAG, "selRect=" + selRect);
     Paint selected = new Paint();
     selected.setColor(getResources().getColor(
           R.color.puzzle_selected));
     canvas.drawRect(selRect, selected);
```

We use the selection rectangle calculated earlier in **onSizeChanged( )** to draw an alpha-blended color on top of the selected tile.

Next we provide a way to move the selection by overriding the **onKeyDown( )** method:

```
 @Override
  public boolean onKeyDown(int keyCode, KeyEvent event) {
     Log.d(TAG, "onKeyDown: keycode=" + keyCode + ", event="
           + event);
     switch (keyCode) {
     case KeyEvent.KEYCODE_DPAD_UP:
        select(selX, selY - 1);
        break;
     case KeyEvent.KEYCODE_DPAD_DOWN:
        select(selX, selY + 1);
        break;
     case KeyEvent.KEYCODE_DPAD_LEFT:
        select(selX - 1, selY);
        break;
     case KeyEvent.KEYCODE_DPAD_RIGHT:
        select(selX + 1, selY);
        break;


     case KeyEvent.KEYCODE_0:
     case KeyEvent.KEYCODE_SPACE: setSelectedTile(0); break;
     case KeyEvent.KEYCODE_1:     setSelectedTile(1); break;
     case KeyEvent.KEYCODE_2:     setSelectedTile(2); break;
     case KeyEvent.KEYCODE_3:     setSelectedTile(3); break;
     case KeyEvent.KEYCODE_4:     setSelectedTile(4); break;
     case KeyEvent.KEYCODE_5:     setSelectedTile(5); break;
     case KeyEvent.KEYCODE_6:     setSelectedTile(6); break;
     case KeyEvent.KEYCODE_7:     setSelectedTile(7); break;
     case KeyEvent.KEYCODE_8:     setSelectedTile(8); break;
     case KeyEvent.KEYCODE_9:     setSelectedTile(9); break;
     case KeyEvent.KEYCODE_ENTER:
     case KeyEvent.KEYCODE_DPAD_CENTER:
        game.showKeypadOrError(selX, selY);
        break;


     default:
        return super.onKeyDown(keyCode, event);
     }
     return true;
```

If the user has a directional pad (D-pad) and they press the up, down, left, or right button, we call **select( )** to move the selection cursor in that direction.

How about a trackball? We could override the **onTrackballEvent( )** method, but it turns out that if you dont handle trackball events, Android will translate them into D-pad events automatically. Therefore, we can leave it out for this example.

Inside the **select( )** method, we calculate the new x and y coordinates of the selection and then use getRect( ) again to calculate the new selection rectangle.

```
private void select(int x, int y) {
    invalidate(selRect);
    selX = Math.min(Math.max(x, 0), 8);
    selY = Math.min(Math.max(y, 0), 8);
    getRect(selX, selY, selRect);
    invalidate(selRect);
}
```

Notice the two calls to **invalidate( )**. The first one tells Android that the area covered by the old selection rectangle needs to be redrawn. The second **invalidate( )** call says that the new selection area needs to be redrawn too. We dont actually draw anything here.

This is an important point: never call any drawing functions except in the **onDraw( )** method. Instead, you use the **invalidate( )** method to mark rectangles as dirty. The window manager will combine all the dirty rectangles at some point in the future and call **onDraw( )** again for you. The dirty rectangles become the clip region, so screen updates are optimized to only those areas that change.

Now lets provide a way for the player to enter a new number on the selected tile.

Entering Numbers

To handle keyboard input, we just add a few more cases to the **onKey- Down( )** method for the numbers 0 through 9 (0 or space means erase the number).

```
case KeyEvent.KEYCODE_0:
    case KeyEvent.KEYCODE_SPACE: setSelectedTile(0); break;
    case KeyEvent.KEYCODE_1:     setSelectedTile(1); break;
    case KeyEvent.KEYCODE_2:     setSelectedTile(2); break;
    case KeyEvent.KEYCODE_3:     setSelectedTile(3); break;
    case KeyEvent.KEYCODE_4:     setSelectedTile(4); break;
    case KeyEvent.KEYCODE_5:     setSelectedTile(5); break;
    case KeyEvent.KEYCODE_6:     setSelectedTile(6); break;
    case KeyEvent.KEYCODE_7:     setSelectedTile(7); break;
    case KeyEvent.KEYCODE_8:     setSelectedTile(8); break;
    case KeyEvent.KEYCODE_9:     setSelectedTile(9); break;
    case KeyEvent.KEYCODE_ENTER:
    case KeyEvent.KEYCODE_DPAD_CENTER:
        game.showKeypadOrError(selX, selY);
        break;
```

To support the D-pad, we check for the Enter or center D-pad button in **onKeyDown( )** and have it pop up a keypad that lets the user select which number to place.

For touch, we override the **onTouchEvent( )** method and show the same keypad, which will be defined later:

```
   @Override
   public boolean onTouchEvent(MotionEvent event) {
      if (event.getAction() != MotionEvent.ACTION_DOWN)
         return super.onTouchEvent(event);

      select((int) (event.getX() / width),
            (int) (event.getY() / height));
      game.showKeypadOrError(selX, selY);
      Log.d(TAG, "onTouchEvent: x " + selX + ", y " + selY);
      return true;
   }

............
  public void setSelectedTile(int tile) {
      if (game.setTileIfValid(selX, selY, tile)) {
         invalidate();// may change hints
      } else {
         // Number is not valid for this tile

         Log.d(TAG, "setSelectedTile: invalid: " + tile);

         startAnimation(AnimationUtils.loadAnimation(game,
               R.anim.shake));


      }
...............
```

Ultimately, all roads will lead back to a call to setSelectedTile( ) to change the number on a tile.

Creating the Keypad

The keypad is handy for phones that dont have keyboards. It displays a grid of the numbers 1 through 9 in an activity that appears on top of the puzzle. The whole purpose of the keypad dialog box is to return a number selected by the player

Implementing the Game Logic

The rest of the code in **Game.java** concerns itself with the logic of the game, in particular with determining which are and arent valid moves according to the rules. The setTileIfValid( ) method is a key part of that. Given an x and y position and the new value of a tile, it changes the tile only if the value provided is valid.

To detect valid moves, we create an array for every tile in the grid. For each position, it keeps a list of filled-in tiles that are currently visible from that position. If a number appears on the list, then it wont be valid for the current tile. The **getUsedTiles( )** method retrieves that list for a given tile position.

The array of used tiles is somewhat expensive to compute, so we cache the array and recalculate it only when necessary by calling calculate **UsedTiles( )**

calculateUsedTiles( ) simply calls **calculateUsedTiles(x, y)** on every position in the nine-by-nine grid.

THE SOURCE CODE FOR THE GAME IS ON WEBCT IN the **sample code** Folder

11.5 Blackberry Platform

BlackBerry applications are written in Java Micro Edition (Java ME) J2ME. In this section , youll learn about the basics of BlackBerry application development by creating a simple Hello World application in the BlackBerry JDE Plug-in for Eclipse. Our application is a simple BlackBerry application with a single screen that will display Hello World. Well walk through creating the workspace and project, creating and building out the necessary classes, and compiling and running on a simulator. Then, well add a few extra bits of polish, like a proper application name, and version information.

11.5.1 Creating the Project

When you start Eclipse, youre asked for a workspace location, which can be any directory. Just pick the default. To create a new BlackBerry project, click the File menu, and choose **New** then **Project.** In the **New Project** dialog, select **BlackBerry Project** from the BlackBerry **folder**

Click Next; name your project HelloWorld, and click Finish. Your Eclipse workspace should contain a single project in the Package Explorer on the left-hand side. When expanded, the package should contain a folder named src, which is where all our source files will reside, and a reference to NET RIM BLACKBERRY, which is the BlackBerry runtime library containing the BlackBerry API

11.5.1.1 Creating the Main Application Class

Right-click the HelloWorld project icon in the Package Explorer, and from the pop-up menu, select New then Class. In the dialog, type the following values:

- Package: za.ac.cput
- Name: HelloWorldApp
- Superclass: net.rim.device.api.ui.UiApplication

Under Which method stubs would you like to create? , make sure the first two check boxes for generating a main method and constructors are checked (the third box can be checked or not, there are no abstract methods in UiApplication, so it wont make a difference). Everything else can be left at the default.

Youll get the following source code:

```
 package za.ac.cput;
import net.rim.device.api.ui.UiApplication;
public class HelloWorldApp extends UiApplication {
public HelloWorldApp() {
// TODO Auto-generated constructor stub
}
/**
* @param args
*/
public static void main(String[] args) {
// TODO Auto-generated method stub
}
}
```

There are even TODO markers where we have to write our logic. Well do that, but first, lets create the main screen class

11.5.2 Creating the Main Screen Class

Click New Class again (or if you right-click the package in the tree view and select New then Class, you wont have to reenter the package name). Fill in the following values:

- Package: za.ac.cput
- Name: HelloWorldMainScreen
- Superclass: net.rim.device.api.ui.container.MainScreen

Leave all other values as default, and click Finish to create the following source code:

```
package za.ac.cput;
import net.rim.device.api.ui.container.MainScreen;
public class HelloWorldMainScreen extends MainScreen {
}
```

Filling in the Hello World Classes

Now, well fill in the logic for both of our classes as in the previous sections (the code is repeated here for your convenience). First, add the code for HelloWorldApp.java

```
public class HelloWorldApp extends UiApplication {
    public HelloWorldApp() {
        HelloWorldMainScreen mainScreen = new HelloWorldMainScreen();
        pushScreen(mainScreen);
    }

    /**
     * @param args
     */
    public static void main(String[] args) {
        HelloWorldApp app = new HelloWorldApp();
        app.enterEventDispatcher();
    }
}
```

Then, add the following code for HelloWorldMainScreen.java:

```
import net.rim.device.api.ui.component.LabelField;
import net.rim.device.api.ui.container.MainScreen;

public class HelloWorldMainScreen extends MainScreen {
    public HelloWorldMainScreen() {
        LabelField labelField = new LabelField("Hello World");
        add(labelField);
    }
}
```

Running the Simulator

The application is automatically built and deployed when we launch the simulator (in fact, with Eclipse, the Java code is compiled whenever you make any change, which makes spotting errors easy). Running the simulator involves an extra step or two, because you have to create a debug configuration. The advantage of this is that you can create multiple device configurations for different simulators and quickly select whichever one you need.

Click the arrow next to the debug icon in the Eclipse toolbar, and select Debug Configurations

The Debug Configurations dialog lets you set up different configurations, which may be different simulators or actual devices. Each configuration can have different debug parameters, and as you develop applications, youll likely end up with a few different configurations for debugging different operating system versions, screen sizes, and so on. Feel free to explore these options at any time.

For now, select the BlackBerry Simulator icon on the left side, and click the New button on the toolbar in the dialog window

Well keep all the defaults, so just click the Debug button at the bottom of the dialog , and the simulator will launch with your application deployed. From this point on, you can access your debug configuration directly from the Debug dropdown menu in the main Eclipse toolbar by clicking the downward-facing arrow next to the debug icon.

11.5.2.1 The BlackBerry Application Life Cycle

BlackBerry applications behave very much like ordinary desktop applications. Specifically, the main method for a BlackBerry application is identical to the main method for a Java SE application. While there are exceptions for specific needs, almost all BlackBerry applications will follow the same life cycle of a java application.

11.5.2.2 Starting the Application

An application is generally started in one of three ways:

- The user clicks the applications icon on the BlackBerry home screen.
- The application is an automatically starting application and runs when the device is turned on or after it reboots.
- The application is run by another application.

In all cases, the main method is the first entry point for your application. The BlackBerry device will create a process, which will call that method. Whenever the main method exits, the process is terminated and your application exits. This means that if you want your application to do anything, youd better do it in that main method.

The main method takes an array of java.lang.String objects as parameters. For the most part, this array is empty, but parameters can be passed in if you define them in the project properties or if theyre passed by another process that is starting your application.

11.5.2.3 Creating the Application

All BlackBerry applications that want to present a user interface to the user must extend **UiApplication**. You can **only create one** instance of **UiApplication** for any application process; the BlackBerry runtime will throw an exception if you try to instantiate a second one.

Even applications with no user interface must extend **net.rim.device.api.system.Application**, but those types of applications are outside the scope of this book. You can always access your application instance using the static method **UiApplication.getUiApplication()**. This actually returns an instance of your application class, so from anywhere in Hello World, the following is allowed:

HelloWorldApp helloWorld = (HelloWorldApp)UiApplication.getUiApplication();

11.5.2.4 Invoking the Event Thread

The event thread is started for you by the BlackBerry operating system, but it doesnt start processing events and drawing the UI until you explicitly tell it to. You do this with the **UiApplication.enterEventDispatcher()** call that you saw in the example. Once this method is called, the thread that entered into the main method passes from your direct control and takes up the task of listening for user interface input and drawing the user interface to the screen. Youll still get a chance to do work on the thread, but for the most part, its activities are scheduled by the BlackBerry operating system. **enterEventDispatcher** wont return for the entire life cycle of your application, so if theres anything your main thread must do before calling this (for example, some types of initialization) you have only one chance.

11.5.2.5 Processing Events

The application responds to keyboard input, trackball, or touch screen movements and clicks and to other events like system messages.

11.5.2.6 Exiting the Application

Generally, a BlackBerry application exits when the last screen is removed from the display stack (by closing it). You may have noticed the **System.exit()** method, which will exit the application, but its recommended to avoid this and properly clean up the application on exiting by closing all screens instead. When the application exits, all application state will be cleaned up, and the next time the user clicks the application icon the main method will be called again with a new process.

11.5.3 Threading and the Event Thread

The BlackBerry UI API is single-threaded. This means that all UI updates and events are handled by the same threador more precisely, must be done while holding the event lock, which most of the time is held by the UI thread. It also has a couple of implications for BlackBerry applications: other threads cant directly access the UI without explicitly acquiring the event lock (an exception will be thrown if you try), and if you perform an operation on the event thread that takes a long time, the entire user interface will pause while that operation is taking place.

The message to take away from all this is to get comfortable with using at least one or two other threads in your applications.

11.6 IphoneIpad Platform

Forget it ..licence restrictions apply and we need apple machines running apple crap... I hope Apple burns in software hell.

Appendix A

Appendix Title Here

Write your Appendix content here.