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

February 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.

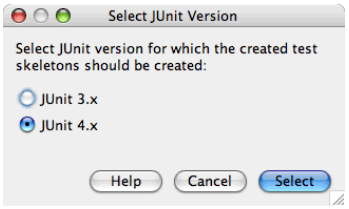
You will use the IDE's create test skeletons based on the classes in your project. The first time that you use the IDE to create some test skeletons for you, the IDE prompts you to choose the JUnit version.

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.

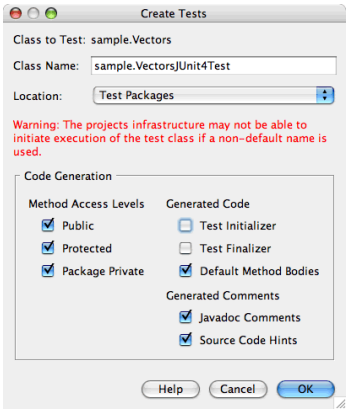
2. Select JUnit 4.x in the Select JUnit Version dialog box.



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



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**$EXPECTED_RESULT, ACTUAL_RESULT$ 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.

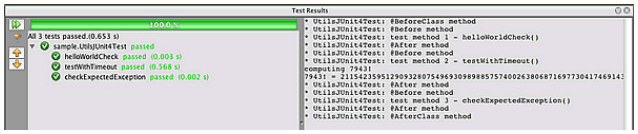
Now that you have written the tests you can run the test and see the test output in the JUnit Test Results window.

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.



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 {
}
```

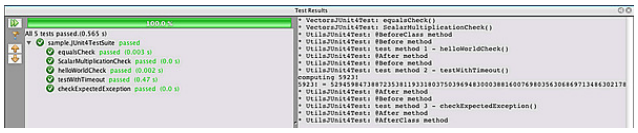
When you run the test suite the IDE will run the test classes UtilsJUnit4Test and VectorsJUnit4Test in the order they are listed.

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.



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.

```
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.

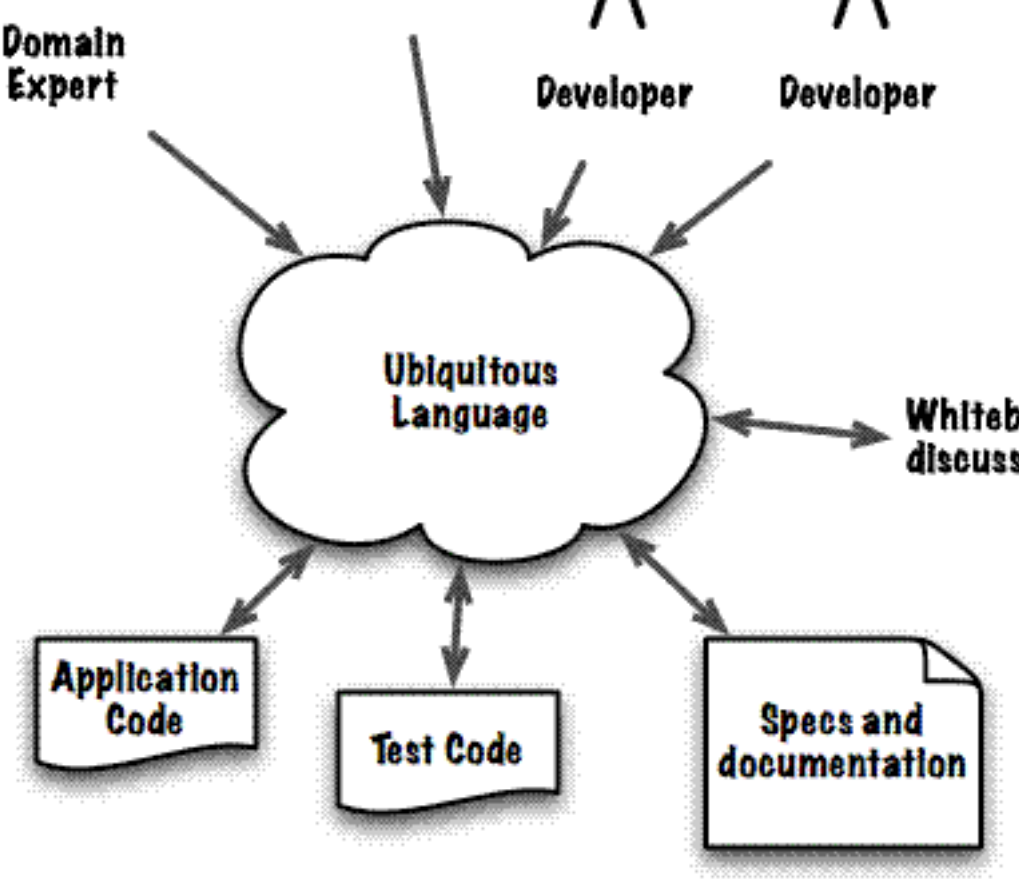


FIGURE 4.1: The Ubiquitous Language should be the only language used to express a model. Everybody in the team should be able to agree on every specific term without ambiguities and no translation should be needed

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.

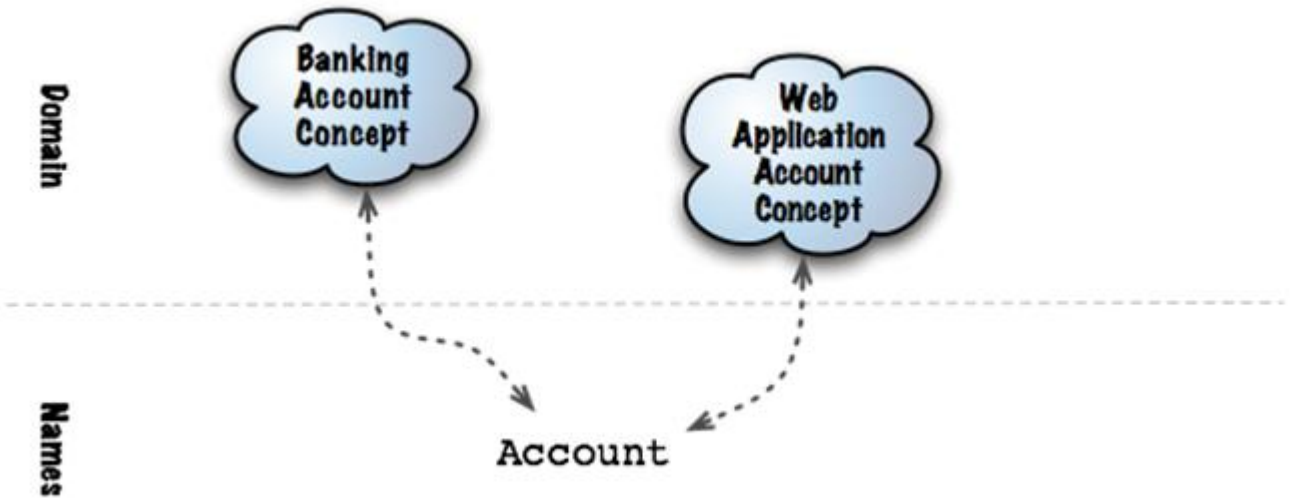


FIGURE 4.2: A somewhat trivial case of ambiguity: the term Account might mean something very different according to the context it's used in

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

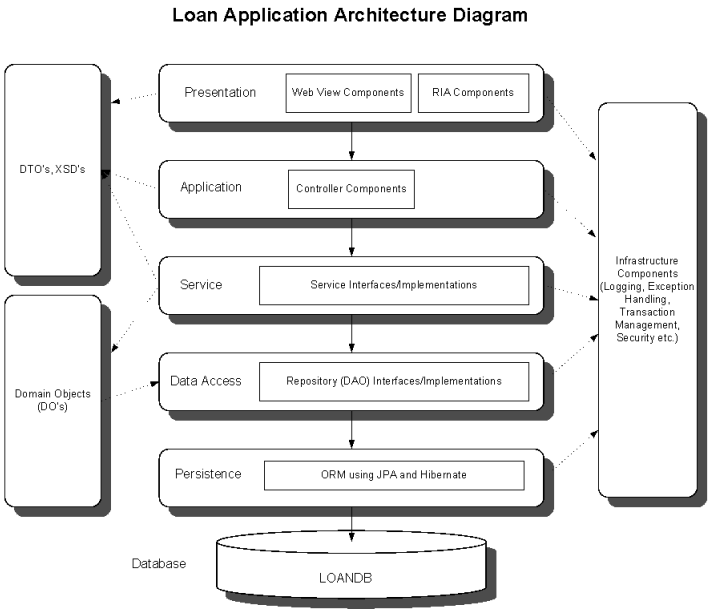


FIGURE 4.3: The Sample Layers in Domain Driven Design

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.

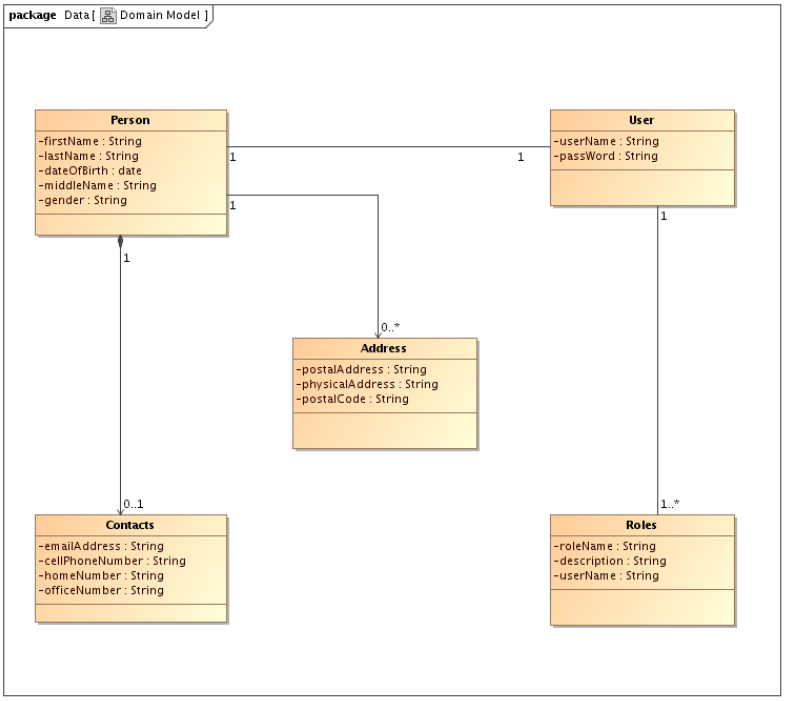


FIGURE 4.4: Domain Model for you to start with

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.

Appendix A

Appendix Title Here

Write your Appendix content here.