10 Classes

with Jeff Langr

So far in this book we have focused on how to write lines and blocks of code well. We have delved into proper composition of functions and how they interrelate. But for all the attention to the expressiveness of code statements and the functions they comprise, we still don't have clean code until we've paid attention to higher levels of code organization. Let's talk about clean classes.

Class Organization

Following the standard Java convention, a class should begin with a list of variables. Public static constants, if any, should come first. Then private static variables, followed by private instance variables. There is seldom a good reason to have a public variable.

Public functions should follow the list of variables. We like to put the private utilities called by a public function right after the public function itself. This follows the stepdown rule and helps the program read like a newspaper article.

Encapsulation

We like to keep our variables and utility functions private, but we're not fanatic about it. Sometimes we need to make a variable or utility function protected so that it can be accessed by a test. For us, tests rule. If a test in the same package needs to call a function or access a variable, we'll make it protected or package scope. However, we'll first look for a way to maintain privacy. Loosening encapsulation is always a last resort.

Classes Should Be Small!

The first rule of classes is that they should be small. The second rule of classes is that they should be smaller than that. No, we're not going to repeat the exact same text from the *Functions* chapter. But as with functions, smaller is the primary rule when it comes to designing classes. As with functions, our immediate question is always "How small?"

With functions we measured size by counting physical lines. With classes we use a different measure. We count *responsibilities*.[1]

Listing 10-1 outlines a class, `SuperDashboard`, that exposes about 70 public methods. Most developers would agree that it's a bit too super in size. Some developers might refer to `SuperDashboard` as a "God class."

Listing 10-1 `Too Many Responsibilities`

```
public class SuperDashboard extends JFrame implements MetaDataUser
  public String getCustomizerLanguagePath()
  public void setSystemConfigPath(String systemConfigPath)
  public String getSystemConfigDocument()
  public void setSystemConfigDocument(String systemConfigDocument)
  public boolean getGuruState()
```

public boolean getNoviceState()
public boolean getOpenSourceState()
public void showObject(MetaObject object)
public void showProgress(String s)
public boolean isMetadataDirty()
public void setIsMetadataDirty(boolean isMetadataDirty)
public Component getLastFocusedComponent()
public void setLastFocused(Component lastFocused)
public void setMouseSelectState(boolean isMouseSelected)
public boolean isMouseSelected()
public LanguageManager getLanguageManager()
public Project getProject()
public Project getFirstProject()
public Project getLastProject()
public String getNewProjectName()
public void setComponentSizes(Dimension dim)
public String getCurrentDir()
public void setCurrentDir(String newDir)
public void updateStatus(int dotPos, int markPos)
public Class[] getDataBaseClasses()
public MetadataFeeder getMetadataFeeder()
public void addProject(Project project)
public boolean setCurrentProject(Project project)
public boolean removeProject(Project project)
public MetaProjectHeader getProgramMetadata()
public void resetDashboard()
public Project loadProject(String fileName, String projectName)
public void setCanSaveMetadata(boolean canSave)
public MetaObject getSelectedObject()
public void deselectObjects()
public void setProject(Project project)
public void editorAction(String actionName, ActionEvent event)
public void setMode(int mode)
public FileManager getFileManager()
public void setFileManager(FileManager fileManager)
public ConfigManager getConfigManager()
public void setConfigManager(ConfigManager configManager)

```
    public ClassLoader getClassLoader()
    public void setClassLoader(ClassLoader classLoader)
    public Properties getProps()
    public String getUserHome()
    public String getBaseDir()
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
    public MetaObject pasting(
      MetaObject target, MetaObject pasted, MetaProject project)
    public void processMenuItems(MetaObject metaObject)
    public void processMenuSeparators(MetaObject metaObject)
    public void processTabPages(MetaObject metaObject)
    public void processPlacement(MetaObject object)
    public void processCreateLayout(MetaObject object)
    public void updateDisplayLayer(MetaObject object, int layerIndex)
    public void propertyEditedRepaint(MetaObject object)
    public void processDeleteObject(MetaObject object)
    public boolean getAttachedToDesigner()
    public void processProjectChangedState(boolean hasProjectChanged)
    public void processObjectNameChanged(MetaObject object)
    public void runProject()
    public void setAçowDragging(boolean allowDragging)
    public boolean allowDragging()
    public boolean isCustomizing()
    public void setTitle(String title)
    public IdeMenuBar getIdeMenuBar()
            public void showHelper(MetaObject metaObject, String
propertyName)
    // … many non-public methods follow …
  }
```

But what if `SuperDashboard` contained only the methods shown in Listing 10-2?

Listing 10-2 `Small Enough?`

```
        public class SuperDashboard extends JFrame implements MetaDataUser
    public Component getLastFocusedComponent()
    public void setLastFocused(Component lastFocused)
    public int getMajorVersionNumber()
    public int getMinorVersionNumber()
    public int getBuildNumber()
}
```

Five methods isn't too much, is it? In this case it is because despite its small number of methods, `SuperDashboard` has too many *responsibilities*.

The name of a class should describe what responsibilities it fulfills. In fact, naming is probably the first way of helping determine class size. If we cannot derive a concise name for a class, then it's likely too large. The more ambiguous the class name, the more likely it has too many responsibilities. For example, class names including weasel words like `Processor` or `Manager` or `Super` often hint at unfortunate aggregation of responsibilities.

We should also be able to write a brief description of the class in about 25 words, without using the words "if," "and," "or," or "but." How would we describe the `SuperDashboard`? "The `SuperDashboard` provides access to the component that last held the focus, and it also allows us to track the version and build numbers." The first "and" is a hint that `SuperDashboard` has too many responsibilities.

The Single Responsibility Principle

The Single Responsibility Principle (SRP)[2] states that a class or module should have one, and only one, *reason to change*. This principle gives us both a definition of responsibility, and a guidelines for class size. Classes should have one responsibility—one reason to change.

The seemingly small `SuperDashboard` class in Listing 10-2 has two reasons to change. First, it tracks version information that would seemingly need to be updated every time the software gets shipped. Second, it manages Java Swing components (it is a derivative of `JFrame`, the Swing representation of a top-level GUI window). No doubt we'll want to update the version number if we change any of the Swing code, but the

converse isn't necessarily true: We might change the version information based on changes to other code in the system.

Trying to identify responsibilities (reasons to change) often helps us recognize and create better abstractions in our code. We can easily extract all three `SuperDashboard` methods that deal with version information into a separate class named `Version`. (See Listing 10-3.) The `Version` class is a construct that has a high potential for reuse in other applications!

Listing 10-3 `A single-responsibility class`

```
public class Version {
public int getMajorVersionNumber()
public int getMinorVersionNumber()
public int getBuildNumber()
}
```

SRP is one of the more important concept in OO design. It's also one of the simpler concepts to understand and adhere to. Yet oddly, SRP is often the most abused class design principle. We regularly encounter classes that do far too many things. Why?

Getting software to work and making software clean are two very different activities. Most of us have limited room in our heads, so we focus on getting our code to work more than organization and cleanliness. This is wholly appropriate. Maintaining a separation of concerns is just as important in our programming *activities* as it is in our programs.

The problem is that too many of us think that we are done once the program works. We fail to switch to the *other* concern of organization and cleanliness. We move on to the next problem rather than going back and breaking the overstuffed classes into decoupled units with single responsibilities.

At the same time, many developers fear that a large number of small, single-purpose classes makes it more difficult to understand the bigger picture. They are concerned that they must navigate from class to class in order to figure out how a larger piece of work gets accomplished.

However, a system with many small classes has no more moving parts than a system with a few large classes. There is just as much to learn in the system with a few large classes. So the question is: Do you want your tools

organized into toolboxes with many small drawers each containing well-defined and well-labeled components? Or do you want a few drawers that you just toss everything into?

Every sizable system will contain a large amount of logic and complexity. The primary goal in managing such complexity is to *organize* it so that a developer knows where to look to find things and need only understand the directly affected complexity at any given time. In contrast, a system with larger, multipurpose classes always hampers us by insisting we wade through lots of things we don't need to know right now.

To restate the former points for emphasis: We want our systems to be composed of many small classes, not a few large ones. Each small class encapsulates a single responsibility, has a single reason to change, and collaborates with a few others to achieve the desired system behaviors.

Cohesion

Classes should have a small number of instance variables. Each of the methods of a class should manipulate one or more of those variables. In general the more variables a method manipulates the more cohesive that method is to its class. A class in which each variable is used by each method is maximally cohesive.

In general it is neither advisable nor possible to create such maximally cohesive classes; on the other hand, we would like cohesion to be high. When cohesion is high, it means that the methods and variables of the class are co-dependent and hang together as a logical whole.

Consider the implementation of a `Stack` in Listing 10-4. This is a very cohesive class. Of the three methods only `size()` fails to use both the variables.

Listing 10-4 `Stack.java A cohesive class.`

```
 public class Stack {
private int topOfStack = 0;
List<Integer> elements = new LinkedList<Integer>();

public int size() {
 return topOfStack;
```

```
  }

  public void push(int element) {
    topOfStack++;
    elements.add(element);
  }

  public int pop() throws PoppedWhenEmpty {
    if (topOfStack == 0)
      throw new PoppedWhenEmpty();
    int element = elements.get(--topOfStack);
    elements.remove(topOfStack);
    return element;
  }
}
```

The strategy of keeping functions small and keeping parameter lists short can sometimes lead to a proliferation of instance variables that are used by a subset of methods. When this happens, it almost always means that there is at least one other class trying to get out of the larger class. You should try to separate the variables and methods into two or more classes such that the new classes are more cohesive.

Maintaining Cohesion Results in Many Small Classes

Just the act of breaking large functions into smaller functions causes a proliferation of classes. Consider a large function with many variables declared within it. Let's say you want to extract one small part of that function into a separate function. However, the code you want to extract uses four of the variables declared in the function. Must you pass all four of those variables into the new function as arguments?

Not at all! If we promoted those four variables to instance variables of the class, then we could extract the code without passing *any* variables at all. It would be *easy* to break the function up into small pieces.

Unfortunately, this also means that our classes lose cohesion because they accumulate more and more instance variables that exist solely to allow a few functions to share them. But wait! If there are a few functions

that want to share certain variables, doesn't that make them a class in their own right? Of course it does. When classes lose cohesion, split them!

So breaking a large function into many smaller functions often gives us the opportunity to split several smaller classes out as well. This gives our program a much better organization and a more transparent structure.

As a demonstration of what I mean, let's use a time-honored example taken from Knuth's wonderful book *Literate Programming*.[3] Listing 10-5 shows a translation into Java of Knuth's `PrintPrimes` program. To be fair to Knuth, this is not the program as he wrote it but rather as it was output by his WEB tool. I'm using it because it makes a great starting place for breaking up a big function into many smaller functions and classes.

Listing 10-5 `PrintPrimes.java`

```
package literatePrimes;

public class PrintPrimes {
  public static void main(String[] args) {
    final int M = 1000;
    final int RR = 50;
    final int CC = 4;
    final int WW = 10;
    final int ORDMAX = 30;
    int P[] = new int[M + 1];
    int PAGENUMBER;
    int PAGEOFFSET;
    int ROWOFFSET;
    int C;

    int J;
    int K;
    boolean JPRIME;
    int ORD;
    int SQUARE;
    int N;
    int MULT[] = new int[ORDMAX + 1];
```

```
      J = 1;
      K = 1;
      P[1] = 2;
      ORD = 2;
      SQUARE = 9;

      while (K < M) {
        do {
          J = J + 2;
          if (J == SQUARE) {
            ORD = ORD + 1;
            SQUARE = P[ORD] * P[ORD];
            MULT[ORD - 1] = J;
          }
          N = 2;
          JPRIME = true;
          while (N < ORD && JPRIME) {
            while (MULT[N] < J)
              MULT[N] = MULT[N] + P[N] + P[N];
            if (MULT[N] == J)
              JPRIME = false;
            N = N + 1;
          }
        } while (!JPRIME);
        K = K + 1;
        P[K] = J;
      }
      {
        PAGENUMBER = 1;
        PAGEOFFSET = 1;
        while (PAGEOFFSET <= M) {
          System.out.println("The First " + M +
                             " Prime Numbers --- Page " + PAGENUMBER);
          System.out.println("");
                    for (ROWOFFSET = PAGEOFFSET; ROWOFFSET <
PAGEOFFSET + RR; ROWOFFSET++){
            for (C = 0; C < CC;C++)
```

```
        if (ROWOFFSET + C * RR <= M)
          System.out.format("%10d", P[ROWOFFSET + C * RR]);
        System.out.println("");
      }
      System.out.println("\f");
      PAGENUMBER = PAGENUMBER + 1;
      PAGEOFFSET = PAGEOFFSET + RR * CC;
    }
   }
  }
}
```

This program, written as a single function, is a mess. It has a deeply indented structure, a plethora of odd variables, and a tightly coupled structure. At the very least, the one big function should be split up into a few smaller functions.

Listing 10-6 through Listing 10-8 show the result of splitting the code in Listing 10-5 into smaller classes and functions, and choosing meaningful names for those classes, functions, and variables.

Listing 10-6 `PrimePrinter.java (refactored)`

```
package literatePrimes;

public class PrimePrinter {
  public static void main(String[] args) {
  final int NUMBER_OF_PRIMES = 1000;
  int[] primes = PrimeGenerator.generate(NUMBER_OF_PRIMES);

  final int ROWS_PER_PAGE = 50;
  final int COLUMNS_PER_PAGE = 4;
  RowColumnPagePrinter tablePrinter =
    new RowColumnPagePrinter(ROWS_PER_PAGE,
                             COLUMNS_PER_PAGE,
                             "The First " + NUMBER_OF_PRIMES +
                                  " Prime Numbers");
    tablePrinter.print(primes);
  }
```

```
}
```

Listing 10-7 `RowColumnPagePrinter.java`

```
   package literatePrimes;

import java.io.PrintStream;

public class RowColumnPagePrinter {
  private int rowsPerPage;
  private int columnsPerPage;
  private int numbersPerPage;
  private String pageHeader;
  private PrintStream printStream;

  public RowColumnPagePrinter(int rowsPerPage,
                              int columnsPerPage,
                              String pageHeader) {
    this.rowsPerPage = rowsPerPage;
    this.columnsPerPage = columnsPerPage;
    this.pageHeader = pageHeader;
    numbersPerPage = rowsPerPage * columnsPerPage;
    printStream = System.out;
}

public void print(int data[]) {
  int pageNumber = 1;
  for (int firstIndexOnPage = 0;
       firstIndexOnPage < data.length;
       firstIndexOnPage += numbersPerPage) {
    int lastIndexOnPage =
      Math.min(firstIndexOnPage + numbersPerPage - 1,
               data.length - 1);
    printPageHeader(pageHeader, pageNumber);
    printPage(firstIndexOnPage, lastIndexOnPage, data);
    printStream.println("\f");
    pageNumber++;
```

```
  }
}

private void printPage(int firstIndexOnPage,
                       int lastIndexOnPage,
                       int[] data) {
 int firstIndexOfLastRowOnPage =
   firstIndexOnPage + rowsPerPage - 1;
 for (int firstIndexInRow = firstIndexOnPage;
      firstIndexInRow <= firstIndexOfLastRowOnPage;
      firstIndexInRow++) {
   printRow(firstIndexInRow, lastIndexOnPage, data);
   printStream.println("");
  }
 }

 private void printRow(int firstIndexInRow,
                       int lastIndexOnPage,
                       int[] data) {
  for (int column = 0; column < columnsPerPage; column++) {
    int index = firstIndexInRow + column * rowsPerPage;
    if (index <= lastIndexOnPage)
     printStream.format("%10d", data[index]);
  }
 }

 private void printPageHeader(String pageHeader,
                              int pageNumber) {
  printStream.println(pageHeader + " --- Page " + pageNumber);
  printStream.println("");
 }

 public void setOutput(PrintStream printStream) {
  this.printStream = printStream;
 }
}
```

Listing 10-8 `PrimeGenerator.java`

```
  package literatePrimes;

import java.util.ArrayList;

public class PrimeGenerator {
 private static int[] primes;
 private static ArrayList<Integer> multiplesOfPrimeFactors;

 protected static int[] generate(int n) {
  primes = new int[n];
  multiplesOfPrimeFactors = new ArrayList<Integer>();
  set2AsFirstPrime();
  checkOddNumbersForSubsequentPrimes();
  return primes;
 }

 private static void set2AsFirstPrime() {
  primes[0] = 2;
  multiplesOfPrimeFactors.add(2);
 }

 private static void checkOddNumbersForSubsequentPrimes() {
  int primeIndex = 1;
  for (int candidate = 3;
       primeIndex < primes.length;
       candidate += 2) {
    if (isPrime(candidate))
      primes[primeIndex++] = candidate;
  }
 }

 private static boolean isPrime(int candidate) {
  if (isLeastRelevantMultipleOfNextLargerPrimeFactor(candidate)) {
    multiplesOfPrimeFactors.add(candidate);
    return false;
  }
```

```
    return isNotMultipleOfAnyPreviousPrimeFactor(candidate);
  }

  private static boolean
  isLeastRelevantMultipleOfNextLargerPrimeFactor(int candidate) {
    int nextLargerPrimeFactor = primes[multiplesOfPrimeFactors.size()];
                int leastRelevantMultiple = nextLargerPrimeFactor *
nextLargerPrimeFactor;
    return candidate == leastRelevantMultiple;
  }

  private static boolean
  isNotMultipleOfAnyPreviousPrimeFactor(int candidate) {
    for (int n = 1; n < multiplesOfPrimeFactors.size(); n++) {
      if (isMultipleOfNthPrimeFactor(candidate, n))
        return false;
    }
    return true;
  }

  private static boolean
  isMultipleOfNthPrimeFactor(int candidate, int n) {
   return
                                                    candidate        ==
smallestOddNthMultipleNotLessThanCandidate(candidate, n);
  }

  private static int
  smallestOddNthMultipleNotLessThanCandidate(int candidate, int n) {
    int multiple = multiplesOfPrimeFactors.get(n);
    while (multiple < candidate)
      multiple += 2 * primes[n];
    multiplesOfPrimeFactors.set(n, multiple);
    return multiple;
  }
 }
```

The first thing you might notice is that the program got a lot longer. It went from a little over one page to nearly three pages in length. There are several reasons for this growth. First, the refactored program uses longer, more descriptive variable names. Second, the refactored program uses function and class declarations as a way to add commentary to the code. Third, we used whitespace and formatting techniques to keep the program readable.

Notice how the program has been split into three main responsibilities. The main program is contained in the `PrimePrinter` class all by itself. Its responsibility is to handle the execution environment. It will change if the method of invocation changes. For example, if this program were converted to a SOAP service, this is the class that would be affected.

The `RowColumnPagePrinter` knows all about how to format a list of numbers into pages with a certain number of rows and columns. If the formatting of the output needed changing, then this is the class that would be affected.

The `PrimeGenerator` class knows how to generate a list prime numbers. Notice that it is not meant to be instantiated as an object. The class is just a useful scope in which its variables can be declared and kept hidden. This class will change if the algorithm for computing prime numbers changes.

This was not a rewrite! We did not start over from scratch and write the program over again. Indeed, if you look closely at the two different programs, you'll see that they use the same algorithm and mechanics to get their work done.

The change was made by writing a test suite that verified the *precise* behavior of the first program. Then a myriad of tiny little changes were made, one at a time. After each change the program was executed to ensure that the behavior had not changed. One tiny step after another, the first program was cleaned up and transformed into the second.

Organizing for Change

For most systems, change is continual. Every change subjects us to the risk that the remainder of the system no longer works as intended. In a clean system we organize our classes so as to reduce the risk of change.

The `Sql` class in Listing 10-9 is used to generate properly formed SQL strings given appropriate metadata. It's a work in progress and, as such, doesn't yet support SQL functionality like `update` statements. When the time comes for the `Sql` class to support an `update` statement, we'll have to "open up" this class to make modifications. The problem with opening a class is that it introduces risk. Any modifications to the class have the potential of breaking other code in the class. It must be fully retested.

Listing 10-9 `A class that must be opened for change`

```
public class Sql {   public Sql(String table, Column[] columns)
  public String create()
  public String insert(Object[] fields)
  public String selectAll()
  public String findByKey(String keyColumn, String keyValue)
  public String select(Column column, String pattern)
  public String select(Criteria criteria)
  public String preparedInsert()
  private String columnList(Column[] columns)
  private String valuesList(Object[] fields, final Column[] columns)
  private String selectWithCriteria(String criteria)
  private String placeholderList(Column[] columns)
}
```

The `Sql` class must change when we add a new type of statement. It also must change when we alter the details of a single statement type—for example, if we need to modify the `select` functionality to support subselects. These two reasons to change mean that the `Sql` class violates the SRP.

We can spot this SRP violation from a simple organizational standpoint. The method outline of `Sql` shows that there are private methods, such as `selectWithCriteria`, that appear to relate only to `select` statements.

Private method behavior that applies only to a small subset of a class can be a useful heuristic for spotting potential areas for improvement. However, the primary spur for taking action should be system change itself. If the `Sql` class is deemed logically complete, then we need not worry about separating the responsibilities. If we won't need `update` functionality for the foreseeable future, then we should leave `Sql` alone.

But as soon as we find ourselves opening up a class, we should consider fixing our design.

What if we considered a solution like that in Listing 10-10? Each public interface method defined in the previous `Sql` from Listing 10-9 is refactored out to its own derivative of the `Sql` class. Note that the private methods, such as `valuesList`, move directly where they are needed. The common private behavior is isolated to a pair of utility classes, `Where` and `ColumnList`.

Listing 10-10 `A set of closed classes`

```
 abstract public class Sql {
  public Sql(String table, Column[] columns)
  abstract public String generate();
}

public class CreateSql extends Sql {
  public CreateSql(String table, Column[] columns)
  @Override public String generate()
}

public class SelectSql extends Sql {
  public SelectSql(String table, Column[] columns)
  @Override public String generate()
}

public class InsertSql extends Sql {
  public InsertSql(String table, Column[] columns, Object[] fields)
  @Override public String generate()
  private String valuesList(Object[] fields, final Column[] columns)
}

public class SelectWithCriteriaSql extends Sql {
  public SelectWithCriteriaSql(
  String table, Column[] columns, Criteria criteria)
  @Override public String generate()
}
```

```
public class SelectWithMatchSql extends Sql {
  public SelectWithMatchSql(
    String table, Column[] columns, Column column, String pattern)
  @Override public String generate()
}

public class FindByKeySql extends Sql
  public FindByKeySql(
    String table, Column[] columns, String keyColumn, String keyValue)
  @Override public String generate()
}

public class PreparedInsertSql extends Sql {
  public PreparedInsertSql(String table, Column[] columns)
    @Override public String generate() {
  private String placeholderList(Column[] columns)
}

public class Where {
  public Where(String criteria)
  public String generate()
}

public class ColumnList {
  public ColumnList(Column[] columns)
  public String generate()
}
```

The code in each class becomes excruciatingly simple. Our required comprehension time to understand any class decreases to almost nothing. The risk that one function could break another becomes vanishingly small. From a test standpoint, it becomes an easier task to prove all bits of logic in this solution, as the classes are all isolated from one another.

Equally important, when it's time to add the `update` statements, none of the existing classes need change! We code the logic to build `update`

statements in a new subclass of `Sql` named `UpdateSql`. No other code in the system will break because of this change.

Our restructured `Sql` logic represents the best of all worlds. It supports the SRP. It also supports another key OO class design principle known as the Open-Closed Principle, or OCP:[4] Classes should be open for extension but closed for modification. Our restructured `Sql` class is open to allow new functionality via subclassing, but we can make this change while keeping every other class closed. We simply drop our `UpdateSql` class in place.

We want to structure our systems so that we muck with as little as possible when we update them with new or changed features. In an ideal system, we incorporate new features by extending the system, not by making modifications to existing code.

Isolating from Change

Needs will change, therefore code will change. We learned in OO 101 that there are concrete classes, which contain implementation details (code), and abstract classes, which represent concepts only. A client class depending upon concrete details is at risk when those details change. We can introduce interfaces and abstract classes to help isolate the impact of those details.

Dependencies upon concrete details create challenges for testing our system. If we're building a `Portfolio` class and it depends upon an external `TokyoStockExchange` API to derive the portfolio's value, our test cases are impacted by the volatility of such a lookup. It's hard to write a test when we get a different answer every five minutes!

Instead of designing `Portfolio` so that it directly depends upon `TokyoStockExchange`, we create an interface, `StockExchange`, that declares a single method:

```
public interface StockExchange {
  Money currentPrice(String symbol);
}
```

We design `TokyoStockExchange` to implement this interface. We also make sure that the constructor of `Portfolio` takes a `StockExchange` reference as an argument:

```
 public Portfolio {
private StockExchange exchange;
public Portfolio(StockExchange exchange) {
  this.exchange = exchange;
}
// …
}
```

Now our test can create a testable implementation of the `StockExchange` interface that emulates the `TokyoStockExchange`. This test implementation will fix the current value for any symbol we use in testing. If our test demonstrates purchasing five shares of Microsoft for our portfolio, we code the test implementation to always return $100 per share of Microsoft. Our test implementation of the `StockExchange` interface reduces to a simple table lookup. We can then write a test that expects $500 for our overall portfolio value.

```
 public class PortfolioTest {
private FixedStockExchangeStub exchange;
private Portfolio portfolio;

@Before
protected void setUp() throws Exception {
  exchange = new FixedStockExchangeStub();
  exchange.fix(”MSFT”, 100);
  portfolio = new Portfolio(exchange);
}

@Test
public void GivenFiveMSFTTotalShouldBe500() throws Exception {
  portfolio.add(5, ”MSFT”);
  Assert.assertEquals(500, portfolio.value());
}
}
```

If a system is decoupled enough to be tested in this way, it will also be more flexible and promote more reuse. The lack of coupling means that the elements of our system are better isolated from each other and from

change. This isolation makes it easier to understand each element of the system.

By minimizing coupling in this way, our classes adhere to another class design principle known as the Dependency Inversion Principle (DIP).[5] In essence, the DIP says that our classes should depend upon abstractions, not on concrete details.

Instead of being dependent upon the implementation details of the `TokyoStock-Exchange` class, our `Portfolio` class is now dependent upon the `StockExchange` interface. The `StockExchange` interface represents the abstract concept of asking for the current price of a symbol. This abstraction isolates all of the specific details of obtaining such a price, including from where that price is obtained.

Bibliography

[RDD]: *Object Design: Roles, Responsibilities, and Collaborations*, Rebecca Wirfs-Brock et al., Addison-Wesley, 2002.

[PPP]: *Agile Software Development: Principles, Patterns, and Practices*, Robert C. Martin, Prentice Hall, 2002.

[Knuth92]: *Literate Programming*, Donald E. Knuth, Center for the Study of language and Information, Leland Stanford Junior University, 1992.