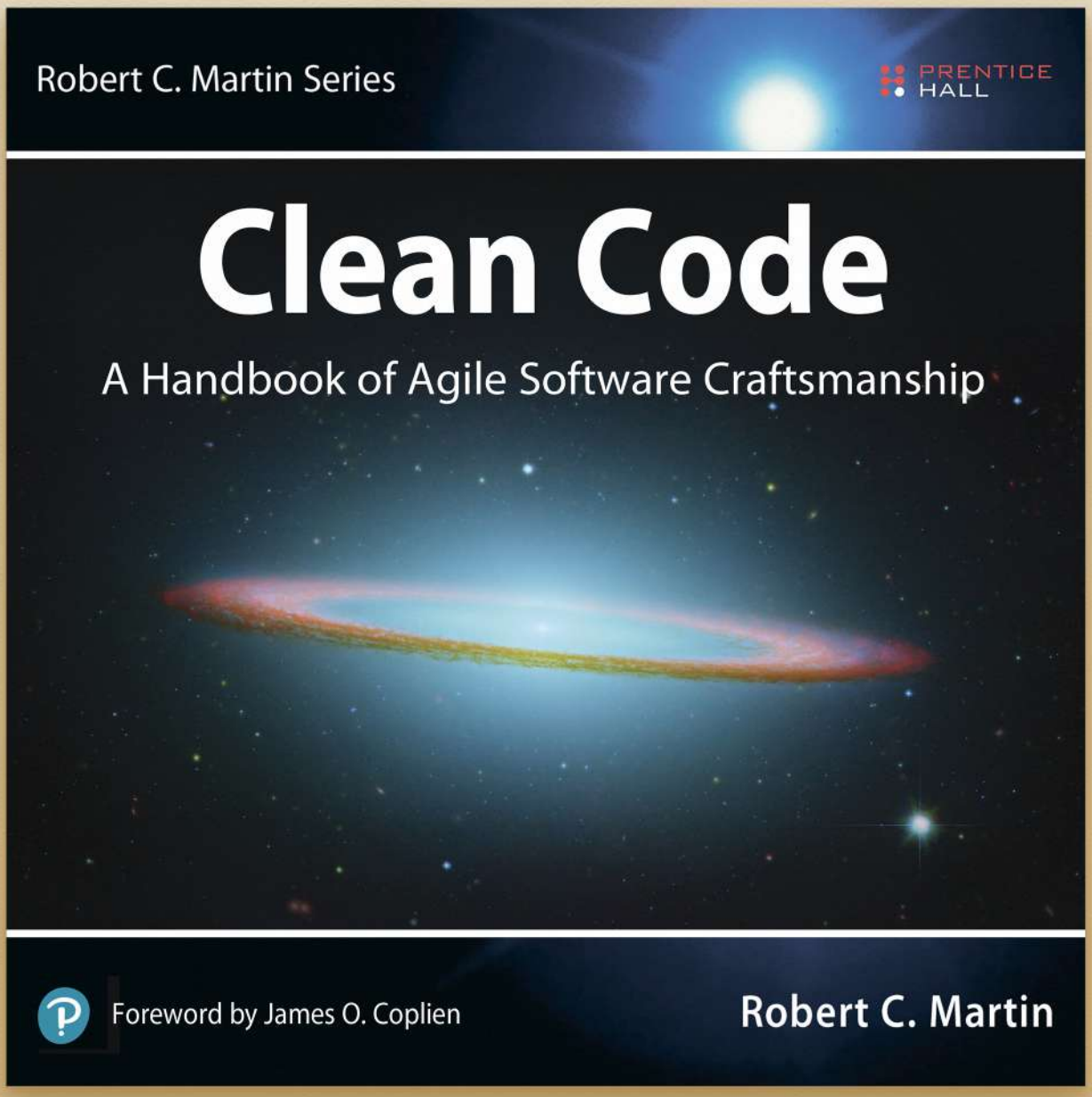


TABLE OF CONTENTS

03
CHAPTER 14

61
CHAPTER 15

76
CHAPTER 16

94
CHAPTER 17

125
APPENDIX A

156
APPENDIX B

215
APPENDIX C

14

Successive Refinement

Case Study of a Command-Line Argument Parser

This chapter is a case study in successive refinement. You will see a module that started well but did not scale. Then you will see how the module was refactored and cleaned.

Most of us have had to parse command-line arguments from time to time. If we don't have a convenient utility, then we simply walk the array of strings that is passed into the `main` function. There are several good utilities available from various sources,

but none of them do exactly what I want. So, of course, I decided to write my own. I call it: `Args`.

`Args` is very simple to use. You simply construct the `Args` class with the input arguments and a format string, and then query the `Args` instance for the values of the arguments. Consider the following simple example:

Listing 14-1
`Simple use of Args`

```
public static void main(String[] args) {
  try {
    Args arg = new Args("l,p#,d*", args);
    boolean logging = arg.getBoolean('l');
    int port = arg.getInt('p');
    String directory = arg.getString('d');
    executeApplication(logging, port, directory);
  } catch (ArgsException e) {
    System.out.printf("Argument error: %s\n", e.errorMessage());
  }
}
```

You can see how simple this is. We just create an instance of the `Args` class with two parameters. The first parameter is the format, or *schema,* string: `"l,p#,d*."` It defines three command-line arguments. The first, `-l`, is a boolean argument. The second, `-p`, is an integer argument. The third, `-d`, is a string argument. The second parameter to the `Args` constructor is simply the array of command-line argument passed into `main`.

If the constructor returns without throwing an `ArgsException`, then the incoming command-line was parsed, and the `Args` instance is ready to be queried. Methods like `getBoolean`, `getInteger`, and `getString` allow us to access the values of the arguments by their names.

If there is a problem, either in the format string or in the command-line arguments themselves, an `ArgsException` will be thrown. A convenient description of what went wrong can be retrieved from the `errorMessage` method of the exception.

Args Implementation

Listing 14-2 is the implementation of the `Args` class. Please read it very carefully. I worked hard on the style and structure and hope it is worth emulating.

Listing 14-2
`Args.java`

```
package com.objectmentor.utilities.args;

import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;
import java.util.*;

public class Args {
  private Map<Character, ArgumentMarshaler> marshalers;
```

Listing 14-2 (continued)
`Args.java`

```
  private Set<Character> argsFound;
  private ListIterator<String> currentArgument;

  public Args(String schema, String[] args) throws ArgsException {
    marshalers = new HashMap<Character, ArgumentMarshaler>();
    argsFound = new HashSet<Character>();

    parseSchema(schema);
    parseArgumentStrings(Arrays.asList(args));
  }

  private void parseSchema(String schema) throws ArgsException {
    for (String element : schema.split(","))
      if (element.length() > 0)
        parseSchemaElement(element.trim());
  }

  private void parseSchemaElement(String element) throws ArgsException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (elementTail.length() == 0)
      marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (elementTail.equals("*"))
      marshalers.put(elementId, new StringArgumentMarshaler());
    else if (elementTail.equals("#"))
      marshalers.put(elementId, new IntegerArgumentMarshaler());
    else if (elementTail.equals("##"))
      marshalers.put(elementId, new DoubleArgumentMarshaler());
    else if (elementTail.equals("[*]"))
      marshalers.put(elementId, new StringArrayArgumentMarshaler());
    else
      throw new ArgsException(INVALID_ARGUMENT_FORMAT, elementId, elementTail);
  }

  private void validateSchemaElementId(char elementId) throws ArgsException {
    if (!Character.isLetter(elementId))
      throw new ArgsException(INVALID_ARGUMENT_NAME, elementId, null);
  }

  private void parseArgumentStrings(List<String> argsList) throws ArgsException
  {
    for (currentArgument = argsList.listIterator(); currentArgument.hasNext();)
    {
      String argString = currentArgument.next();
      if (argString.startsWith("-")) {
        parseArgumentCharacters(argString.substring(1));
      } else {
        currentArgument.previous();
        break;
      }
    }
  }
```

Listing 14-2 (continued)
Args.java

```
  private void parseArgumentCharacters(String argChars) throws ArgsException {
    for (int i = 0; i < argChars.length(); i++)
      parseArgumentCharacter(argChars.charAt(i));
  }

  private void parseArgumentCharacter(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null) {
      throw new ArgsException(UNEXPECTED_ARGUMENT, argChar, null);
    } else {
      argsFound.add(argChar);
      try {
        m.set(currentArgument);
      } catch (ArgsException e) {
        e.setErrorArgumentId(argChar);
        throw e;
      }
    }
  }

  public boolean has(char arg) {
    return argsFound.contains(arg);
  }

  public int nextArgument() {
    return currentArgument.nextIndex();
  }

  public boolean getBoolean(char arg) {
    return BooleanArgumentMarshaler.getValue(marshalers.get(arg));
  }

  public String getString(char arg) {
    return StringArgumentMarshaler.getValue(marshalers.get(arg));
  }

  public int getInt(char arg) {
    return IntegerArgumentMarshaler.getValue(marshalers.get(arg));
  }

  public double getDouble(char arg) {
    return DoubleArgumentMarshaler.getValue(marshalers.get(arg));
  }

  public String[] getStringArray(char arg) {
    return StringArrayArgumentMarshaler.getValue(marshalers.get(arg));
  }
}
```

Notice that you can read this code from the top to the bottom without a lot of jumping around or looking ahead. The one thing you may have had to look ahead for is the definition of `ArgumentMarshaler`, which I left out intentionally. Having read this code carefully,

you should understand what the `ArgumentMarshaler` interface is and what its derivatives do. I'll show a few of them to you now (Listing 14-3 through Listing 14-6).

Listing 14-3
`ArgumentMarshaler.java`

```
public interface ArgumentMarshaler {
  void set(Iterator<String> currentArgument) throws ArgsException;
}
```

Listing 14-4
`BooleanArgumentMarshaler.java`

```
public class BooleanArgumentMarshaler implements ArgumentMarshaler {
  private boolean booleanValue = false;

  public void set(Iterator<String> currentArgument) throws ArgsException {
    booleanValue = true;
  }

  public static boolean getValue(ArgumentMarshaler am) {
    if (am != null && am instanceof BooleanArgumentMarshaler)
      return ((BooleanArgumentMarshaler) am).booleanValue;
    else
      return false;
  }
}
```

Listing 14-5
`StringArgumentMarshaler.java`

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class StringArgumentMarshaler implements ArgumentMarshaler {
  private String stringValue = "";

  public void set(Iterator<String> currentArgument) throws ArgsException {
    try {
      stringValue = currentArgument.next();
    } catch (NoSuchElementException e) {
      throw new ArgsException(MISSING_STRING);
    }
  }

  public static String getValue(ArgumentMarshaler am) {
    if (am != null && am instanceof StringArgumentMarshaler)
      return ((StringArgumentMarshaler) am).stringValue;
    else
      return "";
  }
}
```

Listing 14-6
IntegerArgumentMarshaler.java

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class IntegerArgumentMarshaler implements ArgumentMarshaler {
  private int intValue = 0;

  public void set(Iterator<String> currentArgument) throws ArgsException {
    String parameter = null;
    try {
      parameter = currentArgument.next();
      intValue = Integer.parseInt(parameter);
    } catch (NoSuchElementException e) {
      throw new ArgsException(MISSING_INTEGER);
    } catch (NumberFormatException e) {
      throw new ArgsException(INVALID_INTEGER, parameter);
    }
  }

  public static int getValue(ArgumentMarshaler am) {
    if (am != null && am instanceof IntegerArgumentMarshaler)
      return ((IntegerArgumentMarshaler) am).intValue;
    else
      return 0;
  }
}
```

The other `ArgumentMarshaler` derivatives simply replicate this pattern for `doubles` and `String` arrays and would serve to clutter this chapter. I'll leave them to you as an exercise.

One other bit of information might be troubling you: the definition of the error code constants. They are in the `ArgsException` class (Listing 14-7).

Listing 14-7
ArgsException.java

```
import static com.objectmentor.utilities.args.ArgsException.ErrorCode.*;

public class ArgsException extends Exception {
  private char errorArgumentId = '\0';
  private String errorParameter = null;
  private ErrorCode errorCode = OK;

  public ArgsException() {}

  public ArgsException(String message) {super(message);}

  public ArgsException(ErrorCode errorCode) {
    this.errorCode = errorCode;
  }

  public ArgsException(ErrorCode errorCode, String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
  }
```

Listing 14-7 (continued)
`ArgsException.java`

```
  public ArgsException(ErrorCode errorCode,
                       char errorArgumentId, String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
    this.errorArgumentId = errorArgumentId;
  }

  public char getErrorArgumentId() {
    return errorArgumentId;
  }

  public void setErrorArgumentId(char errorArgumentId) {
    this.errorArgumentId = errorArgumentId;
  }

  public String getErrorParameter() {
    return errorParameter;
  }

  public void setErrorParameter(String errorParameter) {
    this.errorParameter = errorParameter;
  }

  public ErrorCode getErrorCode() {
    return errorCode;
  }

  public void setErrorCode(ErrorCode errorCode) {
    this.errorCode = errorCode;
  }

  public String errorMessage() {
    switch (errorCode) {
      case OK:
        return "TILT: Should not get here.";
      case UNEXPECTED_ARGUMENT:
        return String.format("Argument -%c unexpected.", errorArgumentId);
      case MISSING_STRING:
        return String.format("Could not find string parameter for -%c.",
                             errorArgumentId);
      case INVALID_INTEGER:
        return String.format("Argument -%c expects an integer but was '%s'.",
                             errorArgumentId, errorParameter);
      case MISSING_INTEGER:
        return String.format("Could not find integer parameter for -%c.",
                             errorArgumentId);
      case INVALID_DOUBLE:
        return String.format("Argument -%c expects a double but was '%s'.",
                             errorArgumentId, errorParameter);
      case MISSING_DOUBLE:
        return String.format("Could not find double parameter for -%c.",
                             errorArgumentId);
      case INVALID_ARGUMENT_NAME:
        return String.format("'%c' is not a valid argument name.",
                             errorArgumentId);
```

Listing 14-7 (continued)
ArgsException.java

```
      case INVALID_ARGUMENT_FORMAT:
        return String.format("'%s' is not a valid argument format.",
                             errorParameter);
    }
    return "";
  }

  public enum ErrorCode {
    OK, INVALID_ARGUMENT_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
    MISSING_STRING,
    MISSING_INTEGER, INVALID_INTEGER,
    MISSING_DOUBLE, INVALID_DOUBLE}
}
```

It's remarkable how much code is required to flesh out the details of this simple concept. One of the reasons for this is that we are using a particularly wordy language. Java, being a statically typed language, requires a lot of words in order to satisfy the type system. In a language like Ruby, Python, or Smalltalk, this program is much smaller.[1]

Please read the code over one more time. Pay special attention to the way things are named, the size of the functions, and the formatting of the code. If you are an experienced programmer, you may have some quibbles here and there with various parts of the style or structure. Overall, however, I hope you conclude that this program is nicely written and has a clean structure.

For example, it should be obvious how you would add a new argument type, such as a date argument or a complex number argument, and that such an addition would require a trivial amount of effort. In short, it would simply require a new derivative of `ArgumentMarshaler`, a new `getXXX` function, and a new case statement in the `parseSchemaElement` function. There would also probably be a new `ArgsException.ErrorCode` and a new error message.

How Did I Do This?

Let me set your mind at rest. I did not simply write this program from beginning to end in its current form. More importantly, I am not expecting you to be able to write clean and elegant programs in one pass. If we have learned anything over the last couple of decades, it is that programming is a craft more than it is a science. To write clean code, you must first write dirty code *and then clean it*.

This should not be a surprise to you. We learned this truth in grade school when our teachers tried (usually in vain) to get us to write rough drafts of our compositions. The process, they told us, was that we should write a rough draft, then a second draft, then several subsequent drafts until we had our final version. Writing clean compositions, they tried to tell us, is a matter of successive refinement.

---

1. I recently rewrote this module in Ruby. It was 1/7th the size and had a subtly better structure.

Most freshman programmers (like most grade-schoolers) don't follow this advice particularly well. They believe that the primary goal is to get the program working. Once it's "working," they move on to the next task, leaving the "working" program in whatever state they finally got it to "work." Most seasoned programmers know that this is professional suicide.

Args: The Rough Draft

Listing 14-8 shows an earlier version of the `Args` class. It "works." And it's messy.

Listing 14-8
`Args.java (first draft)`

```
import java.text.ParseException;
import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, Boolean> booleanArgs =
    new HashMap<Character, Boolean>();
  private Map<Character, String> stringArgs = new HashMap<Character, String>();
  private Map<Character, Integer> intArgs = new HashMap<Character, Integer>();
  private Set<Character> argsFound = new HashSet<Character>();
  private int currentArgument;
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;

  private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT}

  public Args(String schema, String[] args) throws ParseException {
    this.schema = schema;
    this.args = args;
    valid = parse();
  }

  private boolean parse() throws ParseException {
    if (schema.length() == 0 && args.length == 0)
      return true;
    parseSchema();
    try {
      parseArguments();
    } catch (ArgsException e) {
    }
    return valid;
  }

  private boolean parseSchema() throws ParseException {
    for (String element : schema.split(",")) {
```

Listing 14-8 (continued)
Args.java (first draft)

```
      if (element.length() > 0) {
        String trimmedElement = element.trim();
        parseSchemaElement(trimmedElement);
      }
    }
    return true;
  }

  private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
      parseBooleanSchemaElement(elementId);
    else if (isStringSchemaElement(elementTail))
      parseStringSchemaElement(elementId);
    else if (isIntegerSchemaElement(elementTail)) {
      parseIntegerSchemaElement(elementId);
    } else {
      throw new ParseException(
        String.format("Argument: %c has invalid format: %s.",
                      elementId, elementTail), 0);
    }
  }

  private void validateSchemaElementId(char elementId) throws ParseException {
    if (!Character.isLetter(elementId)) {
      throw new ParseException(
        "Bad character:" + elementId + "in Args format: " + schema, 0);
    }
  }

  private void parseBooleanSchemaElement(char elementId) {
    booleanArgs.put(elementId, false);
  }

  private void parseIntegerSchemaElement(char elementId) {
    intArgs.put(elementId, 0);
  }

  private void parseStringSchemaElement(char elementId) {
    stringArgs.put(elementId, "");
  }

  private boolean isStringSchemaElement(String elementTail) {
    return elementTail.equals("*");
  }

  private boolean isBooleanSchemaElement(String elementTail) {
    return elementTail.length() == 0;
  }

  private boolean isIntegerSchemaElement(String elementTail) {
    return elementTail.equals("#");
  }
```

Listing 14-8 (continued)
Args.java (first draft)

```
  private boolean parseArguments() throws ArgsException {
    for (currentArgument = 0; currentArgument < args.length; currentArgument++)
    {
      String arg = args[currentArgument];
      parseArgument(arg);
    }
    return true;
  }

  private void parseArgument(String arg) throws ArgsException {
    if (arg.startsWith("-"))
      parseElements(arg);
  }

  private void parseElements(String arg) throws ArgsException {
    for (int i = 1; i < arg.length(); i++)
      parseElement(arg.charAt(i));
  }

  private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
      argsFound.add(argChar);
    else {
      unexpectedArguments.add(argChar);
      errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
      valid = false;
    }
  }

  private boolean setArgument(char argChar) throws ArgsException {
    if (isBooleanArg(argChar))
      setBooleanArg(argChar, true);
    else if (isStringArg(argChar))
      setStringArg(argChar);
    else if (isIntArg(argChar))
      setIntArg(argChar);
    else
      return false;

    return true;
  }

  private boolean isIntArg(char argChar) {return intArgs.containsKey(argChar);}

  private void setIntArg(char argChar) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
      parameter = args[currentArgument];
      intArgs.put(argChar, new Integer(parameter));
    } catch (ArrayIndexOutOfBoundsException e) {
      valid = false;
      errorArgumentId = argChar;
      errorCode = ErrorCode.MISSING_INTEGER;
```

Listing 14-8 (continued)
Args.java (first draft)

```
      throw new ArgsException();
    } catch (NumberFormatException e) {
      valid = false;
      errorArgumentId = argChar;
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw new ArgsException();
    }
  }

  private void setStringArg(char argChar) throws ArgsException {
    currentArgument++;
    try {
      stringArgs.put(argChar, args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
      valid = false;
      errorArgumentId = argChar;
      errorCode = ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }

  private boolean isStringArg(char argChar) {
    return stringArgs.containsKey(argChar);
  }

  private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.put(argChar, value);
  }

  private boolean isBooleanArg(char argChar) {
    return booleanArgs.containsKey(argChar);
  }

  public int cardinality() {
    return argsFound.size();
  }

  public String usage() {
    if (schema.length() > 0)
      return "-[" + schema + "]";
    else
      return "";
  }

  public String errorMessage() throws Exception {
    switch (errorCode) {
      case OK:
        throw new Exception("TILT: Should not get here.");
      case UNEXPECTED_ARGUMENT:
        return unexpectedArgumentMessage();
      case MISSING_STRING:
        return String.format("Could not find string parameter for -%c.",
                             errorArgumentId);
```

Listing 14-8 (continued)
Args.java (first draft)

```
        case INVALID_INTEGER:
          return String.format("Argument -%c expects an integer but was '%s'.",
                               errorArgumentId, errorParameter);
        case MISSING_INTEGER:
          return String.format("Could not find integer parameter for -%c.",
                               errorArgumentId);
      }
      return "";
    }

    private String unexpectedArgumentMessage() {
      StringBuffer message = new StringBuffer("Argument(s) -");
      for (char c : unexpectedArguments) {
        message.append(c);
      }
      message.append(" unexpected.");

      return message.toString();
    }

    private boolean falseIfNull(Boolean b) {
      return b != null && b;
    }

    private int zeroIfNull(Integer i) {
      return i == null ? 0 : i;
    }

    private String blankIfNull(String s) {
      return s == null ? "" : s;
    }

    public String getString(char arg) {
      return blankIfNull(stringArgs.get(arg));
    }

    public int getInt(char arg) {
      return zeroIfNull(intArgs.get(arg));
    }

    public boolean getBoolean(char arg) {
      return falseIfNull(booleanArgs.get(arg));
    }

    public boolean has(char arg) {
      return argsFound.contains(arg);
    }

    public boolean isValid() {
      return valid;
    }

    private class ArgsException extends Exception {
    }
  }
```

I hope your initial reaction to this mass of code is "I'm certainly glad he didn't leave it like that!" If you feel like this, then remember that's how other people are going to feel about code that you leave in rough-draft form.

Actually "rough draft" is probably the kindest thing you can say about this code. It's clearly a work in progress. The sheer number of instance variables is daunting. The odd strings like `"TILT,"` the `HashSets` and `TreeSets`, and the `try-catch-catch` blocks all add up to a festering pile.

I had not wanted to write a festering pile. Indeed, I was trying to keep things reasonably well organized. You can probably tell that from my choice of function and variable names and the fact that there is a crude structure to the program. But, clearly, I had let the problem get away from me.

The mess built gradually. Earlier versions had not been nearly so nasty. For example, Listing 14-9 shows an earlier version in which only `Boolean` arguments were working.

Listing 14-9
Args.java (Boolean only)

```
package com.objectmentor.utilities.getopts;

import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, Boolean> booleanArgs =
    new HashMap<Character, Boolean>();
  private int numberOfArguments = 0;

  public Args(String schema, String[] args) {
    this.schema = schema;
    this.args = args;
    valid = parse();
  }

  public boolean isValid() {
    return valid;
  }

  private boolean parse() {
    if (schema.length() == 0 && args.length == 0)
      return true;
    parseSchema();
    parseArguments();
    return unexpectedArguments.size() == 0;
  }

  private boolean parseSchema() {
    for (String element : schema.split(",")) {
      parseSchemaElement(element);
    }
```

Listing 14-9 (continued)
Args.java (Boolean only)

```
    return true;
  }

  private void parseSchemaElement(String element) {
    if (element.length() == 1) {
      parseBooleanSchemaElement(element);
    }
  }

  private void parseBooleanSchemaElement(String element) {
    char c = element.charAt(0);
    if (Character.isLetter(c)) {
      booleanArgs.put(c, false);
    }
  }

  private boolean parseArguments() {
    for (String arg : args)
      parseArgument(arg);
    return true;
  }

  private void parseArgument(String arg) {
    if (arg.startsWith("-"))
      parseElements(arg);
  }

  private void parseElements(String arg) {
    for (int i = 1; i < arg.length(); i++)
      parseElement(arg.charAt(i));
  }

  private void parseElement(char argChar) {
    if (isBoolean(argChar)) {
      numberOfArguments++;
      setBooleanArg(argChar, true);
    } else
      unexpectedArguments.add(argChar);
  }

  private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.put(argChar, value);
  }

  private boolean isBoolean(char argChar) {
    return booleanArgs.containsKey(argChar);
  }

  public int cardinality() {
    return numberOfArguments;
  }

  public String usage() {
    if (schema.length() > 0)
       return "-["+schema+"]";
```

Listing 14-9 (continued)
Args.java (Boolean only)

```
    else
      return "";
  }

  public String errorMessage() {
    if (unexpectedArguments.size() > 0) {
      return unexpectedArgumentMessage();
    } else
      return "";
  }

  private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(s) -");
    for (char c : unexpectedArguments) {
      message.append(c);
    }
    message.append(" unexpected.");

    return message.toString();
  }

  public boolean getBoolean(char arg) {
    return booleanArgs.get(arg);
  }
}
```

Although you can find plenty to complain about in this code, it's really not that bad. It's compact and simple and easy to understand. However, within this code it is easy to see the seeds of the later festering pile. It's quite clear how this grew into the latter mess.

Notice that the latter mess has only two more argument types than this: `String` and `integer`. The addition of just two more argument types had a massively negative impact on the code. It converted it from something that would have been reasonably maintainable into something that I would expect to become riddled with bugs and warts.

I added the two argument types incrementally. First, I added the `String` argument, which yielded this:

Listing 14-10
Args.java (Boolean and String)

```
package com.objectmentor.utilities.getopts;

import java.text.ParseException;
import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, Boolean> booleanArgs =
    new HashMap<Character, Boolean>();
```

Listing 14-10 (continued)
Args.java (Boolean and String)

```
  private Map<Character, String> stringArgs =
    new HashMap<Character, String>();
  private Set<Character> argsFound = new HashSet<Character>();
  private int currentArgument;
  private char errorArgument = '\0';

  enum ErrorCode {
    OK, MISSING_STRING}

  private ErrorCode errorCode = ErrorCode.OK;

  public Args(String schema, String[] args) throws ParseException {
    this.schema = schema;
    this.args = args;
    valid = parse();
  }

  private boolean parse() throws ParseException {
    if (schema.length() == 0 && args.length == 0)
      return true;
    parseSchema();
    parseArguments();
    return valid;
  }

  private boolean parseSchema() throws ParseException {
    for (String element : schema.split(",")) {
      if (element.length() > 0) {
        String trimmedElement = element.trim();
        parseSchemaElement(trimmedElement);
      }
    }
    return true;
  }

  private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
      parseBooleanSchemaElement(elementId);
    else if (isStringSchemaElement(elementTail))
      parseStringSchemaElement(elementId);
  }

  private void validateSchemaElementId(char elementId) throws ParseException {
    if (!Character.isLetter(elementId)) {
      throw new ParseException(
        "Bad character:" + elementId + "in Args format: " + schema, 0);
    }

  }

  private void parseStringSchemaElement(char elementId) {
    stringArgs.put(elementId, "");
  }
```

Listing 14-10 (continued)
Args.java (Boolean and String)

```
  private boolean isStringSchemaElement(String elementTail) {
    return elementTail.equals("*");
  }

  private boolean isBooleanSchemaElement(String elementTail) {
    return elementTail.length() == 0;
  }

  private void parseBooleanSchemaElement(char elementId) {
    booleanArgs.put(elementId, false);
  }

  private boolean parseArguments() {
    for (currentArgument = 0; currentArgument < args.length; currentArgument++)
    {
      String arg = args[currentArgument];
      parseArgument(arg);
    }
    return true;
  }

  private void parseArgument(String arg) {
    if (arg.startsWith("-"))
      parseElements(arg);
  }

  private void parseElements(String arg) {
    for (int i = 1; i < arg.length(); i++)
      parseElement(arg.charAt(i));
  }

  private void parseElement(char argChar) {
    if (setArgument(argChar))
      argsFound.add(argChar);
    else {
      unexpectedArguments.add(argChar);
      valid = false;
    }
  }

  private boolean setArgument(char argChar) {
    boolean set = true;
    if (isBoolean(argChar))
      setBooleanArg(argChar, true);
    else if (isString(argChar))
      setStringArg(argChar, "");
    else
      set = false;

    return set;
  }

  private void setStringArg(char argChar, String s) {
    currentArgument++;
    try {
```

Listing 14-10 (continued)
Args.java (Boolean and String)

```
      stringArgs.put(argChar, args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
      valid = false;
      errorArgument = argChar;
      errorCode = ErrorCode.MISSING_STRING;
    }
  }

  private boolean isString(char argChar) {
    return stringArgs.containsKey(argChar);
  }

  private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.put(argChar, value);
  }

  private boolean isBoolean(char argChar) {
    return booleanArgs.containsKey(argChar);
  }

  public int cardinality() {
    return argsFound.size();
  }

  public String usage() {
    if (schema.length() > 0)
      return "-[" + schema + "]";
    else
      return "";
  }

  public String errorMessage() throws Exception {
    if (unexpectedArguments.size() > 0) {
      return unexpectedArgumentMessage();
    } else
      switch (errorCode) {
        case MISSING_STRING:
          return String.format("Could not find string parameter for -%c.",
                               errorArgument);
        case OK:
          throw new Exception("TILT: Should not get here.");
      }
    return "";
  }

  private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(s) -");
    for (char c : unexpectedArguments) {
      message.append(c);
    }
    message.append(" unexpected.");

    return message.toString();
  }
```

Listing 14-10 (continued)
Args.java (Boolean and String)

```
  public boolean getBoolean(char arg) {
    return falseIfNull(booleanArgs.get(arg));
  }

  private boolean falseIfNull(Boolean b) {
    return b == null ? false : b;
  }

  public String getString(char arg) {
    return blankIfNull(stringArgs.get(arg));
  }

  private String blankIfNull(String s) {
    return s == null ? "" : s;
  }

  public boolean has(char arg) {
    return argsFound.contains(arg);
  }

  public boolean isValid() {
    return valid;
  }
}
```

You can see that this is starting to get out of hand. It's still not horrible, but the mess is certainly starting to grow. It's a pile, but it's not festering quite yet. It took the addition of the integer argument type to get this pile really fermenting and festering.

So I Stopped

I had at least two more argument types to add, and I could tell that they would make things much worse. If I bulldozed my way forward, I could probably get them to work, but I'd leave behind a mess that was too large to fix. If the structure of this code was ever going to be maintainable, now was the time to fix it.

So I stopped adding features and started refactoring. Having just added the `String` and `integer` arguments, I knew that each argument type required new code in three major places. First, each argument type required some way to parse its schema element in order to select the `HashMap` for that type. Next, each argument type needed to be parsed in the command-line strings and converted to its true type. Finally, each argument type needed a `getXXX` method so that it could be returned to the caller as its true type.

Many different types, all with similar methods—that sounds like a class to me. And so the `ArgumentMarshaler` concept was born.

On Incrementalism

One of the best ways to ruin a program is to make massive changes to its structure in the name of improvement. Some programs never recover from such "improvements." The problem is that it's very hard to get the program working the same way it worked before the "improvement."

To avoid this, I use the discipline of Test-Driven Development (TDD). One of the central doctrines of this approach is to keep the system running at all times. In other words, using TDD, I am not allowed to make a change to the system that breaks that system. Every change I make must keep the system working as it worked before.

To achieve this, I need a suite of automated tests that I can run on a whim and that verifies that the behavior of the system is unchanged. For the `Args` class I had created a suite of unit and acceptance tests while I was building the festering pile. The unit tests were written in `Java` and administered by `JUnit`. The acceptance tests were written as wiki pages in `FitNesse`. I could run these tests any time I wanted, and if they passed, I was confident that the system was working as I specified.

So I proceeded to make a large number of very tiny changes. Each change moved the structure of the system toward the `ArgumentMarshaler` concept. And yet each change kept the system working. The first change I made was to add the skeleton of the `ArgumentMarshaller` to the end of the festering pile (Listing 14-11).

Listing 14-11
`ArgumentMarshaller` appended to `Args.java`

```
private class ArgumentMarshaler {
    private boolean booleanValue = false;

    public void setBoolean(boolean value) {
      booleanValue = value;
    }

    public boolean getBoolean() {return booleanValue;}
  }

  private class BooleanArgumentMarshaler extends ArgumentMarshaler {
  }

  private class StringArgumentMarshaler extends ArgumentMarshaler {
  }

  private class IntegerArgumentMarshaler extends ArgumentMarshaler {
  }
}
```

Clearly, this wasn't going to break anything. So then I made the simplest modification I could, one that would break as little as possible. I changed the `HashMap` for the `Boolean` arguments to take an `ArgumentMarshaler`.

```
private Map<Character, ArgumentMarshaler> booleanArgs =
  new HashMap<Character, ArgumentMarshaler>();
```

This broke a few statements, which I quickly fixed.

```
...
  private void parseBooleanSchemaElement(char elementId) {
    booleanArgs.put(elementId, new BooleanArgumentMarshaler());
  }
..
```

```
  private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.get(argChar).setBoolean(value);
  }
...
  public boolean getBoolean(char arg) {
    return falseIfNull(booleanArgs.get(arg).getBoolean());
  }
```

Notice how these changes are in exactly the areas that I mentioned before: the `parse`, `set`, and `get` for the argument type. Unfortunately, small as this change was, some of the tests started failing. If you look carefully at `getBoolean`, you'll see that if you call it with `'y,'` but there is no `y` argument, then `booleanArgs.get('y')` will return `null`, and the function will throw a `NullPointerException`. The `falseIfNull` function had been used to protect against this, but the change I made caused that function to become irrelevant.

Incrementalism demanded that I get this working quickly before making any other changes. Indeed, the fix was not too difficult. I just had to move the check for `null`. It was no longer the `boolean` being `null` that I needed to check; it was the `ArgumentMarshaller`.

First, I removed the `falseIfNull` call in the `getBoolean` function. It was useless now, so I also eliminated the function itself. The tests still failed in the same way, so I was confident that I hadn't introduced any new errors.

```
public boolean getBoolean(char arg) {
  return booleanArgs.get(arg).getBoolean();
}
```

Next, I split the function into two lines and put the `ArgumentMarshaller` into its own variable named `argumentMarshaller`. I didn't care for the long variable name; it was badly redundant and cluttered up the function. So I shortened it to `am` [N5].

```
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = booleanArgs.get(arg);
  return am.getBoolean();
}
```

And then I put in the null detection logic.

```
public boolean getBoolean(char arg) {
  Args.ArgumentMarshaler am = booleanArgs.get(arg);
  return am != null && am.getBoolean();
}
```

String Arguments

Adding `String` arguments was very similar to adding `boolean` arguments. I had to change the `HashMap` and get the `parse`, `set`, and `get` functions working. There shouldn't be any surprises in what follows except, perhaps, that I seem to be putting all the marshalling implementation in the `ArgumentMarshaller` base class instead of distributing it to the derivatives.

```
  private Map<Character, ArgumentMarshaler> stringArgs =
      new HashMap<Character, ArgumentMarshaler>();
...
```

```
  private void parseStringSchemaElement(char elementId) {
    stringArgs.put(elementId, new StringArgumentMarshaler());
  }
...
  private void setStringArg(char argChar) throws ArgsException {
    currentArgument++;
    try {
      stringArgs.get(argChar).setString(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
      valid = false;
      errorArgumentId = argChar;
      errorCode = ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }
...
  public String getString(char arg) {
    Args.ArgumentMarshaler am = stringArgs.get(arg);
    return am == null ? "" : am.getString();
  }
...
  private class ArgumentMarshaler {
    private boolean booleanValue = false;
    private String stringValue;

    public void setBoolean(boolean value) {
      booleanValue = value;
    }

    public boolean getBoolean() {
      return booleanValue;
    }

    public void setString(String s) {
      stringValue = s;
    }

    public String getString() {
      return stringValue == null ? "" : stringValue;
    }
  }
```

Again, these changes were made one at a time and in such a way that the tests kept running, if not passing. When a test broke, I made sure to get it passing again before continuing with the next change.

By now you should be able to see my intent. Once I get all the current marshalling behavior into the `ArgumentMarshaler` base class, I'm going to start pushing that behavior down into the derivatives. This will allow me to keep everything running while I gradually change the shape of this program.

The obvious next step was to move the `int` argument functionality into the `ArgumentMarshaler`. Again, there weren't any surprises.

```
private Map<Character, ArgumentMarshaler> intArgs =
     new HashMap<Character, ArgumentMarshaler>();
...
```

```
  private void parseIntegerSchemaElement(char elementId) {
    intArgs.put(elementId, new IntegerArgumentMarshaler());
  }
...
  private void setIntArg(char argChar) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
      parameter = args[currentArgument];
      intArgs.get(argChar).setInteger(Integer.parseInt(parameter));
    } catch (ArrayIndexOutOfBoundsException e) {
      valid = false;
      errorArgumentId = argChar;
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (NumberFormatException e) {
      valid = false;
      errorArgumentId = argChar;
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw new ArgsException();
    }
  }
...
  public int getInt(char arg) {
    Args.ArgumentMarshaler am = intArgs.get(arg);
    return am == null ? 0 : am.getInteger();
  }
...
  private class ArgumentMarshaler {
    private boolean booleanValue = false;
    private String stringValue;
    private int integerValue;

    public void setBoolean(boolean value) {
      booleanValue = value;
    }

    public boolean getBoolean() {
      return booleanValue;
    }

    public void setString(String s) {
      stringValue = s;
    }

    public String getString() {
      return stringValue == null ? "" : stringValue;
    }

    public void setInteger(int i) {
      integerValue = i;
    }

    public int getInteger() {
      return integerValue;
    }
  }
```

With all the marshalling moved to the `ArgumentMarshaler`, I started pushing functionality into the derivatives. The first step was to move the `setBoolean` function into the `BooleanArgumentMarshaller` and make sure it got called correctly. So I created an abstract `set` method.

```
private abstract class ArgumentMarshaler {
  protected boolean booleanValue = false;
  private String stringValue;
  private int integerValue;

  public void setBoolean(boolean value) {
    booleanValue = value;
  }

  public boolean getBoolean() {
    return booleanValue;
  }

  public void setString(String s) {
    stringValue = s;
  }

  public String getString() {
    return stringValue == null ? "" : stringValue;
  }

  public void setInteger(int i) {
    integerValue = i;
  }

  public int getInteger() {
    return integerValue;
  }

  public abstract void set(String s);
}
```

Then I implemented the `set` method in `BooleanArgumentMarshaller`.

```
private class BooleanArgumentMarshaler extends ArgumentMarshaler {
  public void set(String s) {
    booleanValue = true;
  }
}
```

And finally I replaced the call to `setBoolean` with a call to `set`.

```
private void setBooleanArg(char argChar, boolean value) {
    booleanArgs.get(argChar).set("true");
  }
```

The tests all still passed. Because this change caused `set` to be deployed to the `BooleanArgumentMarshaler`, I removed the `setBoolean` method from the `ArgumentMarshaler` base class.

Notice that the abstract `set` function takes a `String` argument, but the implementation in the `BooleanArgumentMarshaller` does not use it. I put that argument in there because I knew that the `StringArgumentMarshaller` and `IntegerArgumentMarshaller` *would* use it.

Next, I wanted to deploy the `get` method into `BooleanArgumentMarshaler`. Deploying `get` functions is always ugly because the return type has to be `Object`, and in this case needs to be cast to a `Boolean`.

```
public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am != null && (Boolean)am.get();
  }
```

Just to get this to compile, I added the `get` function to the `ArgumentMarshaler`.

```
private abstract class ArgumentMarshaler {
    ...

    public Object get() {
      return null;
    }
  }
```

This compiled and obviously failed the tests. Getting the tests working again was simply a matter of making `get` abstract and implementing it in `BooleanAgumentMarshaler`.

```
private abstract class ArgumentMarshaler {
    protected boolean booleanValue = false;
    ...

    public abstract Object get();
  }

  private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    public void set(String s) {
      booleanValue = true;
    }

    public Object get() {
      return booleanValue;
    }
  }
```

Once again the tests passed. So both `get` and `set` deploy to the `BooleanArgumentMarshaler`! This allowed me to remove the old `getBoolean` function from `ArgumentMarshaler`, move the protected `booleanValue` variable down to `BooleanArgumentMarshaler`, and make it `private`.

I did the same pattern of changes for `Strings`. I deployed both `set` and `get`, deleted the unused functions, and moved the variables.

```
  private void setStringArg(char argChar) throws ArgsException {
    currentArgument++;
    try {
      stringArgs.get(argChar).set(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
      valid = false;
      errorArgumentId = argChar;
      errorCode = ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }
```

```
...
  public String getString(char arg) {
    Args.ArgumentMarshaler am = stringArgs.get(arg);
    return am == null ? "" : (String) am.get();
  }
...
  private abstract class ArgumentMarshaler {
    private int integerValue;

    public void setInteger(int i) {
      integerValue = i;
    }

    public int getInteger() {
      return integerValue;
    }

    public abstract void set(String s);

    public abstract Object get();
  }

  private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    private boolean booleanValue = false;

    public void set(String s) {
      booleanValue = true;
    }

    public Object get() {
      return booleanValue;
    }
  }

  private class StringArgumentMarshaler extends ArgumentMarshaler {
    private String stringValue = "";

    public void set(String s) {
      stringValue = s;
    }

    public Object get() {
      return stringValue;
    }
  }

  private class IntegerArgumentMarshaler extends ArgumentMarshaler {

    public void set(String s) {

    }

    public Object get() {
      return null;
    }
  }
}
```

Finally, I repeated the process for `integers`. This was just a little more complicated because `integers` needed to be parsed, and the `parse` operation can throw an exception. But the result is better because the whole concept of `NumberFormatException` got buried in the `IntegerArgumentMarshaler`.

```
private boolean isIntArg(char argChar) {return intArgs.containsKey(argChar);}

  private void setIntArg(char argChar) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
      parameter = args[currentArgument];
      intArgs.get(argChar).set(parameter);
    } catch (ArrayIndexOutOfBoundsException e) {
      valid = false;
      errorArgumentId = argChar;
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw e;
    }
  }
...
  private void setBooleanArg(char argChar) {
    try {
      booleanArgs.get(argChar).set("true");
    } catch (ArgsException e) {
    }
  }
...
  public int getInt(char arg) {
    Args.ArgumentMarshaler am = intArgs.get(arg);
    return am == null ? 0 : (Integer) am.get();
  }
...
  private abstract class ArgumentMarshaler {
    public abstract void set(String s) throws ArgsException;
    public abstract Object get();
  }
...
  private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;

    public void set(String s) throws ArgsException {
      try {
        intValue = Integer.parseInt(s);
      } catch (NumberFormatException e) {
        throw new ArgsException();
      }
    }

    public Object get() {
      return intValue;
    }
  }
```

Of course, the tests continued to pass. Next, I got rid of the three different maps up at the top of the algorithm. This made the whole system much more generic. However, I couldn't get rid of them just by deleting them because that would break the system. Instead, I added a new `Map` for the `ArgumentMarshaler` and then one by one changed the methods to use it instead of the three original maps.

```
public class Args {
...
  private Map<Character, ArgumentMarshaler> booleanArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> stringArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> intArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
...
  private void parseBooleanSchemaElement(char elementId) {
    ArgumentMarshaler m = new BooleanArgumentMarshaler();
    booleanArgs.put(elementId, m);
    marshalers.put(elementId, m);
  }

  private void parseIntegerSchemaElement(char elementId) {
    ArgumentMarshaler m = new IntegerArgumentMarshaler();
    intArgs.put(elementId, m);
    marshalers.put(elementId, m);
  }

  private void parseStringSchemaElement(char elementId) {
    ArgumentMarshaler m = new StringArgumentMarshaler();
    stringArgs.put(elementId, m);
    marshalers.put(elementId, m);
  }
```

Of course the tests all still passed. Next, I changed `isBooleanArg` from this:

```
  private boolean isBooleanArg(char argChar) {
    return booleanArgs.containsKey(argChar);
  }
```

to this:

```
  private boolean isBooleanArg(char argChar) {
    ArgumentMarshaler m = marshalers.get(argChar);
    return m instanceof BooleanArgumentMarshaler;
  }
```

The tests still passed. So I made the same change to `isIntArg` and `isStringArg`.

```
  private boolean isIntArg(char argChar) {
    ArgumentMarshaler m = marshalers.get(argChar);
    return m instanceof IntegerArgumentMarshaler;
  }

  private boolean isStringArg(char argChar) {
    ArgumentMarshaler m = marshalers.get(argChar);
    return m instanceof StringArgumentMarshaler;
  }
```

The tests still passed. So I eliminated all the duplicate calls to `marshalers.get` as follows:

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (isBooleanArg(m))
    setBooleanArg(argChar);
  else if (isStringArg(m))
    setStringArg(argChar);
  else if (isIntArg(m))
    setIntArg(argChar);
  else
    return false;

  return true;
}

private boolean isIntArg(ArgumentMarshaler m) {
  return m instanceof IntegerArgumentMarshaler;
}

private boolean isStringArg(ArgumentMarshaler m) {
  return m instanceof StringArgumentMarshaler;
}

private boolean isBooleanArg(ArgumentMarshaler m) {
  return m instanceof BooleanArgumentMarshaler;
}
```

This left no good reason for the three `isxxxArg` methods. So I inlined them:

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m instanceof BooleanArgumentMarshaler)
    setBooleanArg(argChar);
  else if (m instanceof StringArgumentMarshaler)
    setStringArg(argChar);
  else if (m instanceof IntegerArgumentMarshaler)
    setIntArg(argChar);
  else
    return false;

  return true;
}
```

Next, I started using the `marshalers` map in the `set` functions, breaking the use of the other three maps. I started with the `booleans`.

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m instanceof BooleanArgumentMarshaler)
    setBooleanArg(m);
  else if (m instanceof StringArgumentMarshaler)
    setStringArg(argChar);
  else if (m instanceof IntegerArgumentMarshaler)
    setIntArg(argChar);
  else
    return false;
```

```
    return true;
  }
...
  private void setBooleanArg(ArgumentMarshaler m) {
    try {
      m.set("true"); // was: booleanArgs.get(argChar).set("true");
    } catch (ArgsException e) {
    }
  }
```

The tests still passed, so I did the same with `Strings` and `Integers`. This allowed me to integrate some of the ugly exception management code into the `setArgument` function.

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  try {
    if (m instanceof BooleanArgumentMarshaler)
      setBooleanArg(m);
    else if (m instanceof StringArgumentMarshaler)
      setStringArg(m);
    else if (m instanceof IntegerArgumentMarshaler)
      setIntArg(m);
    else
      return false;
  } catch (ArgsException e) {
    valid = false;
    errorArgumentId = argChar;
    throw e;
  }
  return true;
}

private void setIntArg(ArgumentMarshaler m) throws ArgsException {
  currentArgument++;
  String parameter = null;
  try {
    parameter = args[currentArgument];
    m.set(parameter);
  } catch (ArrayIndexOutOfBoundsException e) {
    errorCode = ErrorCode.MISSING_INTEGER;
    throw new ArgsException();
  } catch (ArgsException e) {
    errorParameter = parameter;
    errorCode = ErrorCode.INVALID_INTEGER;
    throw e;
  }
}

private void setStringArg(ArgumentMarshaler m) throws ArgsException {
  currentArgument++;
  try {
    m.set(args[currentArgument]);
  } catch (ArrayIndexOutOfBoundsException e) {
    errorCode = ErrorCode.MISSING_STRING;
    throw new ArgsException();
  }
}
```

I was close to being able to remove the three old maps. First, I needed to change the `getBoolean` function from this:

```
  public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = booleanArgs.get(arg);
    return am != null && (Boolean) am.get();
  }
```

to this:

```
public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    boolean b = false;
    try {
      b = am != null && (Boolean) am.get();
    } catch (ClassCastException e) {
      b = false;
    }
    return b;
  }
```

This last change might have been a surprise. Why did I suddenly decide to deal with the `ClassCastException`? The reason is that I have a set of unit tests and a separate set of acceptance tests written in FitNesse. It turns out that the FitNesse tests made sure that if you called `getBoolean` on a nonboolean argument, you got a `false`. The unit tests did not. Up to this point I had only been running the unit tests.[2]

This last change allowed me to pull out another use of the `boolean` map:

```
  private void parseBooleanSchemaElement(char elementId) {
    ArgumentMarshaler m = new BooleanArgumentMarshaler();
    booleanArgs.put(elementId, m);
    marshalers.put(elementId, m);
  }
```

And now we can delete the `boolean` map.

```
public class Args {
...
  private Map<Character, ArgumentMarshaler> booleanArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> stringArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> intArgs =
    new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
...
```

Next, I migrated the `String` and `Integer` arguments in the same manner and did a little cleanup with the `booleans`.

```
  private void parseBooleanSchemaElement(char elementId) {
    marshalers.put(elementId, new BooleanArgumentMarshaler());
  }
```

2. To prevent further surprises of this kind, I added a new unit test that invoked all the FitNesse tests.

```
  private void parseIntegerSchemaElement(char elementId) {
    marshalers.put(elementId, new IntegerArgumentMarshaler());
  }

  private void parseStringSchemaElement(char elementId) {
    marshalers.put(elementId, new StringArgumentMarshaler());
  }
...
  public String getString(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? "" : (String) am.get();
    } catch (ClassCastException e) {
      return "";
    }
  }

  public int getInt(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? 0 : (Integer) am.get();
    } catch (Exception e) {
      return 0;
    }
  }
...
public class Args {
...
- private Map<Character, ArgumentMarshaler> stringArgs =
-   new HashMap<Character, ArgumentMarshaler>();
- private Map<Character, ArgumentMarshaler> intArgs =
-   new HashMap<Character, ArgumentMarshaler>();
  private Map<Character, ArgumentMarshaler> marshalers =
  new HashMap<Character, ArgumentMarshaler>();
...
```

Next, I inlined the three `parse` methods because they didn't do much anymore:

```
  private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
      marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (isStringSchemaElement(elementTail))
      marshalers.put(elementId, new StringArgumentMarshaler());
    else if (isIntegerSchemaElement(elementTail)) {
      marshalers.put(elementId, new IntegerArgumentMarshaler());
    } else {
      throw new ParseException(String.format(
       "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
    }
  }
```

Okay, so now let's look at the whole picture again. Listing 14-12 shows the current form of the `Args` class.

Listing 14-12
Args.java (After first refactoring)

```
package com.objectmentor.utilities.getopts;

import java.text.ParseException;
import java.util.*;

public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, ArgumentMarshaler> marshalers =
   new HashMap<Character, ArgumentMarshaler>();
  private Set<Character> argsFound = new HashSet<Character>();
  private int currentArgument;
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;

  private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT}

  public Args(String schema, String[] args) throws ParseException {
    this.schema = schema;
    this.args = args;
    valid = parse();
  }

  private boolean parse() throws ParseException {
    if (schema.length() == 0 && args.length == 0)
      return true;
    parseSchema();
    try {
      parseArguments();
    } catch (ArgsException e) {
    }
    return valid;
  }

  private boolean parseSchema() throws ParseException {
    for (String element : schema.split(",")) {
      if (element.length() > 0) {
        String trimmedElement = element.trim();
        parseSchemaElement(trimmedElement);
      }
    }
    return true;
  }

  private void parseSchemaElement(String element) throws ParseException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (isBooleanSchemaElement(elementTail))
      marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (isStringSchemaElement(elementTail))
      marshalers.put(elementId, new StringArgumentMarshaler());
```

Listing 14-12 (continued)
Args.java (After first refactoring)

```
    else if (isIntegerSchemaElement(elementTail)) {
      marshalers.put(elementId, new IntegerArgumentMarshaler());
    } else {
      throw new ParseException(String.format(
       "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
    }
  }

  private void validateSchemaElementId(char elementId) throws ParseException {
    if (!Character.isLetter(elementId)) {
      throw new ParseException(
       "Bad character:" + elementId + "in Args format: " + schema, 0);
    }
  }

  private boolean isStringSchemaElement(String elementTail) {
    return elementTail.equals("*");
  }

  private boolean isBooleanSchemaElement(String elementTail) {
    return elementTail.length() == 0;
  }

  private boolean isIntegerSchemaElement(String elementTail) {
    return elementTail.equals("#");
  }

  private boolean parseArguments() throws ArgsException {
    for (currentArgument=0; currentArgument<args.length; currentArgument++) {
      String arg = args[currentArgument];
      parseArgument(arg);
    }
    return true;
  }

  private void parseArgument(String arg) throws ArgsException {
    if (arg.startsWith("-"))
      parseElements(arg);
  }

  private void parseElements(String arg) throws ArgsException {
    for (int i = 1; i < arg.length(); i++)
      parseElement(arg.charAt(i));
  }

  private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
      argsFound.add(argChar);
    else {
      unexpectedArguments.add(argChar);
      errorCode = ErrorCode.UNEXPECTED_ARGUMENT;
      valid = false;
    }
  }
```

Listing 14-12 (continued)
Args.java (After first refactoring)

```
  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    try {
      if (m instanceof BooleanArgumentMarshaler)
        setBooleanArg(m);
      else if (m instanceof StringArgumentMarshaler)
        setStringArg(m);
      else if (m instanceof IntegerArgumentMarshaler)
        setIntArg(m);
      else
        return false;
    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      throw e;
    }
    return true;
  }

  private void setIntArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    String parameter = null;
    try {
      parameter = args[currentArgument];
      m.set(parameter);
    } catch (ArrayIndexOutOfBoundsException e) {
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (ArgsException e) {
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw e;
    }
  }

  private void setStringArg(ArgumentMarshaler m) throws ArgsException {
    currentArgument++;
    try {
      m.set(args[currentArgument]);
    } catch (ArrayIndexOutOfBoundsException e) {
      errorCode = ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }

  private void setBooleanArg(ArgumentMarshaler m) {
    try {
      m.set("true");
    } catch (ArgsException e) {
    }
  }

  public int cardinality() {
    return argsFound.size();
  }
```

Listing 14-12 (continued)
Args.java (After first refactoring)

```
  public String usage() {
    if (schema.length() > 0)
      return "-[" + schema + "]";
    else
      return "";
  }

  public String errorMessage() throws Exception {
    switch (errorCode) {
      case OK:
        throw new Exception("TILT: Should not get here.");
      case UNEXPECTED_ARGUMENT:
        return unexpectedArgumentMessage();
      case MISSING_STRING:
        return String.format("Could not find string parameter for -%c.",
                             errorArgumentId);
      case INVALID_INTEGER:
        return String.format("Argument -%c expects an integer but was '%s'.",
                             errorArgumentId, errorParameter);
      case MISSING_INTEGER:
        return String.format("Could not find integer parameter for -%c.",
                             errorArgumentId);
    }
    return "";
  }

  private String unexpectedArgumentMessage() {
    StringBuffer message = new StringBuffer("Argument(s) -");
    for (char c : unexpectedArguments) {
      message.append(c);
    }
    message.append(" unexpected.");

    return message.toString();
  }

  public boolean getBoolean(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    boolean b = false;
    try {
      b = am != null && (Boolean) am.get();
    } catch (ClassCastException e) {
      b = false;
    }
    return b;
  }

  public String getString(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? "" : (String) am.get();
    } catch (ClassCastException e) {
      return "";
    }
  }
```

Listing 14-12 (continued)
Args.java (After first refactoring)

```
  public int getInt(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? 0 : (Integer) am.get();
    } catch (Exception e) {
      return 0;
    }
  }

  public boolean has(char arg) {
    return argsFound.contains(arg);
  }

  public boolean isValid() {
    return valid;
  }

  private class ArgsException extends Exception {
  }

  private abstract class ArgumentMarshaler {
    public abstract void set(String s) throws ArgsException;
    public abstract Object get();
  }

  private class BooleanArgumentMarshaler extends ArgumentMarshaler {
    private boolean booleanValue = false;

    public void set(String s) {
      booleanValue = true;
    }

    public Object get() {
      return booleanValue;
    }
  }

  private class StringArgumentMarshaler extends ArgumentMarshaler {
    private String stringValue = "";

    public void set(String s) {
      stringValue = s;
    }

    public Object get() {
      return stringValue;
    }
  }

  private class IntegerArgumentMarshaler extends ArgumentMarshaler {
    private int intValue = 0;

    public void set(String s) throws ArgsException {
      try {
        intValue = Integer.parseInt(s);
```

Listing 14-12 (continued)
Args.java (After first refactoring)

```
      } catch (NumberFormatException e) {
        throw new ArgsException();
      }
    }

    public Object get() {
      return intValue;
    }
  }
}
```

After all that work, this is a bit disappointing. The structure is a bit better, but we still have all those variables up at the top; there's still a horrible type-case in `setArgument`; and all those `set` functions are really ugly. Not to mention all the error processing. We still have a lot of work ahead of us.

I'd really like to get rid of that type-case up in `setArgument` [G23]. What I'd like in `setArgument` is a single call to `ArgumentMarshaler.set`. This means I need to push `setIntArg`, `setStringArg`, and `setBooleanArg` down into the appropriate `ArgumentMarshaler` derivatives. But there is a problem.

If you look closely at `setIntArg`, you'll notice that it uses two instance variables: `args` and `currentArg`. To move `setIntArg` down into `BooleanArgumentMarshaler`, I'll have to pass both `args` and `currentArgs` as function arguments. That's dirty [F1]. I'd rather pass one argument instead of two. Fortunately, there is a simple solution. We can convert the `args` array into a `list` and pass an `Iterator` down to the `set` functions. The following took me ten steps, passing all the tests after each. But I'll just show you the result. You should be able to figure out what most of the tiny little steps were.

```
public class Args {
  private String schema;
  private String[] args;
  private boolean valid = true;
  private Set<Character> unexpectedArguments = new TreeSet<Character>();
  private Map<Character, ArgumentMarshaler> marshalers =
   new HashMap<Character, ArgumentMarshaler>();
  private Set<Character> argsFound = new HashSet<Character>();
  private Iterator<String> currentArgument;
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;
  private List<String> argsList;

  private enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT}

  public Args(String schema, String[] args) throws ParseException {
    this.schema = schema;
    argsList = Arrays.asList(args);
    valid = parse();
  }
```

```
  private boolean parse() throws ParseException {
    if (schema.length() == 0 && argsList.size() == 0)
      return true;
    parseSchema();
    try {
      parseArguments();
    } catch (ArgsException e) {
    }
    return valid;
  }
---
  private boolean parseArguments() throws ArgsException {
    for (currentArgument = argsList.iterator(); currentArgument.hasNext();) {
      String arg = currentArgument.next();
      parseArgument(arg);
    }

    return true;
  }
---
  private void setIntArg(ArgumentMarshaler m) throws ArgsException {
    String parameter = null;
    try {
      parameter = currentArgument.next();
      m.set(parameter);
    } catch (NoSuchElementException e) {
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (ArgsException e) {
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw e;
    }
  }

  private void setStringArg(ArgumentMarshaler m) throws ArgsException {
    try {
      m.set(currentArgument.next());
    } catch (NoSuchElementException e) {
      errorCode = ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }
```

These were simple changes that kept all the tests passing. Now we can start moving the `set` functions down into the appropriate derivatives. First, I need to make the following change in `setArgument`:

```
 private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
      return false;
    try {
      if (m instanceof BooleanArgumentMarshaler)
        setBooleanArg(m);
      else if (m instanceof StringArgumentMarshaler)
        setStringArg(m);
      else if (m instanceof IntegerArgumentMarshaler)
        setIntArg(m);
```

```
      else
        return false;
    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      throw e;
    }
    return true;
  }
```

This change is important because we want to completely eliminate the `if-else` chain. Therefore, we needed to get the error condition out of it.

Now we can start to move the `set` functions. The `setBooleanArg` function is trivial, so we'll prepare that one first. Our goal is to change the `setBooleanArg` function to simply forward to the `BooleanArgumentMarshaler`.

```
private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
      return false;
    try {
      if (m instanceof BooleanArgumentMarshaler)
        setBooleanArg(m, currentArgument);
      else if (m instanceof StringArgumentMarshaler)
        setStringArg(m);
      else if (m instanceof IntegerArgumentMarshaler)
        setIntArg(m);

    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      throw e;
    }
    return true;
  }
---
  private void setBooleanArg(ArgumentMarshaler m,
                             Iterator<String> currentArgument)
                             throws ArgsException {
  try {
      m.set("true");
  catch (ArgsException e) {
  }
  }
```

Didn't we just put that exception processing in? Putting things in so you can take them out again is pretty common in refactoring. The smallness of the steps and the need to keep the tests running means that you move things around a lot. Refactoring is a lot like solving a Rubik's cube. There are lots of little steps required to achieve a large goal. Each step enables the next.

Why did we pass that `iterator` when `setBooleanArg` certainly doesn't need it? Because `setIntArg` and `setStringArg` will! And because I want to deploy all three of these functions through an abstract method in `ArgumentMarshaller`, I need to pass it to `setBooleanArg`.

So now `setBooleanArg` is useless. If there were a `set` function in `ArgumentMarshaler`, we could call it directly. So it's time to make that function! The first step is to add the new abstract method to `ArgumentMarshaler`.

```
private abstract class ArgumentMarshaler {
  public abstract void set(Iterator<String> currentArgument)
                       throws ArgsException;
  public abstract void set(String s) throws ArgsException;
  public abstract Object get();
}
```

Of course this breaks all the derivatives. So let's implement the new method in each.

```
private class BooleanArgumentMarshaler extends ArgumentMarshaler {
  private boolean booleanValue = false;

  public void set(Iterator<String> currentArgument) throws ArgsException {
    booleanValue = true;
  }

  public void set(String s) {
    booleanValue = true;
  }

  public Object get() {
    return booleanValue;
  }
}

private class StringArgumentMarshaler extends ArgumentMarshaler {
  private String stringValue = "";

  public void set(Iterator<String> currentArgument) throws ArgsException {
  }

  public void set(String s) {
    stringValue = s;
  }

  public Object get() {
    return stringValue;
  }
}

private class IntegerArgumentMarshaler extends ArgumentMarshaler {
  private int intValue = 0;

  public void set(Iterator<String> currentArgument) throws ArgsException {
  }

  public void set(String s) throws ArgsException {
    try {
      intValue = Integer.parseInt(s);
    } catch (NumberFormatException e) {
      throw new ArgsException();
    }
  }
```

```
    public Object get() {
      return intValue;
    }
  }
```

And now we can eliminate `setBooleanArg`!

```
private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
      return false;
    try {
      if (m instanceof BooleanArgumentMarshaler)
        m.set(currentArgument);
      else if (m instanceof StringArgumentMarshaler)
        setStringArg(m);
      else if (m instanceof IntegerArgumentMarshaler)
        setIntArg(m);

    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      throw e;
    }
    return true;
  }
```

The tests all pass, and the `set` function is deploying to `BooleanArgumentMarshaler`!

Now we can do the same for `Strings` and `Integers`.

```
  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
      return false;
    try {
      if (m instanceof BooleanArgumentMarshaler)
        m.set(currentArgument);
      else if (m instanceof StringArgumentMarshaler)
        m.set(currentArgument);
      else if (m instanceof IntegerArgumentMarshaler)
        m.set(currentArgument);

    } catch (ArgsException e) {
      valid = false;
      errorArgumentId = argChar;
      throw e;
    }
    return true;
  }
---
  private class StringArgumentMarshaler extends ArgumentMarshaler {
    private String stringValue = "";

    public void set(Iterator<String> currentArgument) throws ArgsException {
      try {
        stringValue = currentArgument.next();
      } catch (NoSuchElementException e) {
        errorCode = ErrorCode.MISSING_STRING;
```

```
      throw new ArgsException();
    }
  }

  public void set(String s) {
  }

  public Object get() {
    return stringValue;
  }
}

private class IntegerArgumentMarshaler extends ArgumentMarshaler {
  private int intValue = 0;

public void set(Iterator<String> currentArgument) throws ArgsException {
  String parameter = null;
  try {
    parameter = currentArgument.next();
    set(parameter);
  } catch (NoSuchElementException e) {
    errorCode = ErrorCode.MISSING_INTEGER;
    throw new ArgsException();
  } catch (ArgsException e) {
    errorParameter = parameter;
    errorCode = ErrorCode.INVALID_INTEGER;
    throw e;
  }
}

  public void set(String s) throws ArgsException {
    try {
      intValue = Integer.parseInt(s);
    } catch (NumberFormatException e) {
      throw new ArgsException();
    }
  }

  public Object get() {
    return intValue;
  }
}
```

And so the *coup de grace*: The type-case can be removed! Touche!

```
private boolean setArgument(char argChar) throws ArgsException {
  ArgumentMarshaler m = marshalers.get(argChar);
  if (m == null)
    return false;
  try {
    m.set(currentArgument);
    return true;
  } catch (ArgsException e) {
    valid = false;
    errorArgumentId = argChar;
    throw e;
  }
}
```

Now we can get rid of some crufty functions in `IntegerArgumentMarshaler` and clean it up a bit.

```
private class IntegerArgumentMarshaler extends ArgumentMarshaler {
  private int intValue = 0

  public void set(Iterator<String> currentArgument) throws ArgsException {
    String parameter = null;
    try {
      parameter = currentArgument.next();
      intValue = Integer.parseInt(parameter);
    } catch (NoSuchElementException e) {
      errorCode = ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (NumberFormatException e) {
      errorParameter = parameter;
      errorCode = ErrorCode.INVALID_INTEGER;
      throw new ArgsException();
    }
  }

  public Object get() {
    return intValue;
  }
}
```

We can also turn **`ArgumentMarshaler`** into an interface.

```
private interface ArgumentMarshaler {
  void set(Iterator<String> currentArgument) throws ArgsException;
  Object get();
}
```

So now let's see how easy it is to add a new argument type to our structure. It should require very few changes, and those changes should be isolated. First, we begin by adding a new test case to check that the `double` argument works correctly.

```
public void testSimpleDoublePresent() throws Exception {
  Args args = new Args("x##", new String[] {"-x","42.3"});
  assertTrue(args.isValid());
  assertEquals(1, args.cardinality());
  assertTrue(args.has('x'));
  assertEquals(42.3, args.getDouble('x'), .001);
}
```

Now we clean up the schema parsing code and add the `##` detection for the `double` argument type.

```
private void parseSchemaElement(String element) throws ParseException {
  char elementId = element.charAt(0);
  String elementTail = element.substring(1);
  validateSchemaElementId(elementId);
  if (elementTail.length() == 0)
    marshalers.put(elementId, new BooleanArgumentMarshaler());
  else if (elementTail.equals("*"))
    marshalers.put(elementId, new StringArgumentMarshaler());
  else if (elementTail.equals("#"))
    marshalers.put(elementId, new IntegerArgumentMarshaler());
```

```
    else if (elementTail.equals("##"))
      marshalers.put(elementId, new DoubleArgumentMarshaler());
    else
      throw new ParseException(String.format(
        "Argument: %c has invalid format: %s.", elementId, elementTail), 0);
  }
```

Next, we write the `DoubleArgumentMarshaler` class.

```
  private class DoubleArgumentMarshaler implements ArgumentMarshaler {
    private double doubleValue = 0;

    public void set(Iterator<String> currentArgument) throws ArgsException {
      String parameter = null;
      try {
        parameter = currentArgument.next();
        doubleValue = Double.parseDouble(parameter);
      } catch (NoSuchElementException e) {
        errorCode = ErrorCode.MISSING_DOUBLE;
        throw new ArgsException();
      } catch (NumberFormatException e) {
        errorParameter = parameter;
        errorCode = ErrorCode.INVALID_DOUBLE;
        throw new ArgsException();
      }
    }

    public Object get() {
      return doubleValue;
    }
  }
```

This forces us to add a new `ErrorCode`.

```
  private enum ErrorCode {
      OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT,
      MISSING_DOUBLE, INVALID_DOUBLE}
```

And we need a `getDouble` function.

```
  public double getDouble(char arg) {
    Args.ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? 0 : (Double) am.get();
    } catch (Exception e) {
      return 0.0;
    }
  }
```

And all the tests pass! That was pretty painless. So now let's make sure all the error processing works correctly. The next test case checks that an error is declared if an unparseable string is fed to a `##` argument.

```
  public void testInvalidDouble() throws Exception {
    Args args = new Args("x##", new String[] {"-x","Forty two"});
    assertFalse(args.isValid());
    assertEquals(0, args.cardinality());
    assertFalse(args.has('x'));
    assertEquals(0, args.getInt('x'));
```

```
    assertEquals("Argument -x expects a double but was 'Forty two'.",
                 args.errorMessage());
  }
---
  public String errorMessage() throws Exception {
    switch (errorCode) {
      case OK:
        throw new Exception("TILT: Should not get here.");
      case UNEXPECTED_ARGUMENT:
        return unexpectedArgumentMessage();
      case MISSING_STRING:
        return String.format("Could not find string parameter for -%c.",
                             errorArgumentId);
      case INVALID_INTEGER:
        return String.format("Argument -%c expects an integer but was '%s'.",
                             errorArgumentId, errorParameter);
      case MISSING_INTEGER:
        return String.format("Could not find integer parameter for -%c.",
                             errorArgumentId);
      case INVALID_DOUBLE:
        return String.format("Argument -%c expects a double but was '%s'.",
                             errorArgumentId, errorParameter);
      case MISSING_DOUBLE:
        return String.format("Could not find double parameter for -%c.",
                             errorArgumentId);
    }
    return "";
  }
```

And the tests pass. The next test makes sure we detect a missing `double` argument properly.

```
public void testMissingDouble() throws Exception {
  Args args = new Args("x##", new String[]{"-x"});
  assertFalse(args.isValid());
  assertEquals(0, args.cardinality());
  assertFalse(args.has('x'));
  assertEquals(0.0, args.getDouble('x'), 0.01);
  assertEquals("Could not find double parameter for -x.",
               args.errorMessage());
}
```

This passes as expected. We wrote it simply for completeness.

The exception code is pretty ugly and doesn't really belong in the `Args` class. We are also throwing out `ParseException`, which doesn't really belong to us. So let's merge all the exceptions into a single `ArgsException` class and move it into its own module.

```
public class ArgsException extends Exception {
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;

  public ArgsException() {}

  public ArgsException(String message) {super(message);}

  public enum ErrorCode {
    OK, MISSING_STRING, MISSING_INTEGER, INVALID_INTEGER, UNEXPECTED_ARGUMENT,
    MISSING_DOUBLE, INVALID_DOUBLE}
}
---
```

```
public class Args {
  ...
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ArgsException.ErrorCode errorCode = ArgsException.ErrorCode.OK;
  private List<String> argsList;

  public Args(String schema, String[] args) throws ArgsException {
    this.schema = schema;
    argsList = Arrays.asList(args);
    valid = parse();
  }

  private boolean parse() throws ArgsException {
    if (schema.length() == 0 && argsList.size() == 0)
      return true;
    parseSchema();
    try {
      parseArguments();
    } catch (ArgsException e) {
    }
    return valid;
  }

  private boolean parseSchema() throws ArgsException {
    ...
  }

  private void parseSchemaElement(String element) throws ArgsException {
    ...
    else
      throw new ArgsException(
        String.format("Argument: %c has invalid format: %s.",
                      elementId,elementTail));
  }

  private void validateSchemaElementId(char elementId) throws ArgsException {
    if (!Character.isLetter(elementId)) {
      throw new ArgsException(
        "Bad character:" + elementId + "in Args format: " + schema);
    }
  }

  ...

  private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
      argsFound.add(argChar);
    else {
      unexpectedArguments.add(argChar);
      errorCode = ArgsException.ErrorCode.UNEXPECTED_ARGUMENT;
      valid = false;
    }
  }

  ...
```

```
private class StringArgumentMarshaler implements ArgumentMarshaler {
  private String stringValue = "";

  public void set(Iterator<String> currentArgument) throws ArgsException {
    try {
      stringValue = currentArgument.next();
    } catch (NoSuchElementException e) {
      errorCode = ArgsException.ErrorCode.MISSING_STRING;
      throw new ArgsException();
    }
  }

  public Object get() {
    return stringValue;
  }
}

private class IntegerArgumentMarshaler implements ArgumentMarshaler {
  private int intValue = 0;

  public void set(Iterator<String> currentArgument) throws ArgsException {
    String parameter = null;
    try {
      parameter = currentArgument.next();
      intValue = Integer.parseInt(parameter);
    } catch (NoSuchElementException e) {
      errorCode = ArgsException.ErrorCode.MISSING_INTEGER;
      throw new ArgsException();
    } catch (NumberFormatException e) {
      errorParameter = parameter;
      errorCode = ArgsException.ErrorCode.INVALID_INTEGER;
      throw new ArgsException();
    }
  }

  public Object get() {
    return intValue;
  }
}

private class DoubleArgumentMarshaler implements ArgumentMarshaler {
  private double doubleValue = 0;

  public void set(Iterator<String> currentArgument) throws ArgsException {
    String parameter = null;
    try {
      parameter = currentArgument.next();
      doubleValue = Double.parseDouble(parameter);
    } catch (NoSuchElementException e) {
      errorCode = ArgsException.ErrorCode.MISSING_DOUBLE;
      throw new ArgsException();
    } catch (NumberFormatException e) {
      errorParameter = parameter;
      errorCode = ArgsException.ErrorCode.INVALID_DOUBLE;
      throw new ArgsException();
    }
  }
```

```
      public Object get() {
        return doubleValue;
      }
    }
  }
```

This is nice. Now the only exception thrown by `Args` is `ArgsException`. Moving `ArgsException` into its own module means that we can move a lot of the miscellaneous error support code into that module and out of the `Args` module. It provides a natural and obvious place to put all that code and will really help us clean up the `Args` module going forward.

So now we have completely separated the exception and error code from the `Args` module. (See Listing 14-13 through Listing 14-16.) This was achieved through a series of about 30 tiny steps, keeping the tests passing between each step.

Listing 14-13
ArgsTest.java

```
package com.objectmentor.utilities.args;

import junit.framework.TestCase;

public class ArgsTest extends TestCase {
  public void testCreateWithNoSchemaOrArguments() throws Exception {
    Args args = new Args("", new String[0]);
    assertEquals(0, args.cardinality());
  }

  public void testWithNoSchemaButWithOneArgument() throws Exception {
    try {
      new Args("", new String[]{"-x"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                   e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
    }
  }

  public void testWithNoSchemaButWithMultipleArguments() throws Exception {
    try {
      new Args("", new String[]{"-x", "-y"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                   e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
    }

  }

  public void testNonLetterSchema() throws Exception {
    try {
      new Args("*", new String[]{});
      fail("Args constructor should have thrown exception");
    } catch (ArgsException e) {
```

Listing 14-13 (continued)
ArgsTest.java

```
      assertEquals(ArgsException.ErrorCode.INVALID_ARGUMENT_NAME,
                   e.getErrorCode());
      assertEquals('*', e.getErrorArgumentId());
    }
  }

  public void testInvalidArgumentFormat() throws Exception {
    try {
      new Args("f~", new String[]{});
      fail("Args constructor should have throws exception");
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.INVALID_FORMAT, e.getErrorCode());
      assertEquals('f', e.getErrorArgumentId());
    }
  }

  public void testSimpleBooleanPresent() throws Exception {
    Args args = new Args("x", new String[]{"-x"});
    assertEquals(1, args.cardinality());
    assertEquals(true, args.getBoolean('x'));
  }

  public void testSimpleStringPresent() throws Exception {
    Args args = new Args("x*", new String[]{"-x", "param"});
    assertEquals(1, args.cardinality());
    assertTrue(args.has('x'));
    assertEquals("param", args.getString('x'));
  }

  public void testMissingStringArgument() throws Exception {
    try {
      new Args("x*", new String[]{"-x"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.MISSING_STRING, e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
    }
  }

  public void testSpacesInFormat() throws Exception {
    Args args = new Args("x, y", new String[]{"-xy"});
    assertEquals(2, args.cardinality());
    assertTrue(args.has('x'));
    assertTrue(args.has('y'));
  }

  public void testSimpleIntPresent() throws Exception {
    Args args = new Args("x#", new String[]{"-x", "42"});
    assertEquals(1, args.cardinality());
    assertTrue(args.has('x'));
    assertEquals(42, args.getInt('x'));
  }

  public void testInvalidInteger() throws Exception {
    try {
      new Args("x#", new String[]{"-x", "Forty two"});
```

Listing 14-13 (continued)
ArgsTest.java

```
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.INVALID_INTEGER, e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
      assertEquals("Forty two", e.getErrorParameter());
    }

  }

  public void testMissingInteger() throws Exception {
    try {
      new Args("x#", new String[]{"-x"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.MISSING_INTEGER, e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
    }
  }

  public void testSimpleDoublePresent() throws Exception {
    Args args = new Args("x##", new String[]{"-x", "42.3"});
    assertEquals(1, args.cardinality());
    assertTrue(args.has('x'));
    assertEquals(42.3, args.getDouble('x'), .001);
  }

  public void testInvalidDouble() throws Exception {
    try {
      new Args("x##", new String[]{"-x", "Forty two"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.INVALID_DOUBLE, e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
      assertEquals("Forty two", e.getErrorParameter());
    }
  }

  public void testMissingDouble() throws Exception {
    try {
      new Args("x##", new String[]{"-x"});
      fail();
    } catch (ArgsException e) {
      assertEquals(ArgsException.ErrorCode.MISSING_DOUBLE, e.getErrorCode());
      assertEquals('x', e.getErrorArgumentId());
    }
  }
}
```

Listing 14-14
ArgsExceptionTest.java

```
public class ArgsExceptionTest extends TestCase {
  public void testUnexpectedMessage() throws Exception {
    ArgsException e =
```

Listing 14-14 (continued)
ArgsExceptionTest.java

```
      new ArgsException(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                        'x', null);
    assertEquals("Argument -x unexpected.", e.errorMessage());
  }

  public void testMissingStringMessage() throws Exception {
    ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING_STRING,
                                        'x', null);
    assertEquals("Could not find string parameter for -x.", e.errorMessage());
  }

  public void testInvalidIntegerMessage() throws Exception {
    ArgsException e =
      new ArgsException(ArgsException.ErrorCode.INVALID_INTEGER,
                        'x', "Forty two");
    assertEquals("Argument -x expects an integer but was 'Forty two'.",
                 e.errorMessage());
  }

  public void testMissingIntegerMessage() throws Exception {
    ArgsException e =
      new ArgsException(ArgsException.ErrorCode.MISSING_INTEGER, 'x', null);
    assertEquals("Could not find integer parameter for -x.", e.errorMessage());
  }

  public void testInvalidDoubleMessage() throws Exception {
    ArgsException e = new ArgsException(ArgsException.ErrorCode.INVALID_DOUBLE,
                                        'x', "Forty two");
    assertEquals("Argument -x expects a double but was 'Forty two'.",
                 e.errorMessage());
  }

  public void testMissingDoubleMessage() throws Exception {
    ArgsException e = new ArgsException(ArgsException.ErrorCode.MISSING_DOUBLE,
                                        'x', null);
    assertEquals("Could not find double parameter for -x.", e.errorMessage());
  }
}
```

Listing 14-15
ArgsException.java

```
public class ArgsException extends Exception {
  private char errorArgumentId = '\0';
  private String errorParameter = "TILT";
  private ErrorCode errorCode = ErrorCode.OK;

  public ArgsException() {}

  public ArgsException(String message) {super(message);}

  public ArgsException(ErrorCode errorCode) {
    this.errorCode = errorCode;
  }
```

Listing 14-15 (continued)
ArgsException.java

```
  public ArgsException(ErrorCode errorCode, String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
  }

  public ArgsException(ErrorCode errorCode, char errorArgumentId,
                       String errorParameter) {
    this.errorCode = errorCode;
    this.errorParameter = errorParameter;
    this.errorArgumentId = errorArgumentId;
  }

  public char getErrorArgumentId() {
    return errorArgumentId;
  }

  public void setErrorArgumentId(char errorArgumentId) {
    this.errorArgumentId = errorArgumentId;
  }

  public String getErrorParameter() {
    return errorParameter;
  }

  public void setErrorParameter(String errorParameter) {
    this.errorParameter = errorParameter;
  }

  public ErrorCode getErrorCode() {
    return errorCode;
  }

  public void setErrorCode(ErrorCode errorCode) {
    this.errorCode = errorCode;
  }

  public String errorMessage() throws Exception {
    switch (errorCode) {
      case OK:
        throw new Exception("TILT: Should not get here.");
      case UNEXPECTED_ARGUMENT:
        return String.format("Argument -%c unexpected.", errorArgumentId);
      case MISSING_STRING:
        return String.format("Could not find string parameter for -%c.",
                             errorArgumentId);
      case INVALID_INTEGER:
        return String.format("Argument -%c expects an integer but was '%s'.",
                             errorArgumentId, errorParameter);
      case MISSING_INTEGER:
        return String.format("Could not find integer parameter for -%c.",
                             errorArgumentId);
      case INVALID_DOUBLE:
        return String.format("Argument -%c expects a double but was '%s'.",
                             errorArgumentId, errorParameter);
```

Listing 14-15 (continued)
ArgsException.java

```
      case MISSING_DOUBLE:
        return String.format("Could not find double parameter for -%c.",
                             errorArgumentId);
    }
    return "";
  }

  public enum ErrorCode {
    OK, INVALID_FORMAT, UNEXPECTED_ARGUMENT, INVALID_ARGUMENT_NAME,
    MISSING_STRING,
    MISSING_INTEGER, INVALID_INTEGER,
    MISSING_DOUBLE, INVALID_DOUBLE}
}
```

Listing 14-16
Args.java

```
public class Args {
  private String schema;
  private Map<Character, ArgumentMarshaler> marshalers =
    new HashMap<Character, ArgumentMarshaler>();
  private Set<Character> argsFound = new HashSet<Character>();
  private Iterator<String> currentArgument;
  private List<String> argsList;

  public Args(String schema, String[] args) throws ArgsException {
    this.schema = schema;
    argsList = Arrays.asList(args);
    parse();
  }

  private void parse() throws ArgsException {
    parseSchema();
    parseArguments();
  }

  private boolean parseSchema() throws ArgsException {
    for (String element : schema.split(",")) {
      if (element.length() > 0) {
        parseSchemaElement(element.trim());
      }
    }
    return true;
  }

  private void parseSchemaElement(String element) throws ArgsException {
    char elementId = element.charAt(0);
    String elementTail = element.substring(1);
    validateSchemaElementId(elementId);
    if (elementTail.length() == 0)
      marshalers.put(elementId, new BooleanArgumentMarshaler());
    else if (elementTail.equals("*"))
      marshalers.put(elementId, new StringArgumentMarshaler());
```

Listing 14-16 (continued)
Args.java

```
    else if (elementTail.equals("#"))
      marshalers.put(elementId, new IntegerArgumentMarshaler());
    else if (elementTail.equals("##"))
      marshalers.put(elementId, new DoubleArgumentMarshaler());
    else
      throw new ArgsException(ArgsException.ErrorCode.INVALID_FORMAT,
                              elementId, elementTail);
  }

  private void validateSchemaElementId(char elementId) throws ArgsException {
    if (!Character.isLetter(elementId)) {
      throw new ArgsException(ArgsException.ErrorCode.INVALID_ARGUMENT_NAME,
                              elementId, null);
    }
  }

  private void parseArguments() throws ArgsException {
    for (currentArgument = argsList.iterator(); currentArgument.hasNext();) {
      String arg = currentArgument.next();
      parseArgument(arg);
    }
  }

  private void parseArgument(String arg) throws ArgsException {
    if (arg.startsWith("-"))
      parseElements(arg);
  }

  private void parseElements(String arg) throws ArgsException {
    for (int i = 1; i < arg.length(); i++)
      parseElement(arg.charAt(i));
  }

  private void parseElement(char argChar) throws ArgsException {
    if (setArgument(argChar))
      argsFound.add(argChar);
    else {
      throw new ArgsException(ArgsException.ErrorCode.UNEXPECTED_ARGUMENT,
                              argChar, null);
    }
  }

  private boolean setArgument(char argChar) throws ArgsException {
    ArgumentMarshaler m = marshalers.get(argChar);
    if (m == null)
      return false;
    try {
      m.set(currentArgument);
      return true;
    } catch (ArgsException e) {
      e.setErrorArgumentId(argChar);
      throw e;
    }
  }
```

Listing 14-16 (continued)
`Args.java`

```
  public int cardinality() {
    return argsFound.size();
  }

  public String usage() {
    if (schema.length() > 0)
      return "-[" + schema + "]";
    else
      return "";
  }

  public boolean getBoolean(char arg) {
    ArgumentMarshaler am = marshalers.get(arg);
    boolean b = false;
    try {
      b = am != null && (Boolean) am.get();
    } catch (ClassCastException e) {
      b = false;
    }
    return b;
  }

  public String getString(char arg) {
    ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? "" : (String) am.get();
    } catch (ClassCastException e) {
      return "";
    }
  }

  public int getInt(char arg) {
    ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? 0 : (Integer) am.get();
    } catch (Exception e) {
      return 0;
    }
  }


  public double getDouble(char arg) {
    ArgumentMarshaler am = marshalers.get(arg);
    try {
      return am == null ? 0 : (Double) am.get();
    } catch (Exception e) {
      return 0.0;
    }
  }

  public boolean has(char arg) {
    return argsFound.contains(arg);
  }
}
```

The majority of the changes to the `Args` class were deletions. A lot of code just got moved out of `Args` and put into `ArgsException`. Nice. We also moved all the `ArgumentMarshaller`s into their own files. Nicer!

Much of good software design is simply about partitioning—creating appropriate places to put different kinds of code. This separation of concerns makes the code much simpler to understand and maintain.

Of special interest is the `errorMessage` method of `ArgsException`. Clearly it was a violation of the SRP to put the error message formatting into `Args`. `Args` should be about the processing of arguments, not about the format of the error messages. However, does it really make sense to put the error message formatting code into `ArgsException`?

Frankly, it's a compromise. Users who don't like the error messages supplied by `ArgsException` will have to write their own. But the convenience of having canned error messages already prepared for you is not insignificant.

By now it should be clear that we are within striking distance of the final solution that appeared at the start of this chapter. I'll leave the final transformations to you as an exercise.

Conclusion

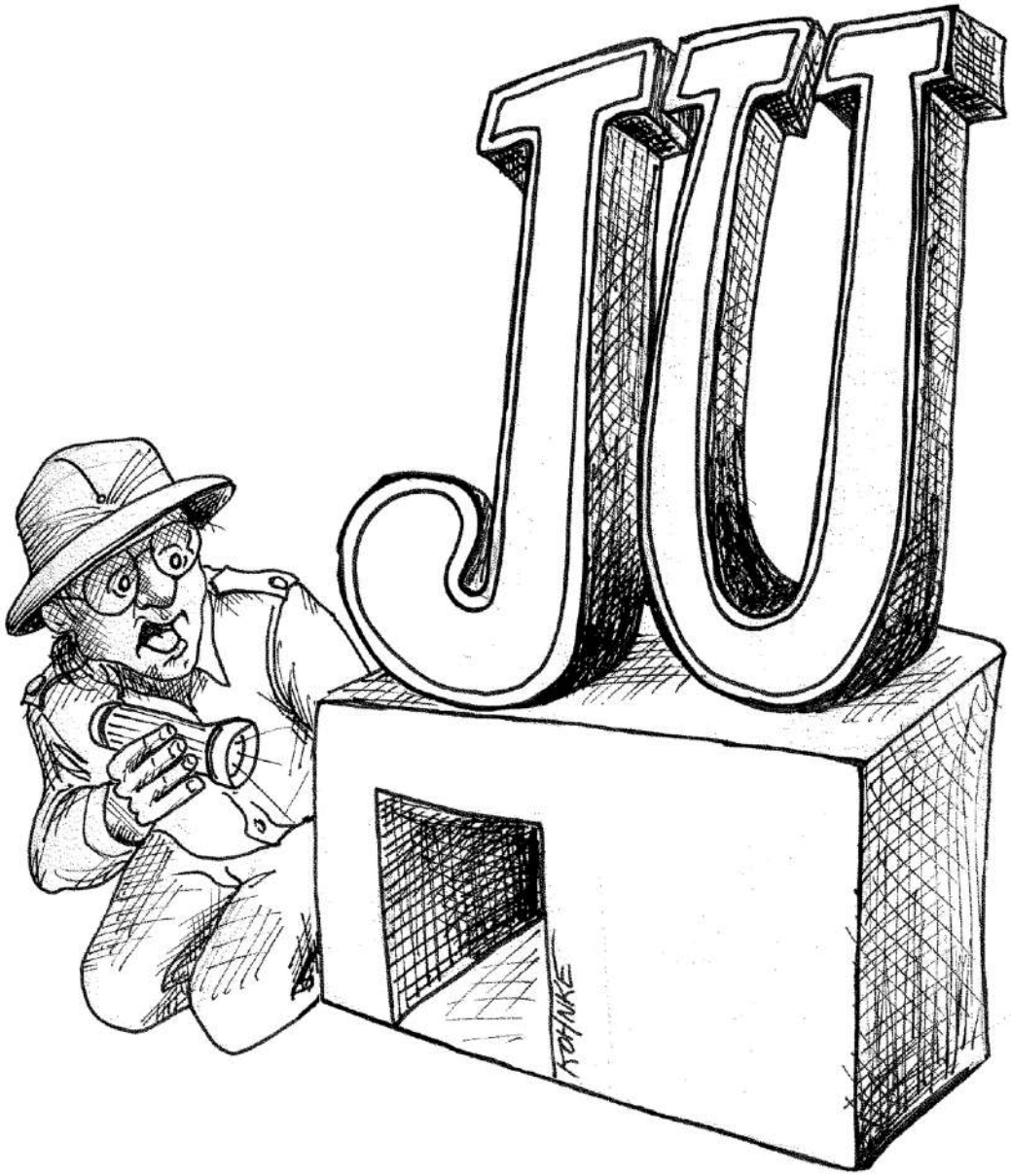
It is not enough for code to work. Code that works is often badly broken. Programmers who satisfy themselves with merely working code are behaving unprofessionally. They may fear that they don't have time to improve the structure and design of their code, but I disagree. Nothing has a more profound and long-term degrading effect upon a development project than bad code. Bad schedules can be redone, bad requirements can be redefined. Bad team dynamics can be repaired. But bad code rots and ferments, becoming an inexorable weight that drags the team down. Time and time again I have seen teams grind to a crawl because, in their haste, they created a malignant morass of code that forever thereafter dominated their destiny.

Of course bad code can be cleaned up. But it's very expensive. As code rots, the modules insinuate themselves into each other, creating lots of hidden and tangled dependencies. Finding and breaking old dependencies is a long and arduous task. On the other hand, keeping code clean is relatively easy. If you made a mess in a module in the morning, it is easy to clean it up in the afternoon. Better yet, if you made a mess five minutes ago, it's very easy to clean it up right now.

So the solution is to continuously keep your code as clean and simple as it can be. Never let the rot get started.

15

JUnit Internals



JUnit is one of the most famous of all Java frameworks. As frameworks go, it is simple in conception, precise in definition, and elegant in implementation. But what does the code look like? In this chapter we'll critique an example drawn from the JUnit framework.

The JUnit Framework

JUnit has had many authors, but it began with Kent Beck and Eric Gamma together on a plane to Atlanta. Kent wanted to learn Java, and Eric wanted to learn about Kent's Smalltalk testing framework. "What could be more natural to a couple of geeks in cramped quarters than to pull out our laptops and start coding?"[1] After three hours of high-altitude work, they had written the basics of JUnit.

The module we'll look at is the clever bit of code that helps identify string comparison errors. This module is called `ComparisonCompactor`. Given two strings that differ, such as `ABCDE` and `ABXDE`, it will expose the difference by generating a string such as `<...B[X]D...>`.

I could explain it further, but the test cases do a better job. So take a look at Listing 15-1 and you will understand the requirements of this module in depth. While you are at it, critique the structure of the tests. Could they be simpler or more obvious?

Listing 15-1
`ComparisonCompactorTest.java`

```
package junit.tests.framework;

import junit.framework.ComparisonCompactor;
import junit.framework.TestCase;

public class ComparisonCompactorTest extends TestCase {

  public void testMessage() {
    String failure= new ComparisonCompactor(0, "b", "c").compact("a");
    assertTrue("a expected:<[b]> but was:<[c]>".equals(failure));
  }

  public void testStartSame() {
    String failure= new ComparisonCompactor(1, "ba", "bc").compact(null);
    assertEquals("expected:<b[a]> but was:<b[c]>", failure);
  }

  public void testEndSame() {
    String  failure= new ComparisonCompactor(1, "ab", "cb").compact(null);
    assertEquals("expected:<[a]b> but was:<[c]b>", failure);
  }

  public void testSame() {
    String failure= new ComparisonCompactor(1, "ab", "ab").compact(null);
    assertEquals("expected:<ab> but was:<ab>", failure);
  }

  public void testNoContextStartAndEndSame() {
    String failure= new ComparisonCompactor(0, "abc", "adc").compact(null);
    assertEquals("expected:<...[b]...> but was:<...[d]...>", failure);
  }
```

1. *JUnit Pocket Guide*, Kent Beck, O'Reilly, 2004, p. 43.

Listing 15-1 (continued)
ComparisonCompactorTest.java

```
  public void testStartAndEndContext() {
    String failure= new ComparisonCompactor(1, "abc", "adc").compact(null);
    assertEquals("expected:<a[b]c> but was:<a[d]c>", failure);
  }

  public void testStartAndEndContextWithEllipses() {
    String failure=
      new ComparisonCompactor(1, "abcde", "abfde").compact(null);
    assertEquals("expected:<...b[c]d...> but was:<...b[f]d...>", failure);
  }

  public void testComparisonErrorStartSameComplete() {
    String failure= new ComparisonCompactor(2, "ab", "abc").compact(null);
    assertEquals("expected:<ab[]> but was:<ab[c]>", failure);
  }

  public void testComparisonErrorEndSameComplete() {
    String failure= new ComparisonCompactor(0, "bc", "abc").compact(null);
    assertEquals("expected:<[]...> but was:<[a]...>", failure);
  }

  public void testComparisonErrorEndSameCompleteContext() {
    String failure= new ComparisonCompactor(2, "bc", "abc").compact(null);
    assertEquals("expected:<[]bc> but was:<[a]bc>", failure);
  }

  public void testComparisonErrorOverlapingMatches() {
    String failure= new ComparisonCompactor(0, "abc", "abbc").compact(null);
    assertEquals("expected:<...[]...> but was:<...[b]...>", failure);
  }

  public void testComparisonErrorOverlapingMatchesContext() {
    String failure= new ComparisonCompactor(2, "abc", "abbc").compact(null);
    assertEquals("expected:<ab[]c> but was:<ab[b]c>", failure);
  }

  public void testComparisonErrorOverlapingMatches2() {
    String failure= new ComparisonCompactor(0, "abcdde",
"abcde").compact(null);
    assertEquals("expected:<...[d]...> but was:<...[]...>", failure);
  }

  public void testComparisonErrorOverlapingMatches2Context() {
    String failure=
      new ComparisonCompactor(2, "abcdde", "abcde").compact(null);
    assertEquals("expected:<...cd[d]e> but was:<...cd[]e>", failure);
  }

  public void testComparisonErrorWithActualNull() {
    String failure= new ComparisonCompactor(0, "a", null).compact(null);
    assertEquals("expected:<a> but was:<null>", failure);
  }

  public void testComparisonErrorWithActualNullContext() {
    String failure= new ComparisonCompactor(2, "a", null).compact(null);
```

Listing 15-1 (continued)
ComparisonCompactorTest.java

```
      assertEquals("expected:<a> but was:<null>", failure);
    }

    public void testComparisonErrorWithExpectedNull() {
      String failure= new ComparisonCompactor(0, null, "a").compact(null);
      assertEquals("expected:<null> but was:<a>", failure);
    }

    public void testComparisonErrorWithExpectedNullContext() {
      String failure= new ComparisonCompactor(2, null, "a").compact(null);
      assertEquals("expected:<null> but was:<a>", failure);
    }

    public void testBug609972() {
      String failure= new ComparisonCompactor(10, "S&P500", "0").compact(null);
      assertEquals("expected:<[S&P50]0> but was:<[]0>", failure);
    }
  }
```

I ran a code coverage analysis on the `ComparisonCompactor` using these tests. The code is 100 percent covered. Every line of code, every `if` statement and `for` loop, is executed by the tests. This gives me a high degree of confidence that the code works and a high degree of respect for the craftsmanship of the authors.

The code for `ComparisonCompactor` is in Listing 15-2. Take a moment to look over this code. I think you'll find it to be nicely partitioned, reasonably expressive, and simple in structure. Once you are done, then we'll pick the nits together.

Listing 15-2
ComparisonCompactor.java (Original)

```
package junit.framework;

public class ComparisonCompactor {

  private static final String ELLIPSIS = "...";
  private static final String DELTA_END = "]";
  private static final String DELTA_START = "[";

  private int fContextLength;
  private String fExpected;
  private String fActual;
  private int fPrefix;
  private int fSuffix;

  public ComparisonCompactor(int contextLength,
                             String expected,
                               String actual) {
    fContextLength = contextLength;
    fExpected = expected;
    fActual = actual;
  }
```

Listing 15-2 (continued)
ComparisonCompactor.java (Original)

```
public String compact(String message) {
  if (fExpected == null || fActual == null || areStringsEqual())
    return Assert.format(message, fExpected, fActual);

  findCommonPrefix();
  findCommonSuffix();
  String expected = compactString(fExpected);
  String actual = compactString(fActual);
  return Assert.format(message, expected, actual);
}

private String compactString(String source) {
  String result = DELTA_START +
                  source.substring(fPrefix, source.length() -
                    fSuffix + 1) + DELTA_END;
  if (fPrefix > 0)
    result = computeCommonPrefix() + result;
  if (fSuffix > 0)
    result = result + computeCommonSuffix();
  return result;
}

private void findCommonPrefix() {
  fPrefix = 0;
  int end = Math.min(fExpected.length(), fActual.length());
  for (; fPrefix < end; fPrefix++) {
    if (fExpected.charAt(fPrefix) != fActual.charAt(fPrefix))
      break;
  }
}

private void findCommonSuffix() {
  int expectedSuffix = fExpected.length() - 1;
  int actualSuffix = fActual.length() - 1;
  for (;
       actualSuffix >= fPrefix && expectedSuffix >= fPrefix;
        actualSuffix--, expectedSuffix--) {
    if (fExpected.charAt(expectedSuffix) != fActual.charAt(actualSuffix))
      break;
  }
  fSuffix = fExpected.length() - expectedSuffix;
}

private String computeCommonPrefix() {
  return (fPrefix > fContextLength ? ELLIPSIS : "") +
           fExpected.substring(Math.max(0, fPrefix - fContextLength),
                               fPrefix);
}

private String computeCommonSuffix() {
  int end = Math.min(fExpected.length() - fSuffix + 1 + fContextLength,
                     fExpected.length());
  return fExpected.substring(fExpected.length() - fSuffix + 1, end) +
         (fExpected.length() - fSuffix + 1 < fExpected.length() -
          fContextLength ? ELLIPSIS : "");
}
```

Listing 15-2 (continued)
ComparisonCompactor.java (Original)

```
  private boolean areStringsEqual() {
    return fExpected.equals(fActual);
  }
}
```

You might have a few complaints about this module. There are some long expressions and some strange +1s and so forth. But overall this module is pretty good. After all, it might have looked like Listing 15-3.

Listing 15-3
ComparisonCompator.java (defactored)

```
package junit.framework;

public class ComparisonCompactor {
  private int ctxt;
  private String s1;
  private String s2;
  private int pfx;
  private int sfx;

  public ComparisonCompactor(int ctxt, String s1, String s2) {
    this.ctxt = ctxt;
    this.s1 = s1;
    this.s2 = s2;
  }

  public String compact(String msg) {
    if (s1 == null || s2 == null || s1.equals(s2))
      return Assert.format(msg, s1, s2);

    pfx = 0;
    for (; pfx < Math.min(s1.length(), s2.length()); pfx++) {
      if (s1.charAt(pfx) != s2.charAt(pfx))
        break;
    }
    int sfx1 = s1.length() - 1;
    int sfx2 = s2.length() - 1;
    for (; sfx2 >= pfx && sfx1 >= pfx; sfx2--, sfx1--) {
      if (s1.charAt(sfx1) != s2.charAt(sfx2))
        break;
    }
    sfx = s1.length() - sfx1;
    String cmp1 = compactString(s1);
    String cmp2 = compactString(s2);
    return Assert.format(msg, cmp1, cmp2);
  }

  private String compactString(String s) {
    String result =
      "[" + s.substring(pfx, s.length() - sfx + 1) + "]";
    if (pfx > 0)
      result = (pfx > ctxt ? "..." : "") +
        s1.substring(Math.max(0, pfx - ctxt), pfx) + result;
```

Listing 15-3 (continued)
ComparisonCompator.java (defactored)

```
    if (sfx > 0) {
      int end = Math.min(s1.length() - sfx + 1 + ctxt, s1.length());
      result = result + (s1.substring(s1.length() - sfx + 1, end) +
        (s1.length() - sfx + 1 < s1.length() - ctxt ? "..." : ""));
    }
    return result;
  }

}
```

Even though the authors left this module in very good shape, the *Boy Scout Rule*[2] tells us we should leave it cleaner than we found it. So, how can we improve on the original code in Listing 15-2?

The first thing I don't care for is the `f` prefix for the member variables [N6]. Today's environments make this kind of scope encoding redundant. So let's eliminate all the `f`'s.

```
private int contextLength;
private String expected;
private String actual;
private int prefix;
private int suffix;
```

Next, we have an unencapsulated conditional at the beginning of the `compact` function [G28].

```
public String compact(String message) {
  if (expected == null || actual == null || areStringsEqual())
    return Assert.format(message, expected, actual);

  findCommonPrefix();
  findCommonSuffix();
  String expected = compactString(this.expected);
  String actual = compactString(this.actual);
  return Assert.format(message, expected, actual);
}
```

This conditional should be encapsulated to make our intent clear. So let's extract a method that explains it.

```
public String compact(String message) {
  if (shouldNotCompact())
    return Assert.format(message, expected, actual);

  findCommonPrefix();
  findCommonSuffix();
  String expected = compactString(this.expected);
  String actual = compactString(this.actual);
  return Assert.format(message, expected, actual);
}
```

2. See "The Boy Scout Rule" on page 14.

```
private boolean shouldNotCompact() {
  return expected == null || actual == null || areStringsEqual();
}
```

I don't much care for the `this.expected` and `this.actual` notation in the `compact` function. This happened when we changed the name of `fExpected` to `expected`. Why are there variables in this function that have the same names as the member variables? Don't they represent something else [N4]? We should make the names unambiguous.

```
  String compactExpected = compactString(expected);
  String compactActual = compactString(actual);
```

Negatives are slightly harder to understand than positives [G29]. So let's turn that `if` statement on its head and invert the sense of the conditional.

```
public String compact(String message) {
  if (canBeCompacted()) {
    findCommonPrefix();
    findCommonSuffix();
    String compactExpected = compactString(expected);
    String compactActual = compactString(actual);
    return Assert.format(message, compactExpected, compactActual);
  } else {
    return Assert.format(message, expected, actual);
  }
}

private boolean canBeCompacted() {
  return expected != null && actual != null && !areStringsEqual();
}
```

The name of the function is strange [N7]. Although it does compact the strings, it actually might not compact the strings if `canBeCompacted` returns `false`. So naming this function `compact` hides the side effect of the error check. Notice also that the function returns a formatted message, not just the compacted strings. So the name of the function should really be `formatCompactedComparison`. That makes it read a lot better when taken with the function argument:

```
public String formatCompactedComparison(String message) {
```

The body of the `if` statement is where the true compacting of the expected and actual strings is done. We should extract that as a method named `compactExpectedAndActual`. However, we want the `formatCompactedComparison` function to do all the formatting. The `compact...` function should do nothing but compacting [G30]. So let's split it up as follows:

```
...
  private String compactExpected;
  private String compactActual;

...

  public String formatCompactedComparison(String message) {
    if (canBeCompacted()) {
      compactExpectedAndActual();
      return Assert.format(message, compactExpected, compactActual);
    } else {
```

```
    return Assert.format(message, expected, actual);
  }
}

private void compactExpectedAndActual() {
  findCommonPrefix();
  findCommonSuffix();
  compactExpected = compactString(expected);
  compactActual = compactString(actual);
}
```

Notice that this required us to promote `compactExpected` and `compactActual` to member variables. I don't like the way that the last two lines of the new function return variables, but the first two don't. They aren't using consistent conventions [G11]. So we should change `findCommonPrefix` and `findCommonSuffix` to return the prefix and suffix values.

```
private void compactExpectedAndActual() {
  prefixIndex = findCommonPrefix();
  suffixIndex = findCommonSuffix();
  compactExpected = compactString(expected);
  compactActual = compactString(actual);
}

private int findCommonPrefix() {
  int prefixIndex = 0;
  int end = Math.min(expected.length(), actual.length());
  for (; prefixIndex < end; prefixIndex++) {
    if (expected.charAt(prefixIndex) != actual.charAt(prefixIndex))
      break;
  }
  return prefixIndex;
}

private int findCommonSuffix() {
  int expectedSuffix = expected.length() - 1;
  int actualSuffix = actual.length() - 1;
  for (; actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
       actualSuffix--, expectedSuffix--) {
    if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
      break;
  }
  return expected.length() - expectedSuffix;
}
```

We should also change the names of the member variables to be a little more accurate [N1]; after all, they are both indices.

Careful inspection of `findCommonSuffix` exposes a *hidden temporal coupling* [G31]; it depends on the fact that `prefixIndex` is calculated by `findCommonPrefix`. If these two functions were called out of order, there would be a difficult debugging session ahead. So, to expose this temporal coupling, let's have `findCommonSuffix` take the `prefixIndex` as an argument.

```
private void compactExpectedAndActual() {
  prefixIndex = findCommonPrefix();
  suffixIndex = findCommonSuffix(prefixIndex);
```

```
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
  }

  private int findCommonSuffix(int prefixIndex) {
    int expectedSuffix = expected.length() - 1;
    int actualSuffix = actual.length() - 1;
    for (; actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
         actualSuffix--, expectedSuffix--) {
      if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
        break;
    }
    return expected.length() - expectedSuffix;
  }
```

I'm not really happy with this. The passing of the `prefixIndex` argument is a bit arbitrary [G32]. It works to establish the ordering but does nothing to explain the need for that ordering. Another programmer might undo what we have done because there's no indication that the parameter is really needed. So let's take a different tack.

```
  private void compactExpectedAndActual() {
    findCommonPrefixAndSuffix();
    compactExpected = compactString(expected);
    compactActual = compactString(actual);
  }

  private void findCommonPrefixAndSuffix() {
    findCommonPrefix();
    int expectedSuffix = expected.length() - 1;
    int actualSuffix = actual.length() - 1;
    for (;
         actualSuffix >= prefixIndex && expectedSuffix >= prefixIndex;
         actualSuffix--, expectedSuffix--
      ) {
      if (expected.charAt(expectedSuffix) != actual.charAt(actualSuffix))
        break;
    }
    suffixIndex = expected.length() - expectedSuffix;
  }

  private void findCommonPrefix() {
    prefixIndex = 0;
    int end = Math.min(expected.length(), actual.length());
    for (; prefixIndex < end; prefixIndex++)
      if (expected.charAt(prefixIndex) != actual.charAt(prefixIndex))
        break;
  }
```

We put `findCommonPrefix` and `findCommonSuffix` back the way they were, changing the name of `findCommonSuffix` to `findCommonPrefixAndSuffix` and having it call `findCommonPrefix` before doing anything else. That establishes the temporal nature of the two functions in a much more dramatic way than the previous solution. It also points out how ugly `findCommonPrefixAndSuffix` is. Let's clean it up now.

```
  private void findCommonPrefixAndSuffix() {
    findCommonPrefix();
    int suffixLength = 1;
```

```
    for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) {
      if (charFromEnd(expected, suffixLength) !=
          charFromEnd(actual, suffixLength))
        break;
    }
    suffixIndex = suffixLength;
  }

  private char charFromEnd(String s, int i) {
    return s.charAt(s.length()-i);}

  private boolean suffixOverlapsPrefix(int suffixLength) {
    return actual.length() - suffixLength < prefixLength ||
      expected.length() - suffixLength < prefixLength;
  }
```

This is much better. It exposes that the `suffixIndex` is really the length of the suffix and is not well named. The same is true of the `prefixIndex`, though in that case "index" and "length" are synonymous. Even so, it is more consistent to use "length." The problem is that the `suffixIndex` variable is not zero based; it is 1 based and so is not a true length. This is also the reason that there are all those `+1`s in `computeCommonSuffix` [G33]. So let's fix that. The result is in Listing 15-4.

Listing 15-4
ComparisonCompactor.java (interim)

```
public class ComparisonCompactor {
...
  private int suffixLength;
...
  private void findCommonPrefixAndSuffix() {
    findCommonPrefix();
    suffixLength = 0;
    for (; !suffixOverlapsPrefix(suffixLength); suffixLength++) {
      if (charFromEnd(expected, suffixLength) !=
          charFromEnd(actual, suffixLength))
        break;
    }
  }

  private char charFromEnd(String s, int i) {
    return s.charAt(s.length() - i - 1);
  }

  private boolean suffixOverlapsPrefix(int suffixLength) {
    return actual.length() - suffixLength <= prefixLength ||
      expected.length() - suffixLength <= prefixLength;
  }

...
  private String compactString(String source) {
    String result =
      DELTA_START +
      source.substring(prefixLength, source.length() - suffixLength) +
      DELTA_END;
    if (prefixLength > 0)
      result = computeCommonPrefix() + result;
```

Listing 15-4 (continued)
ComparisonCompactor.java (interim)

```
    if (suffixLength > 0)
      result = result + computeCommonSuffix();
    return result;
  }

...
  private String computeCommonSuffix() {
    int end = Math.min(expected.length() - suffixLength +
      contextLength, expected.length()
    );
    return
      expected.substring(expected.length() - suffixLength, end) +
      (expected.length() - suffixLength <
        expected.length() - contextLength ?
        ELLIPSIS : "");
  }
```

We replaced the `+1`s in `computeCommonSuffix` with a `-1` in `charFromEnd`, where it makes perfect sense, and two `<=` operators in `suffixOverlapsPrefix`, where they also make perfect sense. This allowed us to change the name of `suffixIndex` to `suffixLength`, greatly enhancing the readability of the code.

There is a problem however. As I was eliminating the +1s, I noticed the following line in `compactString`:

```
if (suffixLength > 0)
```

Take a look at it in Listing 15-4. By rights, because `suffixLength` is now one less than it used to be, I should change the `>` operator to a `>=` operator. But that makes no sense. It makes sense *now!* This means that it didn't use to make sense and was probably a bug. Well, not quite a bug. Upon further analysis we see that the `if` statement now prevents a zero length suffix from being appended. Before we made the change, the `if` statement was nonfunctional because `suffixIndex` could never be less than one!

This calls into question *both* `if` statements in `compactString`! It looks as though they could both be eliminated. So let's comment them out and run the tests. They passed! So let's restructure `compactString` to eliminate the extraneous `if` statements and make the function much simpler [G9].

```
private String compactString(String source) {
    return
      computeCommonPrefix() +
      DELTA_START +
      source.substring(prefixLength, source.length() - suffixLength) +
      DELTA_END +
      computeCommonSuffix();
  }
```

This is much better! Now we see that the `compactString` function is simply composing the fragments together. We can probably make this even clearer. Indeed, there are lots of little

cleanups we could do. But rather than drag you through the rest of the changes, I'll just show you the result in Listing 15-5.

Listing 15-5
ComparisonCompactor.java (final)

```
package junit.framework;

public class ComparisonCompactor {

  private static final String ELLIPSIS = "...";
  private static final String DELTA_END = "]";
  private static final String DELTA_START = "[";

  private int contextLength;
  private String expected;
  private String actual;
  private int prefixLength;
  private int suffixLength;

  public ComparisonCompactor(
    int contextLength, String expected, String actual
  ) {
    this.contextLength = contextLength;
    this.expected = expected;
    this.actual = actual;
  }

  public String formatCompactedComparison(String message) {
    String compactExpected = expected;
    String compactActual = actual;
    if (shouldBeCompacted()) {
      findCommonPrefixAndSuffix();
      compactExpected = compact(expected);
      compactActual = compact(actual);
    }
    return Assert.format(message, compactExpected, compactActual);
  }

  private boolean shouldBeCompacted() {
    return !shouldNotBeCompacted();
  }

  private boolean shouldNotBeCompacted() {
    return expected == null ||
           actual == null ||
           expected.equals(actual);
  }

  private void findCommonPrefixAndSuffix() {
    findCommonPrefix();
    suffixLength = 0;
    for (; !suffixOverlapsPrefix(); suffixLength++) {
      if (charFromEnd(expected, suffixLength) !=
          charFromEnd(actual, suffixLength)
      )
```

Listing 15-5 (continued)
ComparisonCompactor.java (final)

```
        break;
    }
  }

  private char charFromEnd(String s, int i) {
    return s.charAt(s.length() - i - 1);
  }

  private boolean suffixOverlapsPrefix() {
    return actual.length() - suffixLength <= prefixLength ||
      expected.length() - suffixLength <= prefixLength;
  }

  private void findCommonPrefix() {
    prefixLength = 0;
    int end = Math.min(expected.length(), actual.length());
    for (; prefixLength < end; prefixLength++)
      if (expected.charAt(prefixLength) != actual.charAt(prefixLength))
        break;
  }

  private String compact(String s) {
    return new StringBuilder()
      .append(startingEllipsis())
      .append(startingContext())
      .append(DELTA_START)
      .append(delta(s))
      .append(DELTA_END)
      .append(endingContext())
      .append(endingEllipsis())
      .toString();
  }

  private String startingEllipsis() {
    return prefixLength > contextLength ? ELLIPSIS : "";
  }

  private String startingContext() {
    int contextStart = Math.max(0, prefixLength - contextLength);
    int contextEnd = prefixLength;
    return expected.substring(contextStart, contextEnd);
  }

  private String delta(String s) {
    int deltaStart = prefixLength;
    int deltaEnd = s.length() - suffixLength;
    return s.substring(deltaStart, deltaEnd);
  }

  private String endingContext() {
    int contextStart = expected.length() - suffixLength;
    int contextEnd =
      Math.min(contextStart + contextLength, expected.length());
    return expected.substring(contextStart, contextEnd);
  }
```

Listing 15-5 (continued)
ComparisonCompactor.java (final)

```
  private String endingEllipsis() {
    return (suffixLength > contextLength ? ELLIPSIS : "");
  }
}
```

This is actually quite pretty. The module is separated into a group of analysis functions and another group of synthesis functions. They are topologically sorted so that the definition of each function appears just after it is used. All the analysis functions appear first, and all the synthesis functions appear last.

If you look carefully, you will notice that I reversed several of the decisions I made earlier in this chapter. For example, I inlined some extracted methods back into `formatCompactedComparison`, and I changed the sense of the `shouldNotBeCompacted` expression. This is typical. Often one refactoring leads to another that leads to the undoing of the first. Refactoring is an iterative process full of trial and error, inevitably converging on something that we feel is worthy of a professional.

Conclusion

And so we have satisfied the Boy Scout Rule. We have left this module a bit cleaner than we found it. Not that it wasn't clean already. The authors had done an excellent job with it. But no module is immune from improvement, and each of us has the responsibility to leave the code a little better than we found it.

16

Refactoring SerialDate

If you go to `http://www.jfree.org/jcommon/index.php`, you will find the JCommon library. Deep within that library there is a package named `org.jfree.date`. Within that package there is a class named `SerialDate`. We are going to explore that class.

The author of `SerialDate` is David Gilbert. David is clearly an experienced and competent programmer. As we shall see, he shows a significant degree of professionalism and discipline within his code. For all intents and purposes, this is "good code." And I am going to rip it to pieces.

This is not an activity of malice. Nor do I think that I am so much better than David that I somehow have a right to pass judgment on his code. Indeed, if you were to find some of my code, I'm sure you could find plenty of things to complain about.

No, this is not an activity of nastiness or arrogance. What I am about to do is nothing more and nothing less than a professional review. It is something that we should all be comfortable doing. And it is something we should welcome when it is done for us. It is only through critiques like these that we will learn. Doctors do it. Pilots do it. Lawyers do it. And we programmers need to learn how to do it too.

One more thing about David Gilbert: David is more than just a good programmer. David had the courage and good will to offer his code to the community at large for free. He placed it out in the open for all to see and invited public usage and public scrutiny. This was well done!

`SerialDate` (Listing B-1, page 349) is a class that represents a date in Java. Why have a class that represents a date, when Java already has `java.util.Date` and `java.util.Calendar`, and others? The author wrote this class in response to a pain that I have often felt myself. The comment in his opening Javadoc (line 67) explains it well. We could quibble about his intention, but I have certainly had to deal with this issue, and I welcome a class that is about dates instead of times.

First, Make It Work

There are some unit tests in a class named `SerialDateTests` (Listing B-2, page 366). The tests all pass. Unfortunately a quick inspection of the tests shows that they don't test everything [T1]. For example, doing a "Find Usages" search on the method `MonthCodeToQuarter` (line 334) indicates that it is not used [F4]. Therefore, the unit tests don't test it.

So I fired up Clover to see what the unit tests covered and what they didn't. Clover reported that the unit tests executed only 91 of the 185 executable statements in `SerialDate` (~50 percent) [T2]. The coverage map looks like a patchwork quilt, with big gobs of unexecuted code littered all through the class.

It was my goal to completely understand and also refactor this class. I couldn't do that without much greater test coverage. So I wrote my own suite of completely independent unit tests (Listing B-4, page 374).

As you look through these tests, you will note that many of them are commented out. These tests didn't pass. They represent behavior that I think `SerialDate` should have. So as I refactor `SerialDate`, I'll be working to make these tests pass too.

Even with some of the tests commented out, Clover reports that the new unit tests are executing 170 (92 percent) out of the 185 executable statements. This is pretty good, and I think we'll be able to get this number higher.

The first few commented-out tests (lines 23-63) were a bit of conceit on my part. The program was not designed to pass these tests, but the behavior seemed obvious [G2] to me.

I'm not sure why the `testWeekdayCodeToString` method was written in the first place, but because it is there, it seems obvious that it should not be case sensitive. Writing these tests was trivial [T3]. Making them pass was even easier; I just changed lines 259 and 263 to use `equalsIgnoreCase`.

I left the tests at line 32 and line 45 commented out because it's not clear to me that the "tues" and "thurs" abbreviations ought to be supported.

The tests on line 153 and line 154 don't pass. Clearly, they should [G2]. We can easily fix this, and the tests on line 163 through line 213, by making the following changes to the `stringToMonthCode` function.

```
457         if ((result < 1) || (result > 12)) {
                result = -1;
458             for (int i = 0; i < monthNames.length; i++) {
459                 if (s.equalsIgnoreCase(shortMonthNames[i])) {
460                     result = i + 1;
461                     break;
462                 }
463                 if (s.equalsIgnoreCase(monthNames[i])) {
464                     result = i + 1;
465                     break;
466                 }
467             }
468         }
```

The commented test on line 318 exposes a bug in the `getFollowingDayOfWeek` method (line 672). December 25th, 2004, was a Saturday. The following Saturday was January 1st, 2005. However, when we run the test, we see that `getFollowingDayOfWeek` returns December 25th as the Saturday that follows December 25th. Clearly, this is wrong [G3],[T1]. We see the problem in line 685. It is a typical boundary condition error [T5]. It should read as follows:

```
685         if (baseDOW >= targetWeekday) {
```

It is interesting to note that this function was the target of an earlier repair. The change history (line 43) shows that "bugs" were fixed in `getPreviousDayOfWeek`, `getFollowing-DayOfWeek`, and `getNearestDayOfWeek` [T6].

The `testGetNearestDayOfWeek` unit test (line 329), which tests the `getNearestDayOfWeek` method (line 705), did not start out as long and exhaustive as it currently is. I added a lot of test cases to it because my initial test cases did not all pass [T6]. You can see the pattern of failure by looking at which test cases are commented out. That pattern is revealing [T7]. It shows that the algorithm fails if the nearest day is in the future. Clearly there is some kind of boundary condition error [T5].

The pattern of test coverage reported by Clover is also interesting [T8]. Line 719 never gets executed! This means that the `if` statement in line 718 is always false. Sure enough, a look at the code shows that this must be true. The `adjust` variable is always negative and so cannot be greater or equal to 4. So this algorithm is just wrong.

The right algorithm is shown below:

```
int delta = targetDOW - base.getDayOfWeek();
int positiveDelta = delta + 7;
int adjust = positiveDelta % 7;
if (adjust > 3)
  adjust -= 7;

return SerialDate.addDays(adjust, base);
```

Finally, the tests at line 417 and line 429 can be made to pass simply by throwing an `IllegalArgumentException` instead of returning an error string from `weekInMonthToString` and `relativeToString`.

With these changes all the unit tests pass, and I believe `SerialDate` now works. So now it's time to make it "right."

Then Make It Right

We are going to walk from the top to the bottom of `SerialDate`, improving it as we go along. Although you won't see this in the discussion, I will be running all of the `JCommon` unit tests, including my improved unit test for `SerialDate`, after every change I make. So rest assured that every change you see here works for all of `JCommon`.

Starting at line 1, we see a ream of comments with license information, copyrights, authors, and change history. I acknowledge that there are certain legalities that need to be addressed, and so the copyrights and licenses must stay. On the other hand, the change history is a leftover from the 1960s. We have source code control tools that do this for us now. This history should be deleted [C1].

The import list starting at line 61 could be shortened by using `java.text.*` and `java.util.*`. [J1]

I wince at the HTML formatting in the Javadoc (line 67). Having a source file with more than one language in it troubles me. This comment has *four* languages in it: Java, English, Javadoc, and html [G1]. With that many languages in use, it's hard to keep things straight. For example, the nice positioning of line 71 and line 72 are lost when the Javadoc is generated, and yet who wants to see `<ul>` and `<li>` in the source code? A better strategy might be to just surround the whole comment with `<pre>` so that the formatting that is apparent in the source code is preserved within the Javadoc.[1]

Line 86 is the class declaration. Why is this class named `SerialDate`? What is the significance of the world "serial"? Is it because the class is derived from `Serializable`? That doesn't seem likely.

1. An even better solution would have been for Javadoc to present all comments as preformatted, so that comments appear the same in both code and document.

I won't keep you guessing. I know why (or at least I think I know why) the word "serial" was used. The clue is in the constants SERIAL_LOWER_BOUND and SERIAL_UPPER_BOUND on line 98 and line 101. An even better clue is in the comment that begins on line 830. This class is named `SerialDate` because it is implemented using a "serial number," which happens to be the number of days since December 30th, 1899.

I have two problems with this. First, the term "serial number" is not really correct. This may be a quibble, but the representation is more of a relative offset than a serial number. The term "serial number" has more to do with product identification markers than dates. So I don't find this name particularly descriptive [N1]. A more descriptive term might be "ordinal."

The second problem is more significant. The name `SerialDate` implies an implementation. This class is an abstract class. There is no need to imply anything at all about the implementation. Indeed, there is good reason to hide the implementation! So I find this name to be at the wrong level of abstraction [N2]. In my opinion, the name of this class should simply be `Date`.

Unfortunately, there are already too many classes in the Java library named `Date`, so this is probably not the best name to choose. Because this class is all about days, instead of time, I considered naming it `Day`, but this name is also heavily used in other places. In the end, I chose `DayDate` as the best compromise.

From now on in this discussion I will use the term `DayDate`. I leave it to you to remember that the listings you are looking at still use `SerialDate`.

I understand why `DayDate` inherits from `Comparable` and `Serializable`. But why does it inherit from `MonthConstants`? The class `MonthConstants` (Listing B-3, page 372) is just a bunch of static final constants that define the months. Inheriting from classes with constants is an old trick that Java programmers used so that they could avoid using expressions like `MonthConstants.January`, but it's a bad idea [J2]. `MonthConstants` should really be an enum.

```
public abstract class DayDate implements Comparable,
                                         Serializable {
  public static enum Month {
    JANUARY(1),
    FEBRUARY(2),
    MARCH(3),
    APRIL(4),
    MAY(5),
    JUNE(6),
    JULY(7),
    AUGUST(8),
    SEPTEMBER(9),
    OCTOBER(10),
    NOVEMBER(11),
    DECEMBER(12);

    Month(int index) {
      this.index = index;
    }
```

```
      public static Month make(int monthIndex) {
        for (Month m : Month.values()) {
          if (m.index == monthIndex)
            return m;
        }
        throw new IllegalArgumentException("Invalid month index " + monthIndex);
      }
      public final int index;
    }
```

Changing `MonthConstants` to this `enum` forces quite a few changes to the `DayDate` class and all it's users. It took me an hour to make all the changes. However, any function that used to take an `int` for a month, now takes a `Month` enumerator. This means we can get rid of the `isValidMonthCode` method (line 326), and all the month code error checking such as that in `monthCodeToQuarter` (line 356) [G5].

Next, we have line 91, `serialVersionUID`. This variable is used to control the serializer. If we change it, then any `DayDate` written with an older version of the software won't be readable anymore and will result in an `InvalidClassException`. If you don't declare the `serialVersionUID` variable, then the compiler automatically generates one for you, and it will be different every time you make a change to the module. I know that all the documents recommend manual control of this variable, but it seems to me that automatic control of serialization is a lot safer [G4]. After all, I'd much rather debug an `InvalidClassException` than the odd behavior that would ensue if I forgot to change the `serialVersionUID`. So I'm going to delete the variable—at least for the time being.[2]

I find the comment on line 93 redundant. Redundant comments are just places to collect lies and misinformation [C2]. So I'm going to get rid of it and its ilk.

The comments at line 97 and line 100 talk about serial numbers, which I discussed earlier [C1]. The variables they describe are the earliest and latest possible dates that `DayDate` can describe. This can be made a bit clearer [N1].

```
    public static final int EARLIEST_DATE_ORDINAL = 2;     // 1/1/1900
    public static final int LATEST_DATE_ORDINAL = 2958465; // 12/31/9999
```

It's not clear to me why `EARLIEST_DATE_ORDINAL` is 2 instead of 0. There is a hint in the comment on line 829 that suggests that this has something to do with the way dates are represented in Microsoft Excel. There is a much deeper insight provided in a derivative of `DayDate` called `SpreadsheetDate` (Listing B-5, page 382). The comment on line 71 describes the issue nicely.

The problem I have with this is that the issue seems to be related to the implementation of `SpreadsheetDate` and has nothing to do with `DayDate`. I conclude from this that

2. Several of the reviewers of this text have taken exception to this decision. They contend that in an open source framework it is better to assert manual control over the serial ID so that minor changes to the software don't cause old serialized dates to be invalid. This is a fair point. However, at least the failure, inconvenient though it might be, has a clear-cut cause. On the other hand, if the author of the class forgets to update the ID, then the failure mode is undefined and might very well be silent. I think the real moral of this story is that you should not expect to deserialize across versions.

`EARLIEST_DATE_ORDINAL` and `LATEST_DATE_ORDINAL` do not really belong in `DayDate` and should be moved to `SpreadsheetDate` [G6].

Indeed, a search of the code shows that these variables are used only within `SpreadsheetDate`. Nothing in `DayDate`, nor in any other class in the `JCommon` framework, uses them. Therefore, I'll move them down into `SpreadsheetDate`.

The next variables, `MINIMUM_YEAR_SUPPORTED`, and `MAXIMUM_YEAR_SUPPORTED` (line 104 and line 107), provide something of a dilemma. It seems clear that if `DayDate` is an abstract class that provides no foreshadowing of implementation, then it should not inform us about a minimum or maximum year. Again, I am tempted to move these variables down into `SpreadsheetDate` [G6]. However, a quick search of the users of these variables shows that one other class uses them: `RelativeDayOfWeekRule` (Listing B-6, page 390). We see that usage at line 177 and line 178 in the `getDate` function, where they are used to check that the argument to `getDate` is a valid year. The dilemma is that a user of an abstract class needs information about its implementation.

What we need to do is provide this information without polluting `DayDate` itself. Usually, we would get implementation information from an instance of a derivative. However, the `getDate` function is not passed an instance of a `DayDate`. It does, however, return such an instance, which means that somewhere it must be creating it. Line 187 through line 205 provide the hint. The `DayDate` instance is being created by one of the three functions, `getPreviousDayOfWeek`, `getNearestDayOfWeek`, or `getFollowingDayOfWeek`. Looking back at the `DayDate` listing, we see that these functions (lines 638–724) all return a date created by `addDays` (line 571), which calls `createInstance` (line 808), which creates a `SpreadsheetDate`! [G7].

It's generally a bad idea for base classes to know about their derivatives. To fix this, we should use the ABSTRACT FACTORY[3] pattern and create a `DayDateFactory`. This factory will create the instances of `DayDate` that we need and can also answer questions about the implementation, such as the maximum and minimum dates.

```
public abstract class DayDateFactory {
  private static DayDateFactory factory = new SpreadsheetDateFactory();
  public static void setInstance(DayDateFactory factory) {
    DayDateFactory.factory = factory;
  }

  protected abstract DayDate _makeDate(int ordinal);
  protected abstract DayDate _makeDate(int day, DayDate.Month month, int year);
  protected abstract DayDate _makeDate(int day, int month, int year);
  protected abstract DayDate _makeDate(java.util.Date date);
  protected abstract int _getMinimumYear();
  protected abstract int _getMaximumYear();

  public static DayDate makeDate(int ordinal) {
    return factory._makeDate(ordinal);
  }
```

3. [GOF].

```
    public static DayDate makeDate(int day, DayDate.Month month, int year) {
      return factory._makeDate(day, month, year);
    }

    public static DayDate makeDate(int day, int month, int year) {
      return factory._makeDate(day, month, year);
    }

    public static DayDate makeDate(java.util.Date date) {
      return factory._makeDate(date);
    }

    public static int getMinimumYear() {
      return factory._getMinimumYear();
    }

    public static int getMaximumYear() {
      return factory._getMaximumYear();
    }
  }
```

This factory class replaces the `createInstance` methods with `makeDate` methods, which improves the names quite a bit [N1]. It defaults to a `SpreadsheetDateFactory` but can be changed at any time to use a different factory. The static methods that delegate to abstract methods use a combination of the SINGLETON,[4] DECORATOR,[5] and ABSTRACT FACTORY patterns that I have found to be useful.

The `SpreadsheetDateFactory` looks like this.

```
  public class SpreadsheetDateFactory extends DayDateFactory {
    public DayDate _makeDate(int ordinal) {
      return new SpreadsheetDate(ordinal);
    }

    public DayDate _makeDate(int day, DayDate.Month month, int year) {
      return new SpreadsheetDate(day, month, year);
    }

    public DayDate _makeDate(int day, int month, int year) {
      return new SpreadsheetDate(day, month, year);
    }

    public DayDate _makeDate(Date date) {
      final GregorianCalendar calendar = new GregorianCalendar();
      calendar.setTime(date);
      return new SpreadsheetDate(
        calendar.get(Calendar.DATE),
        DayDate.Month.make(calendar.get(Calendar.MONTH) + 1),
        calendar.get(Calendar.YEAR));
    }
```

4. Ibid.
5. Ibid.

```
    protected int _getMinimumYear() {
      return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;
    }

    protected int _getMaximumYear() {
      return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;
    }
  }
```

As you can see, I have already moved the `MINIMUM_YEAR_SUPPORTED` and `MAXIMUM_YEAR_SUPPORTED` variables into `SpreadsheetDate`, where they belong [G6].

The next issue in `DayDate` are the day constants beginning at line 109. These should really be another enum [J3]. We've seen this pattern before, so I won't repeat it here. You'll see it in the final listings.

Next, we see a series of tables starting with `LAST_DAY_OF_MONTH` at line 140. My first issue with these tables is that the comments that describe them are redundant [C3]. Their names are sufficient. So I'm going to delete the comments.

There seems to be no good reason that this table isn't private [G8], because there is a static function `lastDayOfMonth` that provides the same data.

The next table, `AGGREGATE_DAYS_TO_END_OF_MONTH`, is a bit more mysterious because it is not used anywhere in the `JCommon` framework [G9]. So I deleted it.

The same goes for `LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH`.

The next table, `AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH`, is used only in `SpreadsheetDate` (line 434 and line 473). This begs the question of whether it should be moved to `SpreadsheetDate`. The argument for not moving it is that the table is not specific to any particular implementation [G6]. On the other hand, no implementation other than `SpreadsheetDate` actually exists, and so the table should be moved close to where it is used [G10].

What settles the argument for me is that to be consistent [G11], we should make the table private and expose it through a function like `julianDateOfLastDayOfMonth`. Nobody seems to need a function like that. Moreover, the table can be moved back to `DayDate` easily if any new implementation of `DayDate` needs it. So I moved it.

The same goes for the table, `LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH`.

Next, we see three sets of constants that can be turned into enums (lines 162–205). The first of the three selects a week within a month. I changed it into an enum named `WeekInMonth`.

```
  public enum WeekInMonth {
      FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0);
      public final int index;

      WeekInMonth(int index) {
        this.index = index;
      }
    }
```

The second set of constants (lines 177–187) is a bit more obscure. The `INCLUDE_NONE`, `INCLUDE_FIRST`, `INCLUDE_SECOND`, and `INCLUDE_BOTH` constants are used to describe whether the defining end-point dates of a range should be included in that range. Mathematically, this is described using the terms "open interval," "half-open interval," and "closed interval." I think it is clearer using the mathematical nomenclature [N3], so I changed it to an enum named `DateInterval` with `CLOSED`, `CLOSED_LEFT`, `CLOSED_RIGHT`, and `OPEN` enumerators.

The third set of constants (lines 18–205) describe whether a search for a particular day of the week should result in the last, next, or nearest instance. Deciding what to call this is difficult at best. In the end, I settled for `WeekdayRange` with `LAST`, `NEXT`, and `NEAREST` enumerators.

You might not agree with the names I've chosen. They make sense to me, but they may not make sense to you. The point is that they are now in a form that makes them easy to change [J3]. They aren't passed as integers anymore; they are passed as symbols. I can use the "change name" function of my IDE to change the names, or the types, without worrying that I missed some `-1` or `2` somewhere in the code or that some `int` argument declaration is left poorly described.

The description field at line 208 does not seem to be used by anyone. I deleted it along with its accessor and mutator [G9].

I also deleted the degenerate default constructor at line 213 [G12]. The compiler will generate it for us.

We can skip over the `isValidWeekdayCode` method (lines 216–238) because we deleted it when we created the `Day` enumeration.

This brings us to the `stringToWeekdayCode` method (lines 242–270). Javadocs that don't add much to the method signature are just clutter [C3],[G12]. The only value this Javadoc adds is the description of the `-1` return value. However, because we changed to the `Day` enumeration, the comment is actually wrong [C2]. The method now throws an `IllegalArgumentException`. So I deleted the Javadoc.

I also deleted all the `final` keywords in arguments and variable declarations. As far as I could tell, they added no real value but did add to the clutter [G12]. Eliminating `final` flies in the face of some conventional wisdom. For example, Robert Simmons[6] strongly recommends us to ". . . spread `final` all over your code." Clearly I disagree. I think that there are a few good uses for `final`, such as the occasional `final` constant, but otherwise the keyword adds little value and creates a lot of clutter. Perhaps I feel this way because the kinds of errors that `final` might catch are already caught by the unit tests I write.

I didn't care for the duplicate `if` statements [G5] inside the `for` loop (line 259 and line 263), so I connected them into a single `if` statement using the `||` operator. I also used the `Day` enumeration to direct the `for` loop and made a few other cosmetic changes.

It occurred to me that this method does not really belong in `DayDate`. It's really the parse function of `Day`. So I moved it into the `Day` enumeration. However, that made the `Day`

6. [Simmons04], p. 73.

enumeration pretty large. Because the concept of `Day` does not depend on `DayDate`, I moved the `Day` enumeration outside of the `DayDate` class into its own source file [G13].

I also moved the next function, `weekdayCodeToString` (lines 272–286) into the `Day` enumeration and called it `toString`.

```
public enum Day {
  MONDAY(Calendar.MONDAY),
  TUESDAY(Calendar.TUESDAY),
  WEDNESDAY(Calendar.WEDNESDAY),s
  THURSDAY(Calendar.THURSDAY),
  FRIDAY(Calendar.FRIDAY),
  SATURDAY(Calendar.SATURDAY),
  SUNDAY(Calendar.SUNDAY);

  public final int index;
  private static DateFormatSymbols dateSymbols = new DateFormatSymbols();

  Day(int day) {
    index = day;
  }

  public static Day make(int index) throws IllegalArgumentException {
    for (Day d : Day.values())
      if (d.index == index)
        return d;
    throw new IllegalArgumentException(
      String.format("Illegal day index: %d.", index));
  }

  public static Day parse(String s) throws IllegalArgumentException {
    String[] shortWeekdayNames =
      dateSymbols.getShortWeekdays();
    String[] weekDayNames =
      dateSymbols.getWeekdays();

    s = s.trim();
    for (Day day : Day.values()) {
      if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) ||
          s.equalsIgnoreCase(weekDayNames[day.index])) {
        return day;
      }
    }
    throw new IllegalArgumentException(
      String.format("%s is not a valid weekday string", s));
  }

  public String toString() {
    return dateSymbols.getWeekdays()[index];
  }
}
```

There are two `getMonths` functions (lines 288–316). The first calls the second. The second is never called by anyone but the first. Therefore, I collapsed the two into one and vastly simplified them [G9],[G12],[F4]. Finally, I changed the name to be a bit more self-descriptive [N1].

```
  public static String[] getMonthNames() {
    return dateFormatSymbols.getMonths();
  }
```

The `isValidMonthCode` function (lines 326–346) was made irrelevant by the `Month` enum, so I deleted it [G9].

The `monthCodeToQuarter` function (lines 356–375) smells of FEATURE ENVY[7] [G14] and probably belongs in the `Month` enum as a method named `quarter`. So I replaced it.

```
  public int quarter() {
    return 1 + (index-1)/3;
  }
```

This made the `Month` enum big enough to be in its own class. So I moved it out of `DayDate` to be consistent with the `Day` enum [G11],[G13].

The next two methods are named `monthCodeToString` (lines 377–426). Again, we see the pattern of one method calling its twin with a flag. It is usually a bad idea to pass a flag as an argument to a function, especially when that flag simply selects the format of the output [G15]. I renamed, simplified, and restructured these functions and moved them into the `Month` enum [N1],[N3],[C3],[G14].

```
  public String toString() {
    return dateFormatSymbols.getMonths()[index - 1];
  }

  public String toShortString() {
    return dateFormatSymbols.getShortMonths()[index - 1];
  }
```

The next method is `stringToMonthCode` (lines 428–472). I renamed it, moved it into the `Month` enum, and simplified it [N1],[N3],[C3],[G14],[G12].

```
  public static Month parse(String s) {
    s = s.trim();
    for (Month m : Month.values())
      if (m.matches(s))
        return m;

    try {
      return make(Integer.parseInt(s));
    }
    catch (NumberFormatException e) {}
    throw new IllegalArgumentException("Invalid month " + s);
  }
```

7. [Refactoring].

```
  private boolean matches(String s) {
    return s.equalsIgnoreCase(toString()) ||
           s.equalsIgnoreCase(toShortString());
  }
```

The `isLeapYear` method (lines 495–517) can be made a bit more expressive [G16].

```
  public static boolean isLeapYear(int year) {
    boolean fourth = year % 4 == 0;
    boolean hundredth = year % 100 == 0;
    boolean fourHundredth = year % 400 == 0;
    return fourth && (!hundredth || fourHundredth);
  }
```

The next function, `leapYearCount` (lines 519–536) doesn't really belong in `DayDate`. Nobody calls it except for two methods in `SpreadsheetDate`. So I pushed it down [G6].

The `lastDayOfMonth` function (lines 538–560) makes use of the `LAST_DAY_OF_MONTH` array. This array really belongs in the `Month` enum [G17], so I moved it there. I also simplified the function and made it a bit more expressive [G16].

```
  public static int lastDayOfMonth(Month month, int year) {
    if (month == Month.FEBRUARY && isLeapYear(year))
      return month.lastDay() + 1;
     else
      return month.lastDay();
  }
```

Now things start to get a bit more interesting. The next function is `addDays` (lines 562–576). First of all, because this function operates on the variables of `DayDate`, it should not be static [G18]. So I changed it to an instance method. Second, it calls the function `toSerial`. This function should be renamed `toOrdinal` [N1]. Finally, the method can be simplified.

```
  public DayDate addDays(int days) {
    return DayDateFactory.makeDate(toOrdinal() + days);
  }
```

The same goes for `addMonths` (lines 578–602). It should be an instance method [G18]. The algorithm is a bit complicated, so I used EXPLAINING TEMPORARY VARIABLES[8] [G19] to make it more transparent. I also renamed the method `getYYY` to `getYear` [N1].

```
  public DayDate addMonths(int months) {
    int thisMonthAsOrdinal = 12 * getYear() + getMonth().index - 1;
    int resultMonthAsOrdinal = thisMonthAsOrdinal + months;
    int resultYear = resultMonthAsOrdinal / 12;
    Month resultMonth = Month.make(resultMonthAsOrdinal % 12 + 1);
```

8. [Beck97].

```
    int lastDayOfResultMonth = lastDayOfMonth(resultMonth, resultYear);
    int resultDay = Math.min(getDayOfMonth(), lastDayOfResultMonth);
    return DayDateFactory.makeDate(resultDay, resultMonth, resultYear);
  }
```

The `addYears` function (lines 604–626) provides no surprises over the others.

```
  public DayDate plusYears(int years) {
    int resultYear = getYear() + years;
    int lastDayOfMonthInResultYear = lastDayOfMonth(getMonth(), resultYear);
    int resultDay = Math.min(getDayOfMonth(), lastDayOfMonthInResultYear);
    return DayDateFactory.makeDate(resultDay, getMonth(), resultYear);
  }
```

There is a little itch at the back of my mind that is bothering me about changing these methods from static to instance. Does the expression `date.addDays(5)` make it clear that the `date` object does not change and that a new instance of `DayDate` is returned? Or does it erroneously imply that we are adding five days to the `date` object? You might not think that is a big problem, but a bit of code that looks like the following can be very deceiving [G20].

```
DayDate date = DateFactory.makeDate(5, Month.DECEMBER, 1952);
date.addDays(7); // bump date by one week.
```

Someone reading this code would very likely just accept that `addDays` is changing the `date` object. So we need a name that breaks this ambiguity [N4]. So I changed the names to `plusDays` and `plusMonths`. It seems to me that the intent of the method is captured nicely by

```
DayDate date = oldDate.plusDays(5);
```

whereas the following doesn't read fluidly enough for a reader to simply accept that the `date` object is changed:

```
date.plusDays(5);
```

The algorithms continue to get more interesting. `getPreviousDayOfWeek` (lines 628–660) works but is a bit complicated. After some thought about what was really going on [G21], I was able to simplify it and use EXPLAINING TEMPORARY VARIABLES [G19] to make it clearer. I also changed it from a static method to an instance method [G18], and got rid of the duplicate instance method [G5] (lines 997–1008).

```
  public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {
    int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().index;
    if (offsetToTarget >= 0)
      offsetToTarget -= 7;
    return plusDays(offsetToTarget);
  }
```

The exact same analysis and result occurred for `getFollowingDayOfWeek` (lines 662–693).

```
public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {
    int offsetToTarget = targetDayOfWeek.index - getDayOfWeek().index;
    if (offsetToTarget <= 0)
```

```
        offsetToTarget += 7;
      return plusDays(offsetToTarget);
    }
```

The next function is `getNearestDayOfWeek` (lines 695–726), which we corrected back on page 270. But the changes I made back then aren't consistent with the current pattern in the last two functions [G11]. So I made it consistent and used some EXPLAINING TEMPORARY VARIABLES [G19] to clarify the algorithm.

```
  public DayDate getNearestDayOfWeek(final Day targetDay) {
      int offsetToThisWeeksTarget = targetDay.index - getDayOfWeek().index;
      int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;
      int offsetToPreviousTarget = offsetToFutureTarget - 7;

      if (offsetToFutureTarget > 3)
        return plusDays(offsetToPreviousTarget);
      else
        return plusDays(offsetToFutureTarget);
    }
```

The `getEndOfCurrentMonth` method (lines 728–740) is a little strange because it is an instance method that envies [G14] its own class by taking a `DayDate` argument. I made it a true instance method and clarified a few names.

```
  public DayDate getEndOfMonth() {
      Month month = getMonth();
      int year = getYear();
      int lastDay = lastDayOfMonth(month, year);
      return DayDateFactory.makeDate(lastDay, month, year);
    }
```

Refactoring `weekInMonthToString` (lines 742–761) turned out to be very interesting indeed. Using the refactoring tools of my IDE, I first moved the method to the `WeekInMonth` enum that I created back on page 275. Then I renamed the method to `toString`. Next, I changed it from a static method to an instance method. All the tests still passed. (Can you guess where I am going?)

Next, I deleted the method entirely! Five asserts failed (lines 411–415, Listing B-4, page 374). I changed these lines to use the names of the enumerators (`FIRST`, `SECOND`, . . .). All the tests passed. Can you see why? Can you also see why each of these steps was necessary? The refactoring tool made sure that all previous callers of `weekInMonthToString` now called `toString` on the `weekInMonth` enumerator because all enumerators implement `toString` to simply return their names. . . .

Unfortunately, I was a bit too clever. As elegant as that wonderful chain of refactorings was, I finally realized that the only users of this function were the tests I had just modified, so I deleted the tests.

Fool me once, shame on you. Fool me twice, shame on me! So after determining that nobody other than the tests called `relativeToString` (lines 765–781), I simply deleted the function and its tests.

We have finally made it to the abstract methods of this abstract class. And the first one is as appropriate as they come: `toSerial` (lines 838–844). Back on page 279 I had changed the name to `toOrdinal`. Having looked at it in this context, I decided the name should be changed to `getOrdinalDay`.

The next abstract method is `toDate` (lines 838–844). It converts a `DayDate` to a `java.util.Date`. Why is this method abstract? If we look at its implementation in `SpreadsheetDate` (lines 198–207, Listing B-5, page 382), we see that it doesn't depend on anything in the implementation of that class [G6]. So I pushed it up.

The `getYYYY`, `getMonth`, and `getDayOfMonth` methods are nicely abstract. However, the `getDayOfWeek` method is another one that should be pulled up from `SpreadSheetDate` because it doesn't depend on anything that can't be found in `DayDate` [G6]. Or does it?

If you look carefully (line 247, Listing B-5, page 382), you'll see that the algorithm implicitly depends on the origin of the ordinal day (in other words, the day of the week of day 0). So even though this function has no physical dependencies that couldn't be moved to `DayDate`, it does have a logical dependency.

Logical dependencies like this bother me [G22]. If something logical depends on the implementation, then something physical should too. Also, it seems to me that the algorithm itself could be generic with a much smaller portion of it dependent on the implementation [G6].

So I created an abstract method in `DayDate` named `getDayOfWeekForOrdinalZero` and implemented it in `SpreadsheetDate` to return `Day.SATURDAY`. Then I moved the `getDayOfWeek` method up to `DayDate` and changed it to call `getOrdinalDay` and `getDayOfWeekForOrdinal-Zero`.

```
  public Day getDayOfWeek() {
      Day startingDay = getDayOfWeekForOrdinalZero();
      int startingOffset = startingDay.index - Day.SUNDAY.index;
      return Day.make((getOrdinalDay() + startingOffset) % 7 + 1);
    }
```

As a side note, look carefully at the comment on line 895 through line 899. Was this repetition really necessary? As usual, I deleted this comment along with all the others.

The next method is `compare` (lines 902–913). Again, this method is inappropriately abstract [G6], so I pulled the implementation up into `DayDate`. Also, the name does not communicate enough [N1]. This method actually returns the difference in days since the argument. So I changed the name to `daysSince`. Also, I noted that there weren't any tests for this method, so I wrote them.

The next six functions (lines 915–980) are all abstract methods that should be implemented in `DayDate`. So I pulled them all up from `SpreadsheetDate`.

The last function, `isInRange` (lines 982–995) also needs to be pulled up and refactored. The `switch` statement is a bit ugly [G23] and can be replaced by moving the cases into the `DateInterval` enum.

```
    public enum DateInterval {
        OPEN {
          public boolean isIn(int d, int left, int right) {
            return d > left && d < right;
          }
        },
        CLOSED_LEFT {
          public boolean isIn(int d, int left, int right) {
            return d >= left && d < right;
          }
        },
        CLOSED_RIGHT {
          public boolean isIn(int d, int left, int right) {
            return d > left && d <= right;
          }
        },
        CLOSED {
          public boolean isIn(int d, int left, int right) {
            return d >= left && d <= right;
          }
        };

        public abstract boolean isIn(int d, int left, int right);
      }
```

```
    public boolean isInRange(DayDate d1, DayDate d2, DateInterval interval) {
        int left = Math.min(d1.getOrdinalDay(), d2.getOrdinalDay());
        int right = Math.max(d1.getOrdinalDay(), d2.getOrdinalDay());
        return interval.isIn(getOrdinalDay(), left, right);
      }
```

That brings us to the end of `DayDate`. So now we'll make one more pass over the whole class to see how well it flows.

First, the opening comment is long out of date, so I shortened and improved it [C2].

Next, I moved all the remaining enums out into their own files [G12].

Next, I moved the static variable (`dateFormatSymbols`) and three static methods (`getMonthNames`, `isLeapYear`, `lastDayOfMonth`) into a new class named `DateUtil` [G6].

I moved the abstract methods up to the top where they belong [G24].

I changed `Month.make` to `Month.fromInt` [N1] and did the same for all the other enums. I also created a `toInt()` accessor for all the enums and made the `index` field private.

There was some interesting duplication [G5] in `plusYears` and `plusMonths` that I was able to eliminate by extracting a new method named `correctLastDayOfMonth`, making the all three methods much clearer.

I got rid of the magic number 1 [G25], replacing it with `Month.JANUARY.toInt()` or `Day.SUNDAY.toInt()`, as appropriate. I spent a little time with `SpreadsheetDate`, cleaning up the algorithms a bit. The end result is contained in Listing B-7, page 394, through Listing B-16, page 405.

Interestingly the code coverage in DayDate has *decreased* to 84.9 percent! This is not because less functionality is being tested; rather it is because the class has shrunk so much that the few uncovered lines have a greater weight. DayDate now has 45 out of 53 executable statements covered by tests. The uncovered lines are so trivial that they weren't worth testing.

Conclusion

So once again we've followed the Boy Scout Rule. We've checked the code in a bit cleaner than when we checked it out. It took a little time, but it was worth it. Test coverage was increased, some bugs were fixed, the code was clarified and shrunk. The next person to look at this code will hopefully find it easier to deal with than we did. That person will also probably be able to clean it up a bit more than we did.

Bibliography

[GOF]: *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

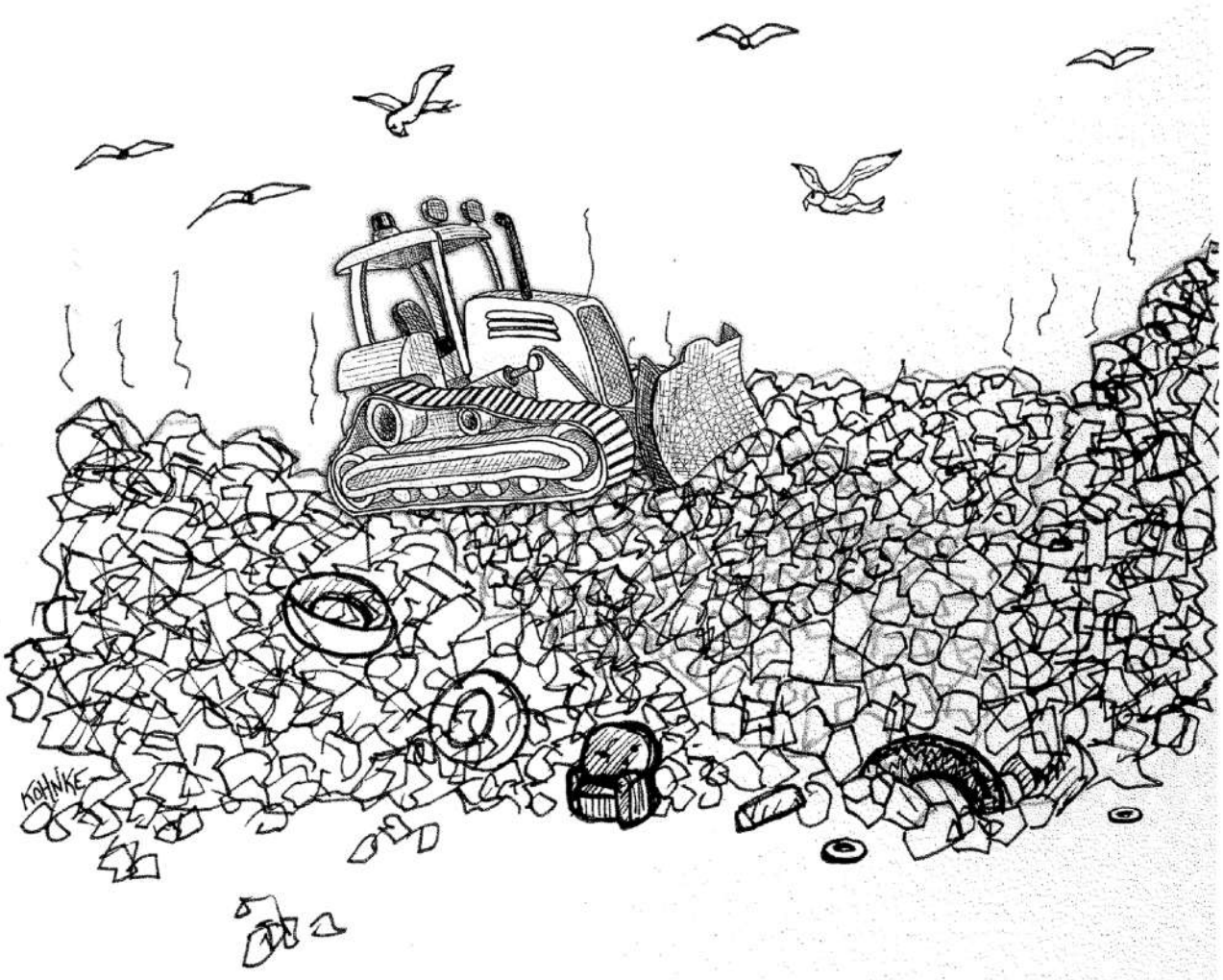
[Simmons04]: *Hardcore Java*, Robert Simmons, Jr., O'Reilly, 2004.

[Refactoring]: *Refactoring: Improving the Design of Existing Code*, Martin Fowler et al., Addison-Wesley, 1999.

[Beck97]: *Smalltalk Best Practice Patterns*, Kent Beck, Prentice Hall, 1997.

17

Smells and Heuristics



In his wonderful book *Refactoring*,[1] Martin Fowler identified many different "Code Smells." The list that follows includes many of Martin's smells and adds many more of my own. It also includes other pearls and heuristics that I use to practice my trade.

I compiled this list by walking through several different programs and refactoring them. As I made each change, I asked myself *why* I made that change and then wrote the reason down here. The result is a rather long list of things that smell bad to me when I read code.

This list is meant to be read from top to bottom and also to be used as a reference. There is a cross-reference for each heuristic that shows you where it is referenced in the rest of the text in “Appendix C” on page 409.

Comments

C1: *Inappropriate Information*

It is inappropriate for a comment to hold information better held in a different kind of system such as your source code control system, your issue tracking system, or any other record-keeping system. Change histories, for example, just clutter up source files with volumes of historical and uninteresting text. In general, meta-data such as authors, last-modified-date, SPR number, and so on should not appear in comments. Comments should be reserved for technical notes about the code and design.

C2: *Obsolete Comment*

A comment that has gotten old, irrelevant, and incorrect is obsolete. Comments get old quickly. It is best not to write a comment that will become obsolete. If you find an obsolete comment, it is best to update it or get rid of it as quickly as possible. Obsolete comments tend to migrate away from the code they once described. They become floating islands of irrelevance and misdirection in the code.

C3: *Redundant Comment*

A comment is redundant if it describes something that adequately describes itself. For example:

```
i++; // increment i
```

Another example is a Javadoc that says nothing more than (or even less than) the function signature:

```
/**
 * @param sellRequest
 * @return
 * @throws ManagedComponentException
 */
public SellResponse beginSellItem(SellRequest sellRequest)
  throws ManagedComponentException
```

Comments should say things that the code cannot say for itself.

C4: *Poorly Written Comment*

A comment worth writing is worth writing well. If you are going to write a comment, take the time to make sure it is the best comment you can write. Choose your words carefully. Use correct grammar and punctuation. Don't ramble. Don't state the obvious. Be brief.

C5: *Commented-Out Code*

It makes me crazy to see stretches of code that are commented out. Who knows how old it is? Who knows whether or not it's meaningful? Yet no one will delete it because everyone assumes someone else needs it or has plans for it.

That code sits there and rots, getting less and less relevant with every passing day. It calls functions that no longer exist. It uses variables whose names have changed. It follows conventions that are long obsolete. It pollutes the modules that contain it and distracts the people who try to read it. Commented-out code is an *abomination*.

When you see commented-out code, *delete it!* Don't worry, the source code control system still remembers it. If anyone really needs it, he or she can go back and check out a previous version. Don't suffer commented-out code to survive.

Environment

E1: *Build Requires More Than One Step*

Building a project should be a single trivial operation. You should not have to check many little pieces out from source code control. You should not need a sequence of arcane commands or context dependent scripts in order to build the individual elements. You should not have to search near and far for all the various little extra JARs, XML files, and other artifacts that the system requires. You *should* be able to check out the system with one simple command and then issue one other simple command to build it.

```
svn get mySystem
cd mySystem
ant all
```

E2: *Tests Require More Than One Step*

You should be able to run *all* the unit tests with just one command. In the best case you can run all the tests by clicking on one button in your IDE. In the worst case you should be able to issue a single simple command in a shell. Being able to run all the tests is so fundamental and so important that it should be quick, easy, and obvious to do.

Functions

F1: *Too Many Arguments*

Functions should have a small number of arguments. No argument is best, followed by one, two, and three. More than three is very questionable and should be avoided with prejudice. (See “Function Arguments” on page 40.)

F2: *Output Arguments*

Output arguments are counterintuitive. Readers expect arguments to be inputs, not outputs. If your function must change the state of something, have it change the state of the object it is called on. (See “Output Arguments” on page 45.)

F3: *Flag Arguments*

Boolean arguments loudly declare that the function does more than one thing. They are confusing and should be eliminated. (See “Flag Arguments” on page 41.)

F4: *Dead Function*

Methods that are never called should be discarded. Keeping dead code around is wasteful. Don’t be afraid to delete the function. Remember, your source code control system still remembers it.

General

G1: *Multiple Languages in One Source File*

Today’s modern programming environments make it possible to put many different languages into a single source file. For example, a Java source file might contain snippets of XML, HTML, YAML, JavaDoc, English, JavaScript, and so on. For another example, in addition to HTML a JSP file might contain Java, a tag library syntax, English comments, Javadocs, XML, JavaScript, and so forth. This is confusing at best and carelessly sloppy at worst.

The ideal is for a source file to contain one, and only one, language. Realistically, we will probably have to use more than one. But we should take pains to minimize both the number and extent of extra languages in our source files.

G2: *Obvious Behavior Is Unimplemented*

Following “The Principle of Least Surprise,”[2] any function or class should implement the behaviors that another programmer could reasonably expect. For example, consider a function that translates the name of a day to an `enum` that represents the day.

---

2. Or “The Principle of Least Astonishment”: `http://en.wikipedia.org/wiki/Principle_of_least_astonishment`

```
Day day = DayDate.StringToDay(String dayName);
```

We would expect the string `"Monday"` to be translated to `Day.MONDAY`. We would also expect the common abbreviations to be translated, and we would expect the function to ignore case.

When an obvious behavior is not implemented, readers and users of the code can no longer depend on their intuition about function names. They lose their trust in the original author and must fall back on reading the details of the code.

G3: *Incorrect Behavior at the Boundaries*

It seems obvious to say that code should behave correctly. The problem is that we seldom realize just how complicated correct behavior is. Developers often write functions that they think will work, and then trust their intuition rather than going to the effort to prove that their code works in all the corner and boundary cases.

There is no replacement for due diligence. Every boundary condition, every corner case, every quirk and exception represents something that can confound an elegant and intuitive algorithm. *Don't rely on your intuition*. Look for every boundary condition and write a test for it.

G4: *Overridden Safeties*

Chernobyl melted down because the plant manager overrode each of the safety mechanisms one by one. The safeties were making it inconvenient to run an experiment. The result was that the experiment did not get run, and the world saw it's first major civilian nuclear catastrophe.

It is risky to override safeties. Exerting manual control over `serialVersionUID` may be necessary, but it is always risky. Turning off certain compiler warnings (or all warnings!) may help you get the build to succeed, but at the risk of endless debugging sessions. Turning off failing tests and telling yourself you'll get them to pass later is as bad as pretending your credit cards are free money.

G5: *Duplication*

This is one of the most important rules in this book, and you should take it very seriously. Virtually every author who writes about software design mentions this rule. Dave Thomas and Andy Hunt called it the DRY[3] principle (Don't Repeat Yourself). Kent Beck made it one of the core principles of Extreme Programming and called it: "Once, and only once." Ron Jeffries ranks this rule second, just below getting all the tests to pass.

Every time you see duplication in the code, it represents a missed opportunity for abstraction. That duplication could probably become a subroutine or perhaps another class outright. By folding the duplication into such an abstraction, you increase the vocabulary of the language of your design. Other programmers can use the abstract facilities

3. [PRAG].

you create. Coding becomes faster and less error prone because you have raised the abstraction level.

The most obvious form of duplication is when you have clumps of identical code that look like some programmers went wild with the mouse, pasting the same code over and over again. These should be replaced with simple methods.

A more subtle form is the `switch/case` or `if/else` chain that appears again and again in various modules, always testing for the same set of conditions. These should be replaced with polymorphism.

Still more subtle are the modules that have similar algorithms, but that don't share similar lines of code. This is still duplication and should be addressed by using the TEMPLATE METHOD,[4] or STRATEGY[5] pattern.

Indeed, most of the design patterns that have appeared in the last fifteen years are simply well-known ways to eliminate duplication. So too the Codd Normal Forms are a strategy for eliminating duplication in database schemae. OO itself is a strategy for organizing modules and eliminating duplication. Not surprisingly, so is structured programming.

I think the point has been made. Find and eliminate duplication wherever you can.

G6: *Code at Wrong Level of Abstraction*

It is important to create abstractions that separate higher level general concepts from lower level detailed concepts. Sometimes we do this by creating abstract classes to hold the higher level concepts and derivatives to hold the lower level concepts. When we do this, we need to make sure that the separation is complete. We want *all* the lower level concepts to be in the derivatives and *all* the higher level concepts to be in the base class.

For example, constants, variables, or utility functions that pertain only to the detailed implementation should not be present in the base class. The base class should know nothing about them.

This rule also pertains to source files, components, and modules. Good software design requires that we separate concepts at different levels and place them in different containers. Sometimes these containers are base classes or derivatives and sometimes they are source files, modules, or components. Whatever the case may be, the separation needs to be complete. We don't want lower and higher level concepts mixed together.

Consider the following code:

```
public interface Stack {
  Object pop() throws EmptyException;
  void push(Object o) throws FullException;
  double percentFull();
```

4. [GOF].
5. [GOF].

```
  class EmptyException extends Exception {}
  class FullException extends Exception {}
}
```

The `percentFull` function is at the wrong level of abstraction. Although there are many implementations of `Stack` where the concept of *fullness* is reasonable, there are other implementations that simply *could not know* how full they are. So the function would be better placed in a derivative interface such as `BoundedStack`.

Perhaps you are thinking that the implementation could just return zero if the stack were boundless. The problem with that is that no stack is truly boundless. You cannot really prevent an `OutOfMemoryException` by checking for

```
stack.percentFull() < 50.0.
```

Implementing the function to return 0 would be telling a lie.

The point is that you cannot lie or fake your way out of a misplaced abstraction. Isolating abstractions is one of the hardest things that software developers do, and there is no quick fix when you get it wrong.

G7: *Base Classes Depending on Their Derivatives*

The most common reason for partitioning concepts into base and derivative classes is so that the higher level base class concepts can be independent of the lower level derivative class concepts. Therefore, when we see base classes mentioning the names of their derivatives, we suspect a problem. In general, base classes should know nothing about their derivatives.

There are exceptions to this rule, of course. Sometimes the number of derivatives is strictly fixed, and the base class has code that selects between the derivatives. We see this a lot in finite state machine implementations. However, in that case the derivatives and base class are strongly coupled and always deploy together in the same jar file. In the general case we want to be able to deploy derivatives and bases in different jar files.

Deploying derivatives and bases in different jar files and making sure the base jar files know nothing about the contents of the derivative jar files allow us to deploy our systems in discrete and independent components. When such components are modified, they can be redeployed without having to redeploy the base components. This means that the impact of a change is greatly lessened, and maintaining systems in the field is made much simpler.

G8: *Too Much Information*

Well-defined modules have very small interfaces that allow you to do a lot with a little. Poorly defined modules have wide and deep interfaces that force you to use many different gestures to get simple things done. A well-defined interface does not offer very many functions to depend upon, so coupling is low. A poorly defined interface provides lots of functions that you must call, so coupling is high.

Good software developers learn to limit what they expose at the interfaces of their classes and modules. The fewer methods a class has, the better. The fewer variables a function knows about, the better. The fewer instance variables a class has, the better.

Hide your data. Hide your utility functions. Hide your constants and your temporaries. Don't create classes with lots of methods or lots of instance variables. Don't create lots of protected variables and functions for your subclasses. Concentrate on keeping interfaces very tight and very small. Help keep coupling low by limiting information.

G9: *Dead Code*

Dead code is code that isn't executed. You find it in the body of an `if` statement that checks for a condition that can't happen. You find it in the `catch` block of a `try` that never `throws`. You find it in little utility methods that are never called or `switch/case` conditions that never occur.

The problem with dead code is that after awhile it starts to smell. The older it is, the stronger and sourer the odor becomes. This is because dead code is not completely updated when designs change. It still *compiles*, but it does not follow newer conventions or rules. It was written at a time when the system was *different*. When you find dead code, do the right thing. Give it a decent burial. Delete it from the system.

G10: *Vertical Separation*

Variables and function should be defined close to where they are used. Local variables should be declared just above their first usage and should have a small vertical scope. We don't want local variables declared hundreds of lines distant from their usages.

Private functions should be defined just below their first usage. Private functions belong to the scope of the whole class, but we'd still like to limit the vertical distance between the invocations and definitions. Finding a private function should just be a matter of scanning downward from the first usage.

G11: *Inconsistency*

If you do something a certain way, do all similar things in the same way. This goes back to the principle of least surprise. Be careful with the conventions you choose, and once chosen, be careful to continue to follow them.

If within a particular function you use a variable named `response` to hold an `HttpServletResponse`, then use the same variable name consistently in the other functions that use `HttpServletResponse` objects. If you name a method `processVerificationRequest`, then use a similar name, such as `processDeletionRequest`, for the methods that process other kinds of requests.

Simple consistency like this, when reliably applied, can make code much easier to read and modify.

G12: *Clutter*

Of what use is a default constructor with no implementation? All it serves to do is clutter up the code with meaningless artifacts. Variables that aren't used, functions that are never called, comments that add no information, and so forth. All these things are clutter and should be removed. Keep your source files clean, well organized, and free of clutter.

G13: *Artificial Coupling*

Things that don't depend upon each other should not be artificially coupled. For example, general `enums` should not be contained within more specific classes because this forces the whole application to know about these more specific classes. The same goes for general purpose `static` functions being declared in specific classes.

In general an artificial coupling is a coupling between two modules that serves no direct purpose. It is a result of putting a variable, constant, or function in a temporarily convenient, though inappropriate, location. This is lazy and careless.

Take the time to figure out where functions, constants, and variables ought to be declared. Don't just toss them in the most convenient place at hand and then leave them there.

G14: *Feature Envy*

This is one of Martin Fowler's code smells.[6] The methods of a class should be interested in the variables and functions of the class they belong to, and not the variables and functions of other classes. When a method uses accessors and mutators of some other object to manipulate the data within that object, then it *envies* the scope of the class of that other object. It wishes that it were inside that other class so that it could have direct access to the variables it is manipulating. For example:

```
public class HourlyPayCalculator {
  public Money calculateWeeklyPay(HourlyEmployee e) {
    int tenthRate = e.getTenthRate().getPennies();
    int tenthsWorked = e.getTenthsWorked();
    int straightTime = Math.min(400, tenthsWorked);
    int overTime = Math.max(0, tenthsWorked - straightTime);
    int straightPay = straightTime * tenthRate;
    int overtimePay = (int)Math.round(overTime*tenthRate*1.5);
    return new Money(straightPay + overtimePay);
  }
}
```

The `calculateWeeklyPay` method reaches into the `HourlyEmployee` object to get the data on which it operates. The `calculateWeeklyPay` method *envies* the scope of `HourlyEmployee`. It "wishes" that it could be inside `HourlyEmployee`.

---

6. [Refactoring].

All else being equal, we want to eliminate Feature Envy because it exposes the internals of one class to another. Sometimes, however, Feature Envy is a necessary evil. Consider the following:

```
public class HourlyEmployeeReport {
  private HourlyEmployee employee ;

  public HourlyEmployeeReport(HourlyEmployee e) {
    this.employee = e;
  }

  String reportHours() {
    return String.format(
      "Name: %s\tHours:%d.%1d\n",
      employee.getName(),
      employee.getTenthsWorked()/10,
      employee.getTenthsWorked()%10);
  }
}
```

Clearly, the `reportHours` method envies the `HourlyEmployee` class. On the other hand, we don't want `HourlyEmployee` to have to know about the format of the report. Moving that format string into the `HourlyEmployee` class would violate several principles of object oriented design.[7] It would couple `HourlyEmployee` to the format of the report, exposing it to changes in that format.

G15: *Selector Arguments*

There is hardly anything more abominable than a dangling `false` argument at the end of a function call. What does it mean? What would it change if it were `true`? Not only is the purpose of a selector argument difficult to remember, each selector argument combines many functions into one. Selector arguments are just a lazy way to avoid splitting a large function into several smaller functions. Consider:

```
public int calculateWeeklyPay(boolean overtime) {
  int tenthRate = getTenthRate();
  int tenthsWorked = getTenthsWorked();
  int straightTime = Math.min(400, tenthsWorked);
  int overTime = Math.max(0, tenthsWorked - straightTime);
  int straightPay = straightTime * tenthRate;
  double overtimeRate = overtime ? 1.5 : 1.0 * tenthRate;
  int overtimePay = (int)Math.round(overTime*overtimeRate);
  return straightPay + overtimePay;
}
```

You call this function with a `true` if overtime is paid as time and a half, and with a `false` if overtime is paid as straight time. It's bad enough that you must remember what `calculateWeeklyPay(false)` means whenever you happen to stumble across it. But the

7. Specifically, the Single Responsibility Principle, the Open Closed Principle, and the Common Closure Principle. See [PPP].

real shame of a function like this is that the author missed the opportunity to write the following:

```
public int straightPay() {
  return getTenthsWorked() * getTenthRate();
}

public int overTimePay() {
  int overTimeTenths = Math.max(0, getTenthsWorked() - 400);
  int overTimePay = overTimeBonus(overTimeTenths);
  return straightPay() + overTimePay;
}

private int overTimeBonus(int overTimeTenths) {
  double bonus = 0.5 * getTenthRate() * overTimeTenths;
  return (int) Math.round(bonus);
}
```

Of course, selectors need not be `boolean`. They can be enums, integers, or any other type of argument that is used to select the behavior of the function. In general it is better to have many functions than to pass some code into a function to select the behavior.

G16: *Obscured Intent*

We want code to be as expressive as possible. Run-on expressions, Hungarian notation, and magic numbers all obscure the author's intent. For example, here is the `overTimePay` function as it might have appeared:

```
public int m_otCalc() {
  return iThsWkd * iThsRte +
    (int) Math.round(0.5 * iThsRte *
      Math.max(0, iThsWkd - 400)
    );
}
```

Small and dense as this might appear, it's also virtually impenetrable. It is worth taking the time to make the intent of our code visible to our readers.

G17: *Misplaced Responsibility*

One of the most important decisions a software developer can make is where to put code. For example, where should the `PI` constant go? Should it be in the `Math` class? Perhaps it belongs in the `Trigonometry` class? Or maybe in the `Circle` class?

The principle of least surprise comes into play here. Code should be placed where a reader would naturally expect it to be. The `PI` constant should go where the trig functions are declared. The OVERTIME_RATE constant should be declared in the `HourlyPayCalculator` class.

Sometimes we get "clever" about where to put certain functionality. We'll put it in a function that's convenient for us, but not necessarily intuitive to the reader. For example, perhaps we need to print a report with the total of hours that an employee worked. We

could sum up those hours in the code that prints the report, or we could try to keep a running total in the code that accepts time cards.

One way to make this decision is to look at the names of the functions. Let's say that our report module has a function named `getTotalHours`. Let's also say that the module that accepts time cards has a `saveTimeCard` function. Which of these two functions, by it's name, implies that it calculates the total? The answer should be obvious.

Clearly, there are sometimes performance reasons why the total should be calculated as time cards are accepted rather than when the report is printed. That's fine, but the names of the functions ought to reflect this. For example, there should be a `computeRunningTotalOfHours` function in the timecard module.

G18: *Inappropriate Static*

`Math.max(double a, double b)` is a good static method. It does not operate on a single instance; indeed, it would be silly to have to say `new Math().max(a,b)` or even `a.max(b)`. All the data that `max` uses comes from its two arguments, and not from any "owning" object. More to the point, there is almost *no chance* that we'd want `Math.max` to be polymorphic.

Sometimes, however, we write static functions that should not be static. For example, consider:

```
HourlyPayCalculator.calculatePay(employee, overtimeRate).
```

Again, this seems like a reasonable `static` function. It doesn't operate on any particular object and gets all it's data from it's arguments. However, there is a reasonable chance that we'll want this function to be polymorphic. We may wish to implement several different algorithms for calculating hourly pay, for example, `OvertimeHourlyPayCalculator` and `StraightTimeHourlyPayCalculator`. So in this case the function should not be static. It should be a nonstatic member function of `Employee`.

In general you should prefer nonstatic methods to static methods. When in doubt, make the function nonstatic. If you really want a function to be static, make sure that there is no chance that you'll want it to behave polymorphically.

G19: *Use Explanatory Variables*

Kent Beck wrote about this in his great book *Smalltalk Best Practice Patterns*[8] and again more recently in his equally great book *Implementation Patterns*.[9] One of the more powerful ways to make a program readable is to break the calculations up into intermediate values that are held in variables with meaningful names.

---

8. [Beck97], p. 108.
9. [Beck07].

Consider this example from FitNesse:

```
Matcher match = headerPattern.matcher(line);
if(match.find())
{
  String key = match.group(1);
  String value = match.group(2);
  headers.put(key.toLowerCase(), value);
}
```

The simple use of explanatory variables makes it clear that the first matched group is the *key,* and the second matched group is the *value*.

It is hard to overdo this. More explanatory variables are generally better than fewer. It is remarkable how an opaque module can suddenly become transparent simply by breaking the calculations up into well-named intermediate values.

G20: *Function Names Should Say What They Do*

Look at this code:

```
Date newDate = date.add(5);
```

Would you expect this to add five days to the date? Or is it weeks, or hours? Is the `date` instance changed or does the function just return a new `Date` without changing the old one? *You can't tell from the call what the function does*.

If the function adds five days to the date and changes the date, then it should be called `addDaysTo` or `increaseByDays`. If, on the other hand, the function returns a new date that is five days later but does not change the date instance, it should be called `daysLater` or `daysSince`.

If you have to look at the implementation (or documentation) of the function to know what it does, then you should work to find a better name or rearrange the functionality so that it can be placed in functions with better names.

G21: *Understand the Algorithm*

Lots of very funny code is written because people don't take the time to understand the algorithm. They get something to work by plugging in enough `if` statements and flags, without really stopping to consider what is really going on.

Programming is often an exploration. You *think* you know the right algorithm for something, but then you wind up fiddling with it, prodding and poking at it, until you get it to "work." How do you know it "works"? Because it passes the test cases you can think of.

There is nothing wrong with this approach. Indeed, often it is the only way to get a function to do what you think it should. However, it is not sufficient to leave the quotation marks around the word "work."

Before you consider yourself to be done with a function, make sure you *understand* how it works. It is not good enough that it passes all the tests. You must *know*[10] that the solution is correct.

Often the best way to gain this knowledge and understanding is to refactor the function into something that is so clean and expressive that it is *obvious* how it works.

G22: *Make Logical Dependencies Physical*

If one module depends upon another, that dependency should be physical, not just logical. The dependent module should not make assumptions (in other words, logical dependencies) about the module it depends upon. Rather it should explicitly ask that module for all the information it depends upon.

For example, imagine that you are writing a function that prints a plain text report of hours worked by employees. One class named `HourlyReporter` gathers all the data into a convenient form and then passes it to `HourlyReportFormatter` to print it. (See Listing 17-1.)

Listing 17-1
`HourlyReporter.java`

```
public class HourlyReporter {
  private HourlyReportFormatter formatter;
  private List<LineItem> page;
  private final int PAGE_SIZE = 55;

  public HourlyReporter(HourlyReportFormatter formatter) {
    this.formatter = formatter;
    page = new ArrayList<LineItem>();
  }

  public void generateReport(List<HourlyEmployee> employees) {
    for (HourlyEmployee e : employees) {
      addLineItemToPage(e);
      if (page.size() == PAGE_SIZE)
        printAndClearItemList();
    }
    if (page.size() > 0)
      printAndClearItemList();
  }

  private void printAndClearItemList() {
    formatter.format(page);
    page.clear();
  }

  private void addLineItemToPage(HourlyEmployee e) {
    LineItem item = new LineItem();
    item.name = e.getName();
    item.hours = e.getTenthsWorked() / 10;
```

10. There is a difference between knowing how the code works and knowing whether the algorithm will do the job required of it. Being unsure that an algorithm is appropriate is often a fact of life. Being unsure what your code does is just laziness.

Listing 17-1 (continued)
HourlyReporter.java

```
      item.tenths = e.getTenthsWorked() % 10;
      page.add(item);
    }

    public class LineItem {
      public String name;
      public int hours;
      public int tenths;
    }
  }
```

This code has a logical dependency that has not been physicalized. Can you spot it? It is the constant `PAGE_SIZE`. Why should the `HourlyReporter` know the size of the page? Page size should be the responsibility of the `HourlyReportFormatter`.

The fact that PAGE_SIZE is declared in `HourlyReporter` represents a misplaced responsibility [G17] that causes `HourlyReporter` to assume that it knows what the page size ought to be. Such an assumption is a logical dependency. `HourlyReporter` depends on the fact that `HourlyReportFormatter` can deal with page sizes of 55. If some implementation of `HourlyReportFormatter` could not deal with such sizes, then there would be an error.

We can physicalize this dependency by creating a new method in `HourlyReportFormatter` named `getMaxPageSize()`. `HourlyReporter` will then call that function rather than using the `PAGE_SIZE` constant.

G23: *Prefer Polymorphism to If/Else or Switch/Case*

This might seem a strange suggestion given the topic of Chapter 6. After all, in that chapter I make the point that switch statements are probably appropriate in the parts of the system where adding new functions is more likely than adding new types.

First, most people use switch statements because it's the obvious brute force solution, not because it's the right solution for the situation. So this heuristic is here to remind us to consider polymorphism before using a switch.

Second, the cases where functions are more volatile than types are relatively rare. So *every* switch statement should be suspect.

I use the following "ONE SWITCH" rule: *There may be no more than one switch statement for a given type of selection. The cases in that switch statement must create polymorphic objects that take the place of other such switch statements in the rest of the system.*

G24: *Follow Standard Conventions*

Every team should follow a coding standard based on common industry norms. This coding standard should specify things like where to declare instance variables; how to name classes, methods, and variables; where to put braces; and so on. The team should not need a document to describe these conventions because their code provides the examples.

Everyone on the team should follow these conventions. This means that each team member must be mature enough to realize that it doesn't matter a whit where you put your braces so long as you all agree on where to put them.

If you would like to know what conventions I follow, you'll see them in the refactored code in Listing B-7 on page 394, through Listing B-14.

G25: *Replace Magic Numbers with Named Constants*

This is probably one of the oldest rules in software development. I remember reading it in the late sixties in introductory COBOL, FORTRAN, and PL/1 manuals. In general it is a bad idea to have raw numbers in your code. You should hide them behind well-named constants.

For example, the number 86,400 should be hidden behind the constant `SECONDS_PER_DAY`. If you are printing 55 lines per page, then the constant 55 should be hidden behind the constant `LINES_PER_PAGE`.

Some constants are so easy to recognize that they don't always need a named constant to hide behind so long as they are used in conjunction with very self-explanatory code. For example:

```
double milesWalked = feetWalked/5280.0;
int dailyPay = hourlyRate * 8;
double circumference = radius * Math.PI * 2;
```

Do we really need the constants `FEET_PER_MILE`, `WORK_HOURS_PER_DAY`, and `TWO` in the above examples? Clearly, the last case is absurd. There are some formulae in which constants are simply better written as raw numbers. You might quibble about the `WORK_HOURS_PER_DAY` case because the laws or conventions might change. On the other hand, that formula reads so nicely with the 8 in it that I would be reluctant to add 17 extra characters to the readers' burden. And in the `FEET_PER_MILE` case, the number 5280 is so very well known and so unique a constant that readers would recognize it even if it stood alone on a page with no context surrounding it.

Constants like 3.141592653589793 are also very well known and easily recognizable. However, the chance for error is too great to leave them raw. Every time someone sees 3.1415927535890793, they know that it is π, and so they fail to scrutinize it. (Did you catch the single-digit error?) We also don't want people using 3.14, 3.14159, 3.142, and so forth. Therefore, it is a good thing that `Math.PI` has already been defined for us.

The term "Magic Number" does not apply only to numbers. It applies to any token that has a value that is not self-describing. For example:

```
assertEquals(7777, Employee.find("John Doe").employeeNumber());
```

There are two magic numbers in this assertion. The first is obviously 7777, though what it might mean is not obvious. The second magic number is `"John Doe,"` and again the intent is not clear.

It turns out that `"John Doe"` is the name of employee #7777 in a well-known test database created by our team. Everyone in the team knows that when you connect to this

database, it will have several employees already cooked into it with well-known values and attributes. It also turns out that `"John Doe"` represents the sole hourly employee in that test database. So this test should really read:

```
assertEquals(
  HOURLY_EMPLOYEE_ID,
  Employee.find(HOURLY_EMPLOYEE_NAME).employeeNumber());
```

G26: *Be Precise*

Expecting the first match to be the *only* match to a query is probably naive. Using floating point numbers to represent currency is almost criminal. Avoiding locks and/or transaction management because you don't think concurrent update is likely is lazy at best. Declaring a variable to be an `ArrayList` when a `List` will due is overly constraining. Making all variables `protected` by default is not constraining enough.

When you make a decision in your code, make sure you make it *precisely*. Know why you have made it and how you will deal with any exceptions. Don't be lazy about the precision of your decisions. If you decide to call a function that might return `null`, make sure you check for `null`. If you query for what you think is the only record in the database, make sure your code checks to be sure there aren't others. If you need to deal with currency, use integers[11] and deal with rounding appropriately. If there is the possibility of concurrent update, make sure you implement some kind of locking mechanism.

Ambiguities and imprecision in code are either a result of disagreements or laziness. In either case they should be eliminated.

G27: *Structure over Convention*

Enforce design decisions with structure over convention. Naming conventions are good, but they are inferior to structures that force compliance. For example, switch/cases with nicely named enumerations are inferior to base classes with abstract methods. No one is forced to implement the `switch/case` statement the same way each time; but the base classes do enforce that concrete classes have all abstract methods implemented.

G28: *Encapsulate Conditionals*

Boolean logic is hard enough to understand without having to see it in the context of an `if` or `while` statement. Extract functions that explain the intent of the conditional.

For example:

```
if (shouldBeDeleted(timer))
```

is preferable to

```
if (timer.hasExpired() && !timer.isRecurrent())
```

---

11. Or better yet, a `Money` class that uses integers.

G29: *Avoid Negative Conditionals*

Negatives are just a bit harder to understand than positives. So, when possible, conditionals should be expressed as positives. For example:

```
if (buffer.shouldCompact())
```

is preferable to

```
if (!buffer.shouldNotCompact())
```

G30: *Functions Should Do One Thing*

It is often tempting to create functions that have multiple sections that perform a series of operations. Functions of this kind do more than *one thing*, and should be converted into many smaller functions, each of which does *one thing*.

For example:

```
public void pay() {
  for (Employee e : employees) {
    if (e.isPayday()) {
      Money pay = e.calculatePay();
      e.deliverPay(pay);
    }
  }
}
```

This bit of code does three things. It loops over all the employees, checks to see whether each employee ought to be paid, and then pays the employee. This code would be better written as:

```
public void pay() {
  for (Employee e : employees)
    payIfNecessary(e);
}

private void payIfNecessary(Employee e) {
  if (e.isPayday())
    calculateAndDeliverPay(e);
}

private void calculateAndDeliverPay(Employee e) {
  Money pay = e.calculatePay();
  e.deliverPay(pay);
}
```

Each of these functions does one thing. (See “Do One Thing” on page 35.)

G31: *Hidden Temporal Couplings*

Temporal couplings are often necessary, but you should not hide the coupling. Structure the arguments of your functions such that the order in which they should be called is obvious. Consider the following:

```
public class MoogDiver {
  Gradient gradient;
  List<Spline> splines;

  public void dive(String reason) {
    saturateGradient();
    reticulateSplines();
    diveForMoog(reason);
  }
  ...
}
```

The order of the three functions is important. You must saturate the gradient before you can reticulate the splines, and only then can you dive for the moog. Unfortunately, the code does not enforce this temporal coupling. Another programmer could call `reticulateSplines` before `saturateGradient` was called, leading to an `UnsaturatedGradientException`. A better solution is:

```
public class MoogDiver {
  Gradient gradient;
  List<Spline> splines;

  public void dive(String reason) {
    Gradient gradient = saturateGradient();
    List<Spline> splines = reticulateSplines(gradient);
    diveForMoog(splines, reason);
  }
  ...
}
```

This exposes the temporal coupling by creating a bucket brigade. Each function produces a result that the next function needs, so there is no reasonable way to call them out of order.

You might complain that this increases the complexity of the functions, and you'd be right. But that extra syntactic complexity exposes the true temporal complexity of the situation.

Note that I left the instance variables in place. I presume that they are needed by private methods in the class. Even so, I want the arguments in place to make the temporal coupling explicit.

G32: *Don't Be Arbitrary*

Have a reason for the way you structure your code, and make sure that reason is communicated by the structure of the code. If a structure appears arbitrary, others will feel empowered to change it. If a structure appears consistently throughout the system, others will use it and preserve the convention. For example, I was recently merging changes to FitNesse and discovered that one of our committers had done this:

```
public class AliasLinkWidget extends ParentWidget
{
  public static class VariableExpandingWidgetRoot {
    ...

  ...
}
```

The problem with this was that `VariableExpandingWidgetRoot` had no need to be inside the scope of `AliasLinkWidget`. Moreover, other unrelated classes made use of `AliasLinkWidget.VariableExpandingWidgetRoot`. These classes had no need to know about `AliasLinkWidget`.

Perhaps the programmer had plopped the `VariableExpandingWidgetRoot` into `AliasWidget` as a matter of convenience, or perhaps he thought it really needed to be scoped inside `AliasWidget`. Whatever the reason, the result wound up being arbitrary. Public classes that are not utilities of some other class should not be scoped inside another class. The convention is to make them public at the top level of their package.

G33: *Encapsulate Boundary Conditions*

Boundary conditions are hard to keep track of. Put the processing for them in one place. Don't let them leak all over the code. We don't want swarms of `+1`s and `-1`s scattered hither and yon. Consider this simple example from FIT:

```
if(level + 1 < tags.length)
{
  parts = new Parse(body, tags, level + 1, offset + endTag);
  body = null;
}
```

Notice that `level+1` appears twice. This is a boundary condition that should be encapsulated within a variable named something like `nextLevel`.

```
int nextLevel = level + 1;
if(nextLevel < tags.length)
{
  parts = new Parse(body, tags, nextLevel, offset + endTag);
  body = null;
}
```

G34: *Functions Should Descend Only One Level of Abstraction*

The statements within a function should all be written at the same level of abstraction, which should be one level below the operation described by the name of the function. This may be the hardest of these heuristics to interpret and follow. Though the idea is plain enough, humans are just far too good at seamlessly mixing levels of abstraction. Consider, for example, the following code taken from FitNesse:

```
public String render() throws Exception
{
  StringBuffer html = new StringBuffer("<hr");
  if(size > 0)
    html.append(" size=\"").append(size + 1).append("\"");
  html.append(">");

  return html.toString();
}
```

A moment's study and you can see what's going on. This function constructs the HTML tag that draws a horizontal rule across the page. The height of that rule is specified in the `size` variable.

Now look again. This method is mixing at least two levels of abstraction. The first is the notion that a horizontal rule has a size. The second is the syntax of the `HR` tag itself. This code comes from the `HruleWidget` module in FitNesse. This module detects a row of four or more dashes and converts it into the appropriate HR tag. The more dashes, the larger the size.

I refactored this bit of code as follows. Note that I changed the name of the `size` field to reflect its true purpose. It held the number of extra dashes.

```
public String render() throws Exception
{
  HtmlTag hr = new HtmlTag("hr");
  if (extraDashes > 0)
    hr.addAttribute("size", hrSize(extraDashes));
  return hr.html();
}

private String hrSize(int height)
{
  int hrSize = height + 1;
  return String.format("%d", hrSize);
}
```

This change separates the two levels of abstraction nicely. The `render` function simply constructs an HR tag, without having to know anything about the HTML syntax of that tag. The `HtmlTag` module takes care of all the nasty syntax issues.

Indeed, by making this change I caught a subtle error. The original code did not put the closing slash on the HR tag, as the XHTML standard would have it. (In other words, it emitted `<hr>` instead of `<hr/>`.) The `HtmlTag` module had been changed to conform to XHTML long ago.

Separating levels of abstraction is one of the most important functions of refactoring, and it's one of the hardest to do well. As an example, look at the code below. This was my first attempt at separating the abstraction levels in the `HruleWidget.render method`.

```
public String render() throws Exception
{
  HtmlTag hr = new HtmlTag("hr");
  if (size > 0) {
    hr.addAttribute("size", ""+(size+1));
  }
  return hr.html();
}
```

My goal, at this point, was to create the necessary separation and get the tests to pass. I accomplished that goal easily, but the result was a function that *still* had mixed levels of abstraction. In this case the mixed levels were the construction of the HR tag and the

interpretation and formatting of the `size` variable. This points out that when you break a function along lines of abstraction, you often uncover new lines of abstraction that were obscured by the previous structure.

G35: *Keep Configurable Data at High Levels*

If you have a constant such as a default or configuration value that is known and expected at a high level of abstraction, do not bury it in a low-level function. Expose it as an argument to that low-level function called from the high-level function. Consider the following code from FitNesse:

```
  public static void main(String[] args) throws Exception
  {
    Arguments arguments = parseCommandLine(args);
    ...
  }

public class Arguments
{
  public static final String DEFAULT_PATH = ".";
  public static final String DEFAULT_ROOT = "FitNesseRoot";
  public static final int DEFAULT_PORT = 80;
  public static final int DEFAULT_VERSION_DAYS = 14;
  ...
}
```

The command-line arguments are parsed in the very first executable line of FitNesse. The default values of those arguments are specified at the top of the `Argument` class. You don't have to go looking in low levels of the system for statements like this one:

```
if (arguments.port == 0) // use 80 by default
```

The configuration constants reside at a very high level and are easy to change. They get passed down to the rest of the application. The lower levels of the application do not own the values of these constants.

G36: *Avoid Transitive Navigation*

In general we don't want a single module to know much about its collaborators. More specifically, if `A` collaborates with `B`, and `B` collaborates with `C`, we don't want modules that use `A` to know about `C`. (For example, we don't want `a.getB().getC().doSomething();`.)

This is sometimes called the Law of Demeter. The Pragmatic Programmers call it "Writing Shy Code."[12] In either case it comes down to making sure that modules know only about their immediate collaborators and do not know the navigation map of the whole system.

If many modules used some form of the statement `a.getB().getC()`, then it would be difficult to change the design and architecture to interpose a `Q` between `B` and `C`. You'd

12. [PRAG], p. 138.

have to find every instance of `a.getB().getC()` and convert it to `a.getB().getQ().getC()`. This is how architectures become rigid. Too many modules know too much about the architecture.

Rather we want our immediate collaborators to offer all the services we need. We should not have to roam through the object graph of the system, hunting for the method we want to call. Rather we should simply be able to say:

```
myCollaborator.doSomething().
```

Java

J1: *Avoid Long Import Lists by Using Wildcards*

If you use two or more classes from a package, then import the whole package with

```
import package.*;
```

Long lists of imports are daunting to the reader. We don't want to clutter up the tops of our modules with 80 lines of imports. Rather we want the imports to be a concise statement about which packages we collaborate with.

Specific imports are hard dependencies, whereas wildcard imports are not. If you specifically import a class, then that class *must* exist. But if you import a package with a wildcard, no particular classes need to exist. The import statement simply adds the package to the search path when hunting for names. So no true dependency is created by such imports, and they therefore serve to keep our modules less coupled.

There are times when the long list of specific imports can be useful. For example, if you are dealing with legacy code and you want to find out what classes you need to build mocks and stubs for, you can walk down the list of specific imports to find out the true qualified names of all those classes and then put the appropriate stubs in place. However, this use for specific imports is very rare. Furthermore, most modern IDEs will allow you to convert the wildcarded imports to a list of specific imports with a single command. So even in the legacy case it's better to import wildcards.

Wildcard imports can sometimes cause name conflicts and ambiguities. Two classes with the same name, but in different packages, will need to be specifically imported, or at least specifically qualified when used. This can be a nuisance but is rare enough that using wildcard imports is still generally better than specific imports.

J2: *Don't Inherit Constants*

I have seen this several times and it always makes me grimace. A programmer puts some constants in an interface and then gains access to those constants by inheriting that interface. Take a look at the following code:

```
public class HourlyEmployee extends Employee {
  private int tenthsWorked;
  private double hourlyRate;
```

```
  public Money calculatePay() {
    int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
    int overTime = tenthsWorked - straightTime;
    return new Money(
      hourlyRate * (tenthsWorked + OVERTIME_RATE * overTime)
    );
  }
  ...
}
```

Where did the constants `TENTHS_PER_WEEK` and `OVERTIME_RATE` come from? They might have come from class `Employee`; so let's take a look at that:

```
public abstract class Employee implements PayrollConstants {
  public abstract boolean isPayday();
  public abstract Money calculatePay();
  public abstract void deliverPay(Money pay);
}
```

Nope, not there. But then where? Look closely at class `Employee`. It implements `PayrollConstants`.

```
public interface PayrollConstants {
  public static final int TENTHS_PER_WEEK = 400;
  public static final double OVERTIME_RATE = 1.5;
}
```

This is a hideous practice! The constants are hidden at the top of the inheritance hierarchy. Ick! Don't use inheritance as a way to cheat the scoping rules of the language. Use a static import instead.

```
import static PayrollConstants.*;

public class HourlyEmployee extends Employee {
  private int tenthsWorked;
  private double hourlyRate;

  public Money calculatePay() {
    int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
    int overTime = tenthsWorked - straightTime;
    return new Money(
      hourlyRate * (tenthsWorked + OVERTIME_RATE * overTime)
    );
  }
  ...
}
```

J3: *Constants versus Enums*

Now that `enum`s have been added to the language (Java 5), use them! Don't keep using the old trick of `public static final int`s. The meaning of `int`s can get lost. The meaning of `enum`s cannot, because they belong to an enumeration that is named.

What's more, study the syntax for `enum`s carefully. They can have methods and fields. This makes them very powerful tools that allow much more expression and flexibility than `int`s. Consider this variation on the payroll code:

```
public class HourlyEmployee extends Employee {
  private int tenthsWorked;
  HourlyPayGrade grade;

  public Money calculatePay() {
    int straightTime = Math.min(tenthsWorked, TENTHS_PER_WEEK);
    int overTime = tenthsWorked - straightTime;
    return new Money(
      grade.rate() * (tenthsWorked + OVERTIME_RATE * overTime)
    );
  }
  ...
}

public enum HourlyPayGrade {
  APPRENTICE {
    public double rate() {
      return 1.0;
    }
  },
  LEUTENANT_JOURNEYMAN {
    public double rate() {
      return 1.2;
    }
  },
  JOURNEYMAN {
    public double rate() {
      return 1.5;
    }
  },
  MASTER {
    public double rate() {
      return 2.0;
    }
  };

  public abstract double rate();
}
```

Names

N1: *Choose Descriptive Names*

Don’t be too quick to choose a name. Make sure the name is descriptive. Remember that meanings tend to drift as software evolves, so frequently reevaluate the appropriateness of the names you choose.

This is not just a “feel-good” recommendation. Names in software are 90 percent of what make software readable. You need to take the time to choose them wisely and keep them relevant. Names are too important to treat carelessly.

Consider the code below. What does it do? If I show you the code with well-chosen names, it will make perfect sense to you, but like this it’s just a hodge-podge of symbols and magic numbers.

```
public int x() {
    int q = 0;
    int z = 0;
    for (int kk = 0; kk < 10; kk++) {
      if (l[z] == 10)
      {
        q += 10 + (l[z + 1] + l[z + 2]);
        z += 1;
      }
      else if (l[z] + l[z + 1] == 10)
      {
        q += 10 + l[z + 2];
        z += 2;
      } else {
        q += l[z] + l[z + 1];
        z += 2;
      }
    }
    return q;
  }
```

Here is the code the way it should be written. This snippet is actually less complete than the one above. Yet you can infer immediately what it is trying to do, and you could very likely write the missing functions based on that inferred meaning. The magic numbers are no longer magic, and the structure of the algorithm is compellingly descriptive.

```
public int score() {
  int score = 0;
  int frame = 0;
  for (int frameNumber = 0; frameNumber < 10; frameNumber++) {
    if (isStrike(frame)) {
      score += 10 + nextTwoBallsForStrike(frame);
      frame += 1;
    } else if (isSpare(frame)) {
      score += 10 + nextBallForSpare(frame);
      frame += 2;
    } else {
      score += twoBallsInFrame(frame);
      frame += 2;
    }
  }
  return score;
}
```

The power of carefully chosen names is that they overload the structure of the code with description. That overloading sets the readers' expectations about what the other functions in the module do. You can infer the implementation of `isStrike()` by looking at the code above. When you read the `isStrike` method, it will be "pretty much what you expected."[13]

```
private boolean isStrike(int frame) {
  return rolls[frame] == 10;
}
```

13. See Ward Cunningham's quote on page 11.

N2: *Choose Names at the Appropriate Level of Abstraction*

Don't pick names that communicate implementation; choose names the reflect the level of abstraction of the class or function you are working in. This is hard to do. Again, people are just too good at mixing levels of abstractions. Each time you make a pass over your code, you will likely find some variable that is named at too low a level. You should take the opportunity to change those names when you find them. Making code readable requires a dedication to continuous improvement. Consider the `Modem` interface below:

```
public interface Modem {
  boolean dial(String phoneNumber);
  boolean disconnect();
  boolean send(char c);
  char recv();
  String getConnectedPhoneNumber();
}
```

At first this looks fine. The functions all seem appropriate. Indeed, for many applications they are. But now consider an application in which some modems aren't connected by dialling. Rather they are connected permanently by hard wiring them together (think of the cable modems that provide Internet access to most homes nowadays). Perhaps some are connected by sending a port number to a switch over a USB connection. Clearly the notion of phone numbers is at the wrong level of abstraction. A better naming strategy for this scenario might be:

```
public interface Modem {
  boolean connect(String connectionLocator);
  boolean disconnect();
  boolean send(char c);
  char recv();
  String getConnectedLocator();
}
```

Now the names don't make any commitments about phone numbers. They can still be used for phone numbers, or they could be used for any other kind of connection strategy.

N3: *Use Standard Nomenclature Where Possible*

Names are easier to understand if they are based on existing convention or usage. For example, if you are using the DECORATOR pattern, you should use the word `Decorator` in the names of the decorating classes. For example, `AutoHangupModemDecorator` might be the name of a class that decorates a `Modem` with the ability to automatically hang up at the end of a session.

Patterns are just one kind of standard. In Java, for example, functions that convert objects to string representations are often named `toString`. It is better to follow conventions like these than to invent your own.

Teams will often invent their own standard system of names for a particular project. Eric Evans refers to this as a *ubiquitous language* for the project.[14] Your code should use

14. [DDD].

the terms from this language extensively. In short, the more you can use names that are overloaded with special meanings that are relevant to your project, the easier it will be for readers to know what your code is talking about.

N4: *Unambiguous Names*

Choose names that make the workings of a function or variable unambiguous. Consider this example from FitNesse:

```
private String doRename() throws Exception
{
  if(refactorReferences)
    renameReferences();
  renamePage();

  pathToRename.removeNameFromEnd();
  pathToRename.addNameToEnd(newName);
  return PathParser.render(pathToRename);
}
```

The name of this function does not say what the function does except in broad and vague terms. This is emphasized by the fact that there is a function named `renamePage` inside the function named `doRename`! What do the names tell you about the difference between the two functions? Nothing.

A better name for that function is `renamePageAndOptionallyAllReferences`. This may seem long, and it is, but it's only called from one place in the module, so it's explanatory value outweighs the length.

N5: *Use Long Names for Long Scopes*

The length of a name should be related to the length of the scope. You can use very short variable names for tiny scopes, but for big scopes you should use longer names.

Variable names like `i` and `j` are just fine if their scope is five lines long. Consider this snippet from the old standard "Bowling Game":

```
private void rollMany(int n, int pins)
{
  for (int i=0; i<n; i++)
    g.roll(pins);
}
```

This is perfectly clear and would be obfuscated if the variable `i` were replaced with something annoying like `rollCount`. On the other hand, variables and functions with short names lose their meaning over long distances. So the longer the scope of the name, the longer and more precise the name should be.

N6: *Avoid Encodings*

Names should not be encoded with type or scope information. Prefixes such as `m_` or `f` are useless in today's environments. Also project and/or subsystem encodings such as

`vis_` (for visual imaging system) are distracting and redundant. Again, today's environments provide all that information without having to mangle the names. Keep your names free of Hungarian pollution.

N7: *Names Should Describe Side-Effects*

Names should describe everything that a function, variable, or class is or does. Don't hide side effects with a name. Don't use a simple verb to describe a function that does more than just that simple action. For example, consider this code from TestNG:

```
public ObjectOutputStream getOos() throws IOException {
  if (m_oos == null) {
    m_oos = new ObjectOutputStream(m_socket.getOutputStream());
  }
  return m_oos;
}
```

This function does a bit more than get an "oos"; it creates the "oos" if it hasn't been created already. Thus, a better name might be `createOrReturnOos`.

Tests

T1: *Insufficient Tests*

How many tests should be in a test suite? Unfortunately, the metric many programmers use is "That seems like enough." A test suite should test everything that could possibly break. The tests are insufficient so long as there are conditions that have not been explored by the tests or calculations that have not been validated.

T2: *Use a Coverage Tool!*

Coverage tools reports gaps in your testing strategy. They make it easy to find modules, classes, and functions that are insufficiently tested. Most IDEs give you a visual indication, marking lines that are covered in green and those that are uncovered in red. This makes it quick and easy to find `if` or `catch` statements whose bodies haven't been checked.

T3: *Don't Skip Trivial Tests*

They are easy to write and their documentary value is higher than the cost to produce them.

T4: *An Ignored Test Is a Question about an Ambiguity*

Sometimes we are uncertain about a behavioral detail because the requirements are unclear. We can express our question about the requirements as a test that is commented out, or as a test that annotated with `@Ignore`. Which you choose depends upon whether the ambiguity is about something that would compile or not.

T5: *Test Boundary Conditions*

Take special care to test boundary conditions. We often get the middle of an algorithm right but misjudge the boundaries.

T6: *Exhaustively Test Near Bugs*

Bugs tend to congregate. When you find a bug in a function, it is wise to do an exhaustive test of that function. You'll probably find that the bug was not alone.

T7: *Patterns of Failure Are Revealing*

Sometimes you can diagnose a problem by finding patterns in the way the test cases fail. This is another argument for making the test cases as complete as possible. Complete test cases, ordered in a reasonable way, expose patterns.

As a simple example, suppose you noticed that all tests with an input larger than five characters failed? Or what if any test that passed a negative number into the second argument of a function failed? Sometimes just seeing the pattern of red and green on the test report is enough to spark the "Aha!" that leads to the solution. Look back at page 267 to see an interesting example of this in the `SerialDate` example.

T8: *Test Coverage Patterns Can Be Revealing*

Looking at the code that is or is not executed by the passing tests gives clues to why the failing tests fail.

T9: *Tests Should Be Fast*

A slow test is a test that won't get run. When things get tight, it's the slow tests that will be dropped from the suite. So *do what you must* to keep your tests fast.

Conclusion

This list of heuristics and smells could hardly be said to be complete. Indeed, I'm not sure that such a list can *ever* be complete. But perhaps completeness should not be the goal, because what this list *does* do is imply a value system.

Indeed, that value system has been the goal, and the topic, of this book. Clean code is not written by following a set of rules. You don't become a software craftsman by learning a list of heuristics. Professionalism and craftsmanship come from values that drive disciplines.

Bibliography

[Refactoring]: *Refactoring: Improving the Design of Existing Code*, Martin Fowler et al., Addison-Wesley, 1999.

[PRAG]: *The Pragmatic Programmer*, Andrew Hunt, Dave Thomas, Addison-Wesley, 2000.

[GOF]: *Design Patterns: Elements of Reusable Object Oriented Software*, Gamma et al., Addison-Wesley, 1996.

[Beck97]: *Smalltalk Best Practice Patterns*, Kent Beck, Prentice Hall, 1997.

[Beck07]: *Implementation Patterns*, Kent Beck, Addison-Wesley, 2008.

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

[DDD]: *Domain Driven Design*, Eric Evans, Addison-Wesley, 2003.

Appendix A

Concurrency II

by Brett L. Schuchert

This appendix supports and amplifies the *Concurrency* chapter on page 177. It is written as a series of independent topics and you can generally read them in any order. There is some duplication between sections to allow for such reading.

Client/Server Example

Imagine a simple client/server application. A server sits and waits listening on a socket for a client to connect. A client connects and sends a request.

The Server

Here is a simplified version of a server application. Full source for this example is available starting on page 343, *Client/Server Nonthreaded*.

```
ServerSocket serverSocket = new ServerSocket(8009);

while (keepProcessing) {
    try {
        Socket socket = serverSocket.accept();
        process(socket);
    } catch (Exception e) {
        handle(e);
    }
}
```

This simple application waits for a connection, processes an incoming message, and then again waits for the next client request to come in. Here's client code that connects to this server:

```
private void connectSendReceive(int i) {
    try {
        Socket socket = new Socket("localhost", PORT);
        MessageUtils.sendMessage(socket, Integer.toString(i));
        MessageUtils.getMessage(socket);
        socket.close();
    } catch (Exception e) {
        e.printStackTrace();

    }

}
```

How well does this client/server pair perform? How can we formally describe that performance? Here's a test that asserts that the performance is "acceptable":

```
@Test(timeout = 10000)
public void shouldRunInUnder10Seconds() throws Exception {
    Thread[] threads = createThreads();
    startAllThreadsw(threads);
    waitForAllThreadsToFinish(threads);
}
```

The setup is left out to keep the example simple (see "`ClientTest.java`" on page 344). This test asserts that it should complete within 10,000 milliseconds.

This is a classic example of validating the throughput of a system. This system should complete a series of client requests in ten seconds. So long as the server can process each individual client request in time, the test will pass.

What happens if the test fails? Short of developing some kind of event polling loop, there is not much to do within a single thread that will make this code any faster. Will using multiple threads solve the problem? It might, but we need to know where the time is being spent. There are two possibilities:

- I/O—using a socket, connecting to a database, waiting for virtual memory swapping, and so on.
- Processor—numerical calculations, regular expression processing, garbage collection, and so on.

Systems typically have some of each, but for a given operation one tends to dominate. If the code is processor bound, more processing hardware can improve throughput, making our test pass. But there are only so many CPU cycles available, so adding threads to a processor-bound problem will not make it go faster.

On the other hand, if the process is I/O bound, then concurrency can increase efficiency. When one part of the system is waiting for I/O, another part can use that wait time to process something else, making more effective use of the available CPU.

Adding Threading

Assume for the moment that the performance test fails. How can we improve the throughput so that the performance test passes? If the `process` method of the server is I/O bound, then here is one way to make the server use threads (just change the `processMessage`):

```
void process(final Socket socket) {
    if (socket == null)
        return;

    Runnable clientHandler = new Runnable() {
        public void run() {
            try {
                String message = MessageUtils.getMessage(socket);
                MessageUtils.sendMessage(socket, "Processed: " + message);
                closeIgnoringException(socket);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    };

    Thread clientConnection = new Thread(clientHandler);
    clientConnection.start();
}
```

Assume that this change causes the test to pass;[1] the code is complete, correct?

Server Observations

The updated server completes the test successfully in just over one second. Unfortunately, this solution is a bit naive and introduces some new problems.

How many threads might our server create? The code sets no limit, so the we could feasibly hit the limit imposed by the Java Virtual Machine (JVM). For many simple systems this may suffice. But what if the system is meant to support many users on the public net? If too many users connect at the same time, the system might grind to a halt.

But set the behavioral problem aside for the moment. The solution shown has problems of cleanliness and structure. How many responsibilities does the server code have?

- Socket connection management
- Client processing
- Threading policy
- Server shutdown policy

Unfortunately, all these responsibilities live in the `process` function. In addition, the code crosses many different levels of abstraction. So, small as the process function is, it needs to be repartitioned.

---

1. You can verify that for yourself by trying out the before and after code. Review the nonthreaded code starting on page 343. Review the threaded code starting on page 346.

The server has several reasons to change; therefore it violates the Single Responsibility Principle. To keep concurrent systems clean, thread management should be kept to a few, well-controlled places. What's more, any code that manages threads should do nothing other than thread management. Why? If for no other reason than that tracking down concurrency issues is hard enough without having to unwind other nonconcurrency issues at the same time.

If we create a separate class for each of the responsibilities listed above, including the thread management responsibility, then when we change the thread management strategy, the change will impact less overall code and will not pollute the other responsibilities. This also makes it much easier to test all the other responsibilities without having to worry about threading. Here is an updated version that does just that:

```
public void run() {
  while (keepProcessing) {
   try {
    ClientConnection clientConnection = connectionManager.awaitClient();
    ClientRequestProcessor requestProcessor
      = new ClientRequestProcessor(clientConnection);
    clientScheduler.schedule(requestProcessor);
    } catch (Exception e) {
      e.printStackTrace();
    }

  }
  connectionManager.shutdown();

}
```

This now focuses all things thread-related into one place, `clientScheduler`. If there are concurrency problems, there is just one place to look:

```
public interface ClientScheduler {
    void schedule(ClientRequestProcessor requestProcessor);
}
```

The current policy is easy to implement:

```
public class ThreadPerRequestScheduler implements ClientScheduler {
    public void schedule(final ClientRequestProcessor requestProcessor) {
        Runnable runnable = new Runnable() {
            public void run() {
                requestProcessor.process();
            }
        };

       Thread thread = new Thread(runnable);
       thread.start();
    }
}
```

Having isolated all the thread management into a single place, it is much easier to change the way we control threads. For example, moving to the Java 5 Executor framework involves writing a new class and plugging it in (Listing A-1).

Listing A-1
ExecutorClientScheduler.java

```
import java.util.concurrent.Executor;
import java.util.concurrent.Executors;

public class ExecutorClientScheduler implements ClientScheduler {
    Executor executor;

    public ExecutorClientScheduler(int availableThreads) {
        executor = Executors.newFixedThreadPool(availableThreads);
    }

    public void schedule(final ClientRequestProcessor requestProcessor) {
        Runnable runnable = new Runnable() {
            public void run() {
                requestProcessor.process();
            }
        };
        executor.execute(runnable);
    }
}
```

Conclusion

Introducing concurrency in this particular example demonstrates a way to improve the throughput of a system and one way of validating that throughput through a testing framework. Focusing all concurrency code into a small number of classes is an example of applying the Single Responsibility Principle. In the case of concurrent programming, this becomes especially important because of its complexity.

Possible Paths of Execution

Review the method `incrementValue`, a one-line Java method with no looping or branching:

```
public class IdGenerator {
  int lastIdUsed;

  public int incrementValue() {
    return ++lastIdUsed;
  }
}
```

Ignore integer overflow and assume that only one thread has access to a single instance of `IdGenerator`. In this case there is a single path of execution and a single guaranteed result:

- The value returned is equal to the value of `lastIdUsed`, both of which are one greater than just before calling the method.

What happens if we use two threads and leave the method unchanged? What are the possible outcomes if each thread calls `incrementValue` once? How many possible paths of execution are there? First, the outcomes (assume `lastIdUsed` starts with a value of 93):

- Thread 1 gets the value of 94, thread 2 gets the value of 95, and `lastIdUsed` is now 95.
- Thread 1 gets the value of 95, thread 2 gets the value of 94, and `lastIdUsed` is now 95.
- Thread 1 gets the value of 94, thread 2 gets the value of 94, and `lastIdUsed` is now 94.

The final result, while surprising, is possible. To see how these different results are possible, we need to understand the number of possible paths of execution and how the Java Virtual Machine executes them.

Number of Paths

To calculate the number of possible execution paths, we'll start with the generated byte-code. The one line of java (`return ++lastIdUsed;`) becomes eight byte-code instructions. It is possible for the two threads to interleave the execution of these eight instructions the way a card dealer interleaves cards as he shuffles a deck.[2] Even with only eight cards in each hand, there are a remarkable number of shuffled outcomes.

For this simple case of N instructions in a sequence, no looping or conditionals, and T threads, the total number of possible execution paths is equal to

$$\frac{(NT)!}{N!^T}$$

Calculating the Possible Orderings

This comes from an email from Uncle Bob to Brett:

With N steps and T threads there are $T * N$ total steps. Prior to each step there is a context switch that chooses between the T threads. Each path can thus be represented as a string of digits denoting the context switches. Given steps A and B and threads 1 and 2, the six possible paths are 1122, 1212, 1221, 2112, 2121, and 2211. Or, in terms of steps it is A1B1A2B2, A1A2B1B2, A1A2B2B1, A2A1B1B2, A2A1B2B1, and A2B2A1B1. For three threads the sequence is 112233, 112323, 113223, 113232, 112233, 121233, 121323, 121332, 123132, 123123, ...

One characteristic of these strings is that there must always be N instances of each T. So the string 111111 is invalid because it has six instances of 1 and zero instances of 2 and 3.

2. This is a bit of a simplification. However, for the purpose of this discussion, we can use this simplifying model.

Calculating the Possible Orderings (continued)

So we want the permutations of *N* 1's, *N* 2's, . . . and *N T*'s. This is really just the permutations of *N* * *T* things taken *N* * *T* at a time, which is (*N* * *T*)!, but with all the duplicates removed. So the trick is to count the duplicates and subtract that from (*N* * *T*)!.

Given two steps and two threads, how many duplicates are there? Each four-digit string has two 1s and two 2s. Each of those pairs could be swapped without changing the sense of the string. You could swap the 1s or the 2s both, or neither. So there are four isomorphs for each string, which means that there are three duplicates. So three out of four of the options are duplicates; alternatively one of four of the permutations are NOT duplicates. 4! * .25 = 6. So this reasoning seems to work.

How many duplicates are there? In the case where *N* = 2 and *T* = 2, I could swap the 1s, the 2s, or both. In the case where *N* = 2 and *T* = 3, I could swap the 1s, the 2s, the 3s, 1s and 2s, 1s and 3s, or 2s and 3s. Swapping is just the permutations of *N*. Let's say there are *P* permutations of *N*. The number of different ways to arrange those permutations are *P***T*.

So the number of possible isomorphs is *N*!***T*. And so the number of paths is (*T***N*)!/(*N*!***T*). Again, in our *T* = 2, *N* = 2 case we get 6 (24/4).

For *N* = 2 and *T* = 3 we get 720/8 = 90.

For *N* = 3 and *T* = 3 we get 9!/6^3 = 1680.

For our simple case of one line of Java code, which equates to eight lines of byte-code and two threads, the total number of possible paths of execution is 12,870. If the type of `lastIdUsed` is a `long`, then every read/write becomes two operations instead of one, and the number of possible orderings becomes 2,704,156.

What happens if we make one change to this method?

```
public synchronized void incrementValue() {
    ++lastIdUsed;
}
```

The number of possible execution pathways becomes two for two threads and N! in the general case.

Digging Deeper

What about the surprising result that two threads could both call the method once (before we added `synchronized`) and get the same numeric result? How is that possible? First things first.

What is an atomic operation? We can define an atomic operation as any operation that is uninterruptable. For example, in the following code, line 5, where 0 is assigned to `lastid`, is atomic because according to the Java Memory model, assignment to a 32-bit value is uninterruptable.

```
01: public class Example {
02:     int lastId;
03:
04:     public void resetId() {
05:         value = 0;
06:     }
07:
08:     public int getNextId() {
09:         ++value;
10:     }
11:}
```

What happens if we change type of `lastId` from `int` to `long`? Is line 5 still atomic? Not according to the JVM specification. It could be atomic on a particular processor, but according to the JVM specification, assignment to any 64-bit value requires two 32-bit assignments. This means that between the first 32-bit assignment and the second 32-bit assignment, some other thread could sneak in and change one of the values.

What about the pre-increment operator, ++, on line 9? The pre-increment operator can be interrupted, so it is not atomic. To understand, let's review the byte-code of both of these methods in detail.

Before we go any further, here are three definitions that will be important:

- Frame—Every method invocation requires a frame. The frame includes the return address, any parameters passed into the method and the local variables defined in the method. This is a standard technique used to define a call stack, which is used by modern languages to allow for basic function/method invocation and to allow for recursive invocation.
- Local variable—Any variables defined in the scope of the method. All nonstatic methods have at least one variable, **`this`**, which represents the current object, the object that received the most recent message (in the current thread), which caused the method invocation.
- Operand stack—Many of the instructions in the Java Virtual Machine take parameters. The operand stack is where those parameters are put. The stack is a standard last-in, first-out (LIFO) data structure.

Here is the byte-code generated for `resetId()`:

Mnemonic	Description	Operand Stack After
`ALOAD 0`	Load the 0th variable onto the operand stack. What is the 0th variable? It is **`this`**., the current object. When the method was called, the receiver of the message, an instance of `Example`, was pushed into the local variable array of the frame created for method invocation. This is always the first variable put in every instance method.	`this`

Mnemonic	Description	Operand Stack After
`ICONST_0`	Put the constant value 0 onto the operand stack.	`this, 0`
`PUTFIELD lastId`	Store the top value on the stack (which is 0) into the field value of the object referred to by the object reference one away from the top of the stack, **`this`**.	<empty>

These three instructions are guaranteed to be atomic because, although the thread executing them could be interrupted after any one of them, the information for the PUTFIELD instruction (the constant value 0 on the top of the stack and the reference to `this` one below the top, along with the field value) cannot be touched by another thread. So when the assignment occurs, we are guaranteed that the value 0 will be stored in the field value. The operation is atomic. The operands all deal with information local to the method, so there is no interference between multiple threads.

So if these three instructions are executed by ten threads, there are 4.38679733629e+24 possible orderings. However, there is only one possible outcome, so the different orderings are irrelevant. It just so happens that the same outcome is guaranteed for longs in this case as well. Why? All ten threads are assigning a constant value. Even if they interleave with each other, the end result is the same.

With the `++` operation in the `getNextId` method, there are going to be problems. Assume that `lastId` holds 42 at the beginning of this method. Here is the byte-code for this new method:

Mnemonic	Description	Operand Stack After
`ALOAD 0`	Load `this` onto the operand stack	`this`
`DUP`	Copy the top of the stack. We now have two copies of `this` on the operand stack.	`this`, `this`
`GETFIELD lastId`	Retrieve the value of the field `lastId` from the object pointed to on the top of the stack (`this`) and store that value back on to the stack.	`this`, 42
`ICONST_1`	Push the integer constant 1 on the stack.	`this`, 42, 1
`IADD`	Integer add the top two values on the operand stack and store the result back on to the operand stack.	`this`, 43
`DUP_X1`	Duplicate the value 43 and put it before `this`.	43, `this`, 43
`PUTFIELD value`	Store the top value on the operand stack, 43, into the field value of the current object, represented by the next-to-top value on the operand stack, `this`.	43
`IRETURN`	return the top (and only) value on the stack.	<empty>

Imagine the case where the first thread completes the first three instructions, up to and including GETFIELD, and then it is interrupted. A second thread takes over and performs the entire method, incrementing `lastId` by one; it gets 43 back. Then the first thread picks up where it left off; 42 is still on the operand stack because that was the value of `lastId` when it executed GETFIELD. It adds one to get 43 again and stores the result. The value 43 is returned to the first thread as well. The result is that one of the increments is lost because the first thread stepped on the second thread after the second thread interrupted the first thread.

Making the `getNexId()` method synchronized fixes this problem.

Conclusion

An intimate understanding of byte-code is not necessary to understand how threads can step on each other. If you can understand this one example, it should demonstrate the possibility of multiple threads stepping on each other, which is enough knowledge.

That being said, what this trivial example demonstrates is a need to understand the memory model enough to know what is and is not safe. It is a common misconception that the ++ (pre- or post-increment) operator is atomic, and it clearly is not. This means you need to know:

- Where there are shared objects/values
- The code that can cause concurrent read/update issues
- How to guard such concurrent issues from happening

Knowing Your Library

Executor Framework

As demonstrated in the `ExecutorClientScheduler.java` on page 321, the `Executor` framework introduced in Java 5 allows for sophisticated execution using thread pools. This is a class in the `java.util.concurrent` package.

If you are creating threads and are not using a thread pool or *are* using a hand-written one, you should consider using the `Executor`. It will make your code cleaner, easier to follow, and smaller.

The `Executor` framework will pool threads, resize automatically, and recreate threads if necessary. It also supports *futures,* a common concurrent programming construct. The `Executor` framework works with classes that implement `Runnable` and also works with classes that implement the `Callable` interface. A `Callable` looks like a `Runnable`, but it can return a result, which is a common need in multithreaded solutions.

A *future* is handy when code needs to execute multiple, independent operations and wait for both to finish:

```
public String processRequest(String message) throws Exception {
    Callable<String> makeExternalCall = new Callable<String>() {
```

```
            public String call() throws Exception {
                String result = "";
                // make external request
                return result;
            }
        };

        Future<String> result = executorService.submit(makeExternalCall);
        String partialResult = doSomeLocalProcessing();
        return result.get() + partialResult;
    }
```

In this example, the method starts executing the `makeExternalCall` object. The method continues other processing. The final line calls `result.get()`, which blocks until the future completes.

Nonblocking Solutions

The Java 5 VM takes advantage of modern processor design, which supports reliable, nonblocking updates. Consider, for example, a class that uses synchronization (and therefore blocking) to provide a thread-safe update of a value:

```
public class ObjectWithValue {
    private int value;
    public void synchronized incrementValue() { ++value; }
    public int getValue() { return value; }
}
```

Java 5 has a series of new classes for situations like this: `AtomicBoolean`, `AtomicInteger`, and `AtomicReference` are three examples; there are several more. We can rewrite the above code to use a nonblocking approach as follows:

```
public class ObjectWithValue {
    private AtomicInteger value = new AtomicInteger(0);

    public void incrementValue() {
        value.incrementAndGet();
    }
    public int getValue() {
        return value.get();
    }
}
```

Even though this uses an object instead of a primitive and sends messages like `incrementAndGet()` instead of ++, the performance of this class will nearly always beat the previous version. In some cases it will only be slightly faster, but the cases where it will be slower are virtually nonexistent.

How is this possible? Modern processors have an operation typically called *Compare and Swap (CAS)*. This operation is analogous to optimistic locking in databases, whereas the synchronized version is analogous to pessimistic locking.

The `synchronized` keyword always acquires a lock, even when a second thread is not trying to update the same value. Even though the performance of intrinsic locks has improved from version to version, they are still costly.

The nonblocking version starts with the assumption that multiple threads generally do not modify the same value often enough that a problem will arise. Instead, it efficiently detects whether such a situation has occurred and retries until the update happens successfully. This detection is almost always less costly than acquiring a lock, even in moderate to high contention situations.

How does the Virtual Machine accomplish this? The CAS operation is atomic. Logically, the CAS operation looks something like the following:

```
int variableBeingSet;

void simulateNonBlockingSet(int newValue) {
    int currentValue;
    do {
        currentValue = variableBeingSet
    } while(currentValue != compareAndSwap(currentValue, newValue));
}

int synchronized compareAndSwap(int currentValue, int newValue) {
    if(variableBeingSet == currentValue) {
        variableBeingSet = newValue;
        return currentValue;
    }
    return variableBeingSet;
}
```

When a method attempts to update a shared variable, the CAS operation verifies that the variable getting set still has the last known value. If so, then the variable is changed. If not, then the variable is not set because another thread managed to get in the way. The method making the attempt (using the CAS operation) sees that the change was not made and retries.

Nonthread-Safe Classes

There are some classes that are inherently not thread safe. Here are a few examples:

- `SimpleDateFormat`
- Database Connections
- Containers in `java.util`
- Servlets

Note that some collection classes have individual methods that are thread-safe. However, any operation that involves calling more than one method is not. For example, if you do not want to replace something in a `HashTable` because it is already there, you might write the following code:

```
if(!hashTable.containsKey(someKey)) {
    hashTable.put(someKey, new SomeValue());
}
```

Each individual method is thread-safe. However, another thread might add a value in between the `containsKey` and `put` calls. There are several options to fix this problem.

- Lock the `HashTable` first, and make sure all other users of the `HashTable` do the same—client-based locking:

```
synchronized(map) {
if(!map.conainsKey(key))
    map.put(key,value);
}
```

- Wrap the `HashTable` in its own object and use a different API—server-based locking using an ADAPTER:

```
public class WrappedHashtable<K, V> {
    private Map<K, V> map = new Hashtable<K, V>();

    public synchronized void putIfAbsent(K key, V value) {
        if (map.containsKey(key))
            map.put(key, value);
    }
}
```

- Use the thread-safe collections:

```
ConcurrentHashMap<Integer, String> map = new ConcurrentHashMap<Integer,
String>();
map.putIfAbsent(key, value);
```

The collections in `java.util.concurrent` have operations like `putIfAbsent()` to accommodate such operations.

Dependencies Between Methods Can Break Concurrent Code

Here is a trivial example of a way to introduce dependencies between methods:

```
public class IntegerIterator implements Iterator<Integer>
    private Integer nextValue = 0;

    public synchronized boolean hasNext() {
        return nextValue < 100000;
    }
    public synchronized Integer next() {
        if (nextValue == 100000)
            throw new IteratorPastEndException();
        return nextValue++;
    }
    public synchronized Integer getNextValue() {
        return nextValue;
    }
}
```

Here is some code to use this `IntegerIterator`:

```
IntegerIterator iterator = new IntegerIterator();
while(iterator.hasNext()) {
```

```
        int nextValue = iterator.next();
        // do something with nextValue
    }
```

If one thread executes this code, there will be no problem. But what happens if two threads attempt to share a single instance of `IngeterIterator` with the intent that each thread will process the values it gets, but that each element of the list is processed only once? Most of the time, nothing bad happens; the threads happily share the list, processing the elements they are given by the iterator and stopping when the iterator is complete. However, there is a small chance that, at the end of the iteration, the two threads will interfere with each other and cause one thread to go beyond the end of the iterator and throw an exception.

Here's the problem: Thread 1 asks the question `hasNext()`, which returns `true`. Thread 1 gets preempted and then Thread 2 asks the same question, which is still `true`. Thread 2 then calls `next()`, which returns a value as expected but has a side effect of making `hasNext()` return `false`. Thread 1 starts up again, thinking `hasNext()` is still `true`, and then calls `next()`. Even though the individual methods are synchronized, the client uses ***two*** methods.

This is a real problem and an example of the kinds of problems that crop up in concurrent code. In this particular situation this problem is especially subtle because the only time where this causes a fault is when it happens during the final iteration of the iterator. If the threads happen to break just right, then one of the threads could go beyond the end of the iterator. This is the kind of bug that happens long after a system has been in production, and it is hard to track down.

You have three options:

- Tolerate the failure.
- Solve the problem by changing the client: client-based locking
- Solve the problem by changing the server, which additionally changes the client: server-based locking

Tolerate the Failure

Sometimes you can set things up such that the failure causes no harm. For example, the above client could catch the exception and clean up. Frankly, this is a bit sloppy. It's rather like cleaning up memory leaks by rebooting at midnight.

Client-Based Locking

To make `IntegerIterator` work correctly with multiple threads, change this client (and every other client) as follows:

```
IntegerIterator iterator = new IntegerIterator();

    while (true) {
      int nextValue;
```

```
      synchronized (iterator) {
        if (!iterator.hasNext())
          break;
        nextValue = iterator.next();
      }
      doSometingWith(nextValue);
    }
```

Each client introduces a lock via the `synchronized` keyword. This duplication violates the DRY principle, but it might be necessary if the code uses non-thread-safe third-party tools.

This strategy is risky because all programmers who use the server must remember to lock it before using it and unlock it when done. Many (many!) years ago I worked on a system that employed client-based locking on a shared resource. The resource was used in hundreds of different places throughout the code. One poor programmer forgot to lock the resource in one of those places.

The system was a multi-terminal time-sharing system running accounting software for Local 705 of the trucker's union. The computer was in a raised-floor, environment-controlled room 50 miles north of the Local 705 headquarters. At the headquarters they had dozens of data entry clerks typing union dues postings into the terminals. The terminals were connected to the computer using dedicated phone lines and 600bps half-duplex modems. (This was a very, *very* long time ago.)

About once per day, one of the terminals would "lock up." There was no rhyme or reason to it. The lock up showed no preference for particular terminals or particular times. It was as though there were someone rolling dice choosing the time and terminal to lock up. Sometimes more than one terminal would lock up. Sometimes days would go by without any lock-ups.

At first the only solution was a reboot. But reboots were tough to coordinate. We had to call the headquarters and get everyone to finish what they were doing on all the terminals. Then we could shut down and restart. If someone was doing something important that took an hour or two, the locked up terminal simply had to stay locked up.

After a few weeks of debugging we found that the cause was a ring-buffer counter that had gotten out of sync with its pointer. This buffer controlled output to the terminal. The pointer value indicated that the buffer was empty, but the counter said it was full. Because it was empty, there was nothing to display; but because it was also full, nothing could be added to the buffer to be displayed on the screen.

So we knew why the terminals were locking, but we didn't know why the ring buffer was getting out of sync. So we added a hack to work around the problem. It was possible to read the front panel switches on the computer. (This was a very, very, *very* long time ago.) We wrote a little trap function that detected when one of these switches was thrown and then looked for a ring buffer that was both empty and full. If one was found, it reset that buffer to empty. *Voila!* The locked-up terminal(s) started displaying again.

So now we didn't have to reboot the system when a terminal locked up. The Local would simply call us and tell us we had a lock-up, and then we just walked into the computer room and flicked a switch.

Of course sometimes they worked on the weekends, and we didn't. So we added a function to the scheduler that checked all the ring buffers once per minute and reset any that were both empty and full. This caused the displays to unclog before the Local could even get on the phone.

It was several more weeks of poring over page after page of monolithic assembly language code before we found the culprit. We had done the math and calculated that the frequency of the lock-ups was consistent with a single unprotected use of the ring buffer. So all we had to do was find that one faulty usage. Unfortunately, this was so very long ago that we didn't have search tools or cross references or any other kind of automated help. We simply had to pore over listings.

I learned an important lesson that cold Chicago winter of 1971. Client-based locking really blows.

Server-Based Locking

The duplication can be removed by making the following changes to `IntegerIterator`:

```
public class IntegerIteratorServerLocked {
    private Integer nextValue = 0;
    public synchronized Integer getNextOrNull() {
        if (nextValue < 100000)
            return nextValue++;
        else
            return null;
    }
}
```

And the client code changes as well:

```
while (true) {
    Integer nextValue = iterator.getNextOrNull();
    if (next == null)
        break;
    // do something with nextValue
}
```

In this case we actually change the API of our class to be multithread aware.[3] The client needs to perform a `null` check instead of checking `hasNext()`.

In general you should prefer server-based locking for these reasons:

- It reduces repeated code—Client-based locking forces each client to lock the server properly. By putting the locking code into the server, clients are free to use the object and not worry about writing additional locking code.

3. In fact, the `Iterator` interface is inherently not thread-safe. It was never designed to be used by multiple threads, so this should come as no surprise.

- It allows for better performance—You can swap out a thread-safe server for a non-thread safe one in the case of single-threaded deployment, thereby avoiding all overhead.
- It reduces the possibility of error—All it takes is for one programmer to forget to lock properly.
- It enforces a single policy—The policy is in one place, the server, rather than many places, each client.
- It reduces the scope of the shared variables—The client is not aware of them or how they are locked. All of that is hidden in the server. When things break, the number of places to look is smaller.

What if you do not own the server code?

- Use an ADAPTER to change the API and add locking

```
public class ThreadSafeIntegerIterator {
    private IntegerIterator iterator = new IntegerIterator();

    public synchronized Integer getNextOrNull() {
        if(iterator.hasNext())
            return iterator.next();
        return null;
    }
}
```

- OR better yet, use the thread-safe collections with extended interfaces

Increasing Throughput

Let's assume that we want to go out on the net and read the contents of a set of pages from a list of URLs. As each page is read, we will parse it to accumulate some statistics. Once all the pages are read, we will print a summary report.

The following class returns the contents of one page, given a URL.

```
public class PageReader {
  //...
  public String getPageFor(String url) {
    HttpMethod method = new GetMethod(url);

    try {
      httpClient.executeMethod(method);
      String response = method.getResponseBodyAsString();
      return response;
    } catch (Exception e) {
      handle(e);
    } finally {
      method.releaseConnection();
    }
  }
}
```

The next class is the iterator that provides the contents of the pages based on an iterator of URLs:

```
public class PageIterator {
  private PageReader reader;
  private URLIterator urls;

  public PageIterator(PageReader reader, URLIterator urls) {
    this.urls = urls;
    this.reader = reader;
  }

  public synchronized String getNextPageOrNull() {
    if (urls.hasNext())
      getPageFor(urls.next());
    else
      return null;
  }

  public String getPageFor(String url) {
    return reader.getPageFor(url);
  }
}
```

An instance of the `PageIterator` can be shared between many different threads, each one using it's own instance of the `PageReader` to read and parse the pages it gets from the iterator.

Notice that we've kept the `synchronized` block very small. It contains just the critical section deep inside the `PageIterator`. It is always better to synchronize as little as possible as opposed to synchronizing as much as possible.

Single-Thread Calculation of Throughput

Now lets do some simple calculations. For the purpose of argument, assume the following:

- I/O time to retrieve a page (average): 1 second
- Processing time to parse page (average): .5 seconds
- I/O requires 0 percent of the CPU while processing requires 100 percent.

For *N* pages being processed by a single thread, the total execution time is 1.5 seconds * *N*. Figure A-1 shows a snapshot of 13 pages or about 19.5 seconds.

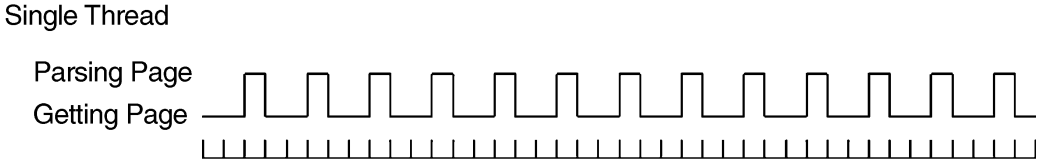


Figure A-1
Single thread

Multithread Calculation of Throughput

If it is possible to retrieve pages in any order and process the pages independently, then it is possible to use multiple threads to increase throughput. What happens if we use three threads? How many pages can we acquire in the same time?

As you can see in Figure A-2, the multithreaded solution allows the process-bound parsing of the pages to overlap with the I/O-bound reading of the pages. In an idealized world this means that the processor is fully utilized. Each one-second page read is overlapped with two parses. Thus, we can process two pages per second, which is three times the throughput of the single-threaded solution.

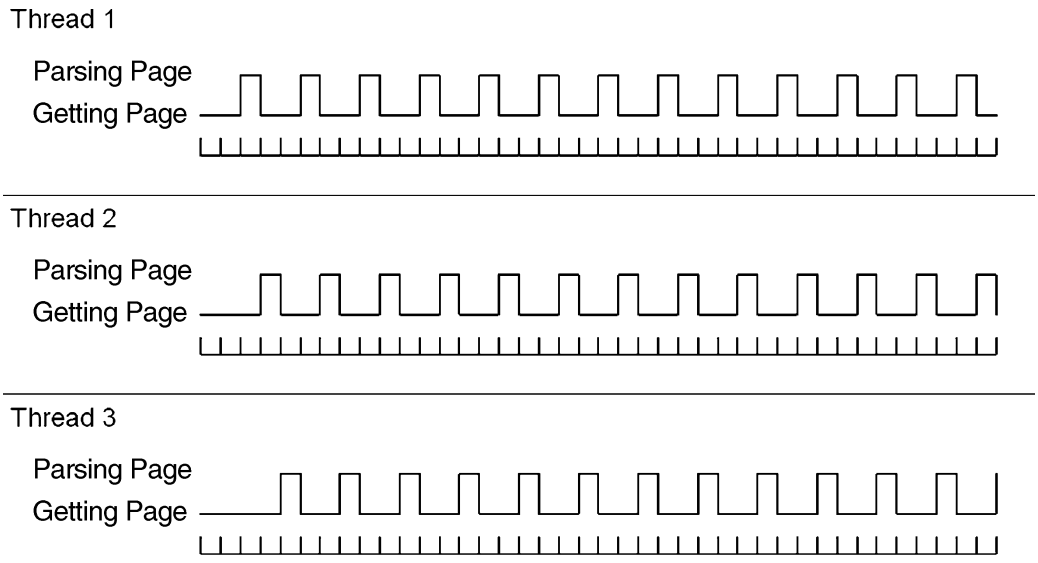


Figure A-2
Three concurrent threads

Deadlock

Imagine a Web application with two shared resource pools of some finite size:

- A pool of database connections for local work in process storage
- A pool of MQ connections to a master repository

Assume there are two operations in this application, create and update:

- Create—Acquire connection to master repository and database. Talk to service master repository and then store work in local work in process database.

- Update—Acquire connection to database and then master repository. Read from work in process database and then send to the master repository

What happens when there are more users than the pool sizes? Consider each pool has a size of ten.

- Ten users attempt to use create, so all ten database connections are acquired, and each thread is interrupted after acquiring a database connection but before acquiring a connection to the master repository.
- Ten users attempt to use update, so all ten master repository connections are acquired, and each thread is interrupted after acquiring the master repository but before acquiring a database connection.
- Now the ten "create" threads must wait to acquire a master repository connection, but the ten "update" threads must wait to acquire a database connection.
- Deadlock. The system never recovers.

This might sound like an unlikely situation, but who wants a system that freezes solid every other week? Who wants to debug a system with symptoms that are so difficult to reproduce? This is the kind of problem that happens in the field, then takes weeks to solve.

A typical "solution" is to introduce debugging statements to find out what is happening. Of course, the debug statements change the code enough so that the deadlock happens in a different situation and takes months to again occur.[4]

To really solve the problem of deadlock, we need to understand what causes it. There are four conditions required for deadlock to occur:

- Mutual exclusion
- Lock & wait
- No preemption
- Circular wait

Mutual Exclusion

Mutual exclusion occurs when multiple threads need to use the same resources and those resources

- Cannot be used by multiple threads at the same time.
- Are limited in number.

A common example of such a resource is a database connection, a file open for write, a record lock, or a semaphore.

---

4. For example, someone adds some debugging output and the problem "disappears." The debugging code "fixes" the problem so it remains in the system.

Lock & Wait

Once a thread acquires a resource, it will not release the resource until it has acquired all of the other resources it requires and has completed its work.

No Preemption

One thread cannot take resources away from another thread. Once a thread holds a resource, the only way for another thread to get it is for the holding thread to release it.

Circular Wait

This is also referred to as the deadly embrace. Imagine two threads, T1 and T2, and two resources, R1 and R2. T1 has R1, T2 has R2. T1 also requires R2, and T2 also requires R1. This gives something like Figure A-3:

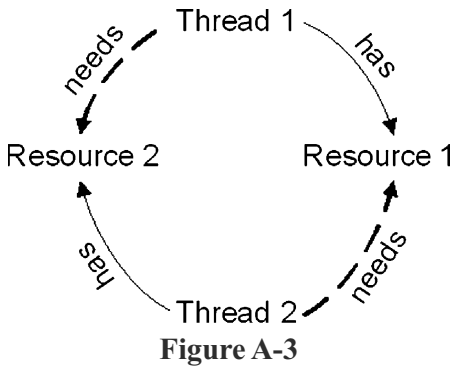


Figure A-3

All four of these conditions must hold for deadlock to be possible. Break any one of these conditions and deadlock is not possible.

Breaking Mutual Exclusion

One strategy for avoiding deadlock is to sidestep the mutual exclusion condition. You might be able to do this by

- Using resources that allow simultaneous use, for example, `AtomicInteger`.
- Increasing the number of resources such that it equals or exceeds the number of competing threads.
- Checking that all your resources are free before seizing any.

Unfortunately, most resources are limited in number and don't allow simultaneous use. And it's not uncommon for the identity of the second resource to be predicated on the results of operating on the first. But don't be discouraged; there are three conditions left.

Breaking Lock & Wait

You can also eliminate deadlock if you refuse to wait. Check each resource before you seize it, and release all resources and start over if you run into one that's busy.

This approach introduces several potential problems:

- Starvation—One thread keeps being unable to acquire the resources it needs (maybe it has a unique combination of resources that seldom all become available).
- Livelock—Several threads might get into lockstep and all acquire one resource and then release one resource, over and over again. This is especially likely with simplistic CPU scheduling algorithms (think embedded devices or simplistic hand-written thread balancing algorithms).

Both of these can cause poor throughput. The first results in low CPU utilization, whereas the second results in high and useless CPU utilization.

As inefficient as this strategy sounds, it's better than nothing. It has the benefit that it can almost always be implemented if all else fails.

Breaking Preemption

Another strategy for avoiding deadlock is to allow threads to take resources away from other threads. This is usually done through a simple request mechanism. When a thread discovers that a resource is busy, it asks the owner to release it. If the owner is also waiting for some other resource, it releases them all and starts over.

This is similar to the previous approach but has the benefit that a thread is allowed to wait for a resource. This decreases the number of startovers. Be warned, however, that managing all those requests can be tricky.

Breaking Circular Wait

This is the most common approach to preventing deadlock. For most systems it requires no more than a simple convention agreed to by all parties.

In the example above with Thread 1 wanting both Resource 1 and Resource 2 and Thread 2 wanting both Resource 2 and then Resource 1, simply forcing both Thread 1 and Thread 2 to allocate resources in the same order makes circular wait impossible.

More generally, if all threads can agree on a global ordering of resources and if they all allocate resources in that order, then deadlock is impossible. Like all the other strategies, this can cause problems:

- The order of acquisition might not correspond to the order of use; thus a resource acquired at the start might not be used until the end. This can cause resources to be locked longer than strictly necessary.

- Sometimes you cannot impose an order on the acquisition of resources. If the ID of the second resource comes from an operation performed on the first, then ordering is not feasible.

So there are many ways to avoid deadlock. Some lead to starvation, whereas others make heavy use of the CPU and reduce responsiveness. TANSTAAFL![5]

Isolating the thread-related part of your solution to allow for tuning and experimentation is a powerful way to gain the insights needed to determine the best strategies.

Testing Multithreaded Code

How can we write a test to demonstrate the following code is broken?

```
01: public class ClassWithThreadingProblem {
02:     int nextId;
03:
04:     public int takeNextId() {
05:         return nextId++;
06:     }
07:}
```

Here's a description of a test that will prove the code is broken:

- Remember the current value of `nextId`.
- Create two threads, both of which call `takeNextId()` once.
- Verify that `nextId` is two more than what we started with.
- Run this until we demonstrate that `nextId` was only incremented by one instead of two.

Listing A-2 shows such a test:

Listing A-2
`ClassWithThreadingProblemTest.java`

```
01: package example;
02:
03: import static org.junit.Assert.fail;
04:
05: import org.junit.Test;
06:
07: public class ClassWithThreadingProblemTest {
08:     @Test
09:     public void twoThreadsShouldFailEventually() throws Exception {
10:         final ClassWithThreadingProblem classWithThreadingProblem
                = new ClassWithThreadingProblem();
11:
```

5. There ain't no such thing as a free lunch.

Listing A-2 (continued)
`ClassWithThreadingProblemTest.java`

```
12:         Runnable runnable = new Runnable() {
13:             public void run() {
14:                 classWithThreadingProblem.takeNextId();
15:             }
16:         };
17:
18:         for (int i = 0; i < 50000; ++i) {
19:             int startingId = classWithThreadingProblem.lastId;
20:             int expectedResult = 2 + startingId;
21:
22:             Thread t1 = new Thread(runnable);
23:             Thread t2 = new Thread(runnable);
24:             t1.start();
25:             t2.start();
26:             t1.join();
27:             t2.join();
28:
29:             int endingId = classWithThreadingProblem.lastId;
30:
31:             if (endingId != expectedResult)
32:                 return;
33:         }
34:
35:         fail("Should have exposed a threading issue but it did not.");
36:     }
37: }
```

Line	Description
10	Create a single instance of `ClassWithThreadingProblem`. Note, we must use the final keyword because we use it below in an anonymous inner class.
12–16	Create an anonymous inner class that uses the single instance of `ClassWithThreadingProblem`.
18	Run this code "enough" times to demonstrate that the code failed, but not so much that the test "takes too long." This is a balancing act; we don't want to wait too long to demonstrate failure. Picking this number is hard—although later we'll see that we can greatly reduce this number.
19	Remember the starting value. This test is trying to prove that the code in `ClassWithThreadingProblem` is broken. If this test passes, it proved that the code was broken. If this test fails, the test was unable to prove that the code is broken.
20	We expect the final value to be two more than the current value.
22–23	Create two threads, both of which use the object we created in lines 12–16. This gives us the potential of two threads trying to use our single instance of `ClassWithThreadingProblem` and interfering with each other.

Line	Description
24-25	Make our two threads eligible to run.
26-27	Wait for both threads to finish before we check the results.
29	Record the actual final value.
31-32	Did our `endingId` differ from what we expected? If so, return end the test—we've proven that the code is broken. If not, try again.
35	If we got to here, our test was unable to prove the production code was broken in a "reasonable" amount of time; our code has failed. Either the code is not broken or we didn't run enough iterations to get the failure condition to occur.

This test certainly sets up the conditions for a concurrent update problem. However, the problem occurs so infrequently that the vast majority of times this test won't detect it.

Indeed, to truly detect the problem we need to set the number of iterations to over one million. Even then, in ten executions with a loop count of 1,000,000, the problem occurred only once. That means we probably ought to set the iteration count to well over one hundred million to get reliable failures. How long are we prepared to wait?

Even if we tuned the test to get reliable failures on one machine, we'll probably have to retune the test with different values to demonstrate the failure on another machine, operating system, or version of the JVM.

And this is a *simple* problem. If we cannot demonstrate broken code easily with this problem, how will we ever detect truly complex problems?

So what approaches can we take to demonstrate this simple failure? And, more importantly, how can we write tests that will demonstrate failures in more complex code? How will we be able to discover if our code has failures when we do not know where to look?

Here are a few ideas:

- **Monte Carlo Testing.** Make tests flexible, so they can be tuned. Then run the test over and over—say on a test server—randomly changing the tuning values. If the tests ever fail, the code is broken. Make sure to start writing those tests early so a continuous integration server starts running them soon. By the way, make sure you carefully log the conditions under which the test failed.
- Run the test on every one of the target deployment platforms. Repeatedly. Continuously. The longer the tests run without failure, the more likely that
  - The production code is correct or
  - The tests aren't adequate to expose problems.
- Run the tests on a machine with varying loads. If you can simulate loads close to a production environment, do so.

Yet, even if you do all of these things, you still don't stand a very good chance of finding threading problems with your code. The most insidious problems are the ones that have such a small cross section that they only occur once in a billion opportunities. Such problems are the terror of complex systems.

Tool Support for Testing Thread-Based Code

IBM has created a tool called ConTest.[6] It instruments classes to make it more likely that non-thread-safe code fails.

We do not have any direct relationship with IBM or the team that developed ConTest. A colleague of ours pointed us to it. We noticed vast improvement in our ability to find threading issues after a few minutes of using it.

Here's an outline of how to use ConTest:

- Write tests and production code, making sure there are tests specifically designed to simulate multiple users under varying loads, as mentioned above.
- Instrument test and production code with ConTest.
- Run the tests.

When we instrumented code with ConTest, our success rate went from roughly one failure in ten million iterations to roughly one failure in *thirty* iterations. Here are the loop values for several runs of the test after instrumentation: 13, 23, 0, 54, 16, 14, 6, 69, 107, 49, 2. So clearly the instrumented classes failed much earlier and with much greater reliability.

Conclusion

This chapter has been a very brief sojourn through the large and treacherous territory of concurrent programming. We barely scratched the surface. Our emphasis here was on disciplines to help keep concurrent code clean, but there is much more you should learn if you are going to be writing concurrent systems. We recommend you start with Doug Lea's wonderful book *Concurrent Programming in Java: Design Principles and Patterns.*[7]

In this chapter we talked about concurrent update, and the disciplines of clean synchronization and locking that can prevent it. We talked about how threads can enhance the throughput of an I/O-bound system and showed the clean techniques for achieving such improvements. We talked about deadlock and the disciplines for preventing it in a clean

6. http://www.haifa.ibm.com/projects/verification/contest/index.html
7. See [Lea99] p. 191.

way. Finally, we talked about strategies for exposing concurrent problems by instrumenting your code.

Tutorial: Full Code Examples

Client/Server Nonthreaded

Listing A-3
`Server.java`

```
package com.objectmentor.clientserver.nonthreaded;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

import common.MessageUtils;

public class Server implements Runnable {
    ServerSocket serverSocket;
    volatile boolean keepProcessing = true;

    public Server(int port, int millisecondsTimeout) throws IOException {
        serverSocket = new ServerSocket(port);
        serverSocket.setSoTimeout(millisecondsTimeout);
    }

    public void run() {
        System.out.printf("Server Starting\n");

        while (keepProcessing) {
            try {
                System.out.printf("accepting client\n");
                Socket socket = serverSocket.accept();
                System.out.printf("got client\n");
                process(socket);
            } catch (Exception e) {
                handle(e);
            }
        }
    }

    private void handle(Exception e) {
        if (!(e instanceof SocketException)) {
            e.printStackTrace();
        }
    }

    public void stopProcessing() {
        keepProcessing = false;
        closeIgnoringException(serverSocket);
    }
```

Listing A-3 (continued)
Server.java

```
    void process(Socket socket) {
        if (socket == null)
            return;

        try {
            System.out.printf("Server: getting message\n");
            String message = MessageUtils.getMessage(socket);
            System.out.printf("Server: got message: %s\n", message);
            Thread.sleep(1000);
            System.out.printf("Server: sending reply: %s\n", message);
            MessageUtils.sendMessage(socket, "Processed: " + message);
            System.out.printf("Server: sent\n");
            closeIgnoringException(socket);
        } catch (Exception e) {
            e.printStackTrace();
        }

    }

    private void closeIgnoringException(Socket socket) {
        if (socket != null)
            try {
                socket.close();
            } catch (IOException ignore) {
            }
    }

    private void closeIgnoringException(ServerSocket serverSocket) {
        if (serverSocket != null)
            try {
                serverSocket.close();
            } catch (IOException ignore) {
            }
    }
}
```

Listing A-4
ClientTest.java

```
package com.objectmentor.clientserver.nonthreaded;

import java.io.IOException;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketException;

import common.MessageUtils;

public class Server implements Runnable {
    ServerSocket serverSocket;
    volatile boolean keepProcessing = true;
```

Listing A-4 (continued)
ClientTest.java

```
    public Server(int port, int millisecondsTimeout) throws IOException {
        serverSocket = new ServerSocket(port);
        serverSocket.setSoTimeout(millisecondsTimeout);
    }

    public void run() {
        System.out.printf("Server Starting\n");

        while (keepProcessing) {
            try {
                System.out.printf("accepting client\n");
                Socket socket = serverSocket.accept();
                System.out.printf("got client\n");
                process(socket);
            } catch (Exception e) {
                handle(e);
            }
        }
    }

    private void handle(Exception e) {
        if (!(e instanceof SocketException)) {
            e.printStackTrace();
        }
    }

    public void stopProcessing() {
        keepProcessing = false;
        closeIgnoringException(serverSocket);
    }

    void process(Socket socket) {
        if (socket == null)
            return;

        try {
            System.out.printf("Server: getting message\n");
            String message = MessageUtils.getMessage(socket);
            System.out.printf("Server: got message: %s\n", message);
            Thread.sleep(1000);
            System.out.printf("Server: sending reply: %s\n", message);
            MessageUtils.sendMessage(socket, "Processed: " + message);
            System.out.printf("Server: sent\n");
            closeIgnoringException(socket);
        } catch (Exception e) {
            e.printStackTrace();
        }

    }

    private void closeIgnoringException(Socket socket) {
        if (socket != null)
            try {
                socket.close();
```

Listing A-4 (continued)
ClientTest.java

```
            } catch (IOException ignore) {
            }
    }

    private void closeIgnoringException(ServerSocket serverSocket) {
        if (serverSocket != null)
            try {
                serverSocket.close();
            } catch (IOException ignore) {
            }
    }
}
```

Listing A-5
MessageUtils.java

```
package common;

import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import java.net.Socket;

public class MessageUtils {
    public static void sendMessage(Socket socket, String message)
            throws IOException {
        OutputStream stream = socket.getOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(stream);
        oos.writeUTF(message);
        oos.flush();
    }

    public static String getMessage(Socket socket) throws IOException {
        InputStream stream = socket.getInputStream();
        ObjectInputStream ois = new ObjectInputStream(stream);
        return ois.readUTF();
    }
}
```

Client/Server Using Threads

Changing the server to use threads simply requires a change to the process message (new lines are emphasized to stand out):

```
void process(final Socket socket) {
    if (socket == null)
        return;

    Runnable clientHandler = new Runnable() {
        public void run() {
```

```
                try {
                    System.out.printf("Server: getting message\n");
                    String message = MessageUtils.getMessage(socket);
                    System.out.printf("Server: got message: %s\n", message);
                    Thread.sleep(1000);
                    System.out.printf("Server: sending reply: %s\n", message);
                    MessageUtils.sendMessage(socket, "Processed: " + message);
                    System.out.printf("Server: sent\n");
                    closeIgnoringException(socket);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        };

        Thread clientConnection = new Thread(clientHandler);
        clientConnection.start();
    }
```

Appendix B

org.jfree.date.SerialDate

Listing B-1

SerialDate.Java

```
/* ===========================================================================
 * JCommon : a free general purpose class library for the Java(tm) platform
 * ===========================================================================
 *
 * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 *
 * Project Info:  http://www.jfree.org/jcommon/index.html
 *
 * This library is free software; you can redistribute it and/or modify it
 * under the terms of the GNU Lesser General Public License as published by
 * the Free Software Foundation; either version 2.1 of the License, or
 * (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
 * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
 * License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
 * USA.
 *
 * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
 * in the United States and other countries.]
 *
 * ---------------
 * SerialDate.java
 * ---------------
 * (C) Copyright 2001-2005, by Object Refinery Limited.
 *
 * Original Author:  David Gilbert (for Object Refinery Limited);
 * Contributor(s):   -;
 *
 * $Id: SerialDate.java,v 1.7 2005/11/03 09:25:17 mungady Exp $
 *
 * Changes (from 11-Oct-2001)
```

Listing B-1 (continued)
`SerialDate.Java`

```
 38  * -------------------------
 39  * 11-Oct-2001 : Re-organised the class and moved it to new package
 40  *               com.jrefinery.date (DG);
 41  * 05-Nov-2001 : Added a getDescription() method, and eliminated NotableDate
 42  *               class (DG);
 43  * 12-Nov-2001 : IBD requires setDescription() method, now that NotableDate
 44  *               class is gone (DG);  Changed getPreviousDayOfWeek(),
 45  *               getFollowingDayOfWeek() and getNearestDayOfWeek() to correct
 46  *               bugs (DG);
 47  * 05-Dec-2001 : Fixed bug in SpreadsheetDate class (DG);
 48  * 29-May-2002 : Moved the month constants into a separate interface
 49  *               (MonthConstants) (DG);
 50  * 27-Aug-2002 : Fixed bug in addMonths() method, thanks to N???levka Petr (DG);
 51  * 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
 52  * 13-Mar-2003 : Implemented Serializable (DG);
 53  * 29-May-2003 : Fixed bug in addMonths method (DG);
 54  * 04-Sep-2003 : Implemented Comparable.  Updated the isInRange javadocs (DG);
 55  * 05-Jan-2005 : Fixed bug in addYears() method (1096282) (DG);
 56  *
 57  */
 58
 59 package org.jfree.date;
 60
 61 import java.io.Serializable;
 62 import java.text.DateFormatSymbols;
 63 import java.text.SimpleDateFormat;
 64 import java.util.Calendar;
 65 import java.util.GregorianCalendar;
 66
 67 /**
 68  *  An abstract class that defines our requirements for manipulating dates,
 69  *  without tying down a particular implementation.
 70  *  <P>
 71  *  Requirement 1 : match at least what Excel does for dates;
 72  *  Requirement 2 : class is immutable;
 73  *  <P>
 74  *  Why not just use java.util.Date?  We will, when it makes sense.  At times,
 75  *  java.util.Date can be *too* precise - it represents an instant in time,
 76  *  accurate to 1/1000th of a second (with the date itself depending on the
 77  *  time-zone).  Sometimes we just want to represent a particular day (e.g. 21
 78  *  January 2015) without concerning ourselves about the time of day, or the
 79  *  time-zone, or anything else.  That's what we've defined SerialDate for.
 80  *  <P>
 81  *  You can call getInstance() to get a concrete subclass of SerialDate,
 82  *  without worrying about the exact implementation.
 83  *
 84  * @author David Gilbert
 85  */
 86 public abstract class SerialDate implements Comparable,
 87                                              Serializable,
 88                                              MonthConstants {
 89
 90     /** For serialization. */
 91     private static final long serialVersionUID = -293716040467423637L;
 92
 93     /** Date format symbols. */
 94     public static final DateFormatSymbols
 95         DATE_FORMAT_SYMBOLS = new SimpleDateFormat().getDateFormatSymbols();
 96
 97     /** The serial number for 1 January 1900. */
 98     public static final int SERIAL_LOWER_BOUND = 2;
 99
100     /** The serial number for 31 December 9999. */
101     public static final int SERIAL_UPPER_BOUND = 2958465;
102
```

Listing B-1 (continued)
`SerialDate.Java`

```
103     /** The lowest year value supported by this date format. */
104     public static final int MINIMUM_YEAR_SUPPORTED = 1900;
105
106     /** The highest year value supported by this date format. */
107     public static final int MAXIMUM_YEAR_SUPPORTED = 9999;
108
109     /** Useful constant for Monday. Equivalent to java.util.Calendar.MONDAY. */
110     public static final int MONDAY = Calendar.MONDAY;
111
112     /**
113      * Useful constant for Tuesday. Equivalent to java.util.Calendar.TUESDAY.
114      */
115     public static final int TUESDAY = Calendar.TUESDAY;
116
117     /**
118      * Useful constant for Wednesday. Equivalent to
119      * java.util.Calendar.WEDNESDAY.
120      */
121     public static final int WEDNESDAY = Calendar.WEDNESDAY;
122
123     /**
124      * Useful constant for Thrusday. Equivalent to java.util.Calendar.THURSDAY.
125      */
126     public static final int THURSDAY = Calendar.THURSDAY;
127
128     /** Useful constant for Friday. Equivalent to java.util.Calendar.FRIDAY. */
129     public static final int FRIDAY = Calendar.FRIDAY;
130
131     /**
132      * Useful constant for Saturday. Equivalent to java.util.Calendar.SATURDAY.
133      */
134     public static final int SATURDAY = Calendar.SATURDAY;
135
136     /** Useful constant for Sunday. Equivalent to java.util.Calendar.SUNDAY. */
137     public static final int SUNDAY = Calendar.SUNDAY;
138
139     /** The number of days in each month in non leap years. */
140     static final int[] LAST_DAY_OF_MONTH =
141         {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
142
143     /** The number of days in a (non-leap) year up to the end of each month. */
144     static final int[] AGGREGATE_DAYS_TO_END_OF_MONTH =
145         {0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
146
147     /** The number of days in a year up to the end of the preceding month. */
148     static final int[] AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
149         {0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
150
151     /** The number of days in a leap year up to the end of each month. */
152     static final int[] LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_MONTH =
153         {0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
154
155     /**
156      * The number of days in a leap year up to the end of the preceding month.
157      */
158     static final int[]
159         LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
160             {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
161
162     /** A useful constant for referring to the first week in a month. */
163     public static final int FIRST_WEEK_IN_MONTH = 1;
164
```

Listing B-1 (continued)
SerialDate.Java

```
165     /** A useful constant for referring to the second week in a month. */
166     public static final int SECOND_WEEK_IN_MONTH = 2;
167
168     /** A useful constant for referring to the third week in a month. */
169     public static final int THIRD_WEEK_IN_MONTH = 3;
170
171     /** A useful constant for referring to the fourth week in a month. */
172     public static final int FOURTH_WEEK_IN_MONTH = 4;
173
174     /** A useful constant for referring to the last week in a month. */
175     public static final int LAST_WEEK_IN_MONTH = 0;
176
177     /** Useful range constant. */
178     public static final int INCLUDE_NONE = 0;
179
180     /** Useful range constant. */
181     public static final int INCLUDE_FIRST = 1;
182
183     /** Useful range constant. */
184     public static final int INCLUDE_SECOND = 2;
185
186     /** Useful range constant. */
187     public static final int INCLUDE_BOTH = 3;
188
189     /**
190      * Useful constant for specifying a day of the week relative to a fixed
191      * date.
192      */
193     public static final int PRECEDING = -1;
194
195     /**
196      * Useful constant for specifying a day of the week relative to a fixed
197      * date.
198      */
199     public static final int NEAREST = 0;
200
201     /**
202      * Useful constant for specifying a day of the week relative to a fixed
203      * date.
204      */
205     public static final int FOLLOWING = 1;
206
207     /** A description for the date. */
208     private String description;
209
210     /**
211      * Default constructor.
212      */
213     protected SerialDate() {
214     }
215
216     /**
217      * Returns <code>true</code> if the supplied integer code represents a
218      * valid day-of-the-week, and <code>false</code> otherwise.
219      *
220      * @param code  the code being checked for validity.
221      *
222      * @return <code>true</code> if the supplied integer code represents a
223      *         valid day-of-the-week, and <code>false</code> otherwise.
224      */
225     public static boolean isValidWeekdayCode(final int code) {
226
```

Listing B-1 (continued)
`SerialDate.Java`

```
227         switch(code) {
228             case SUNDAY:
229             case MONDAY:
230             case TUESDAY:
231             case WEDNESDAY:
232             case THURSDAY:
233             case FRIDAY:
234             case SATURDAY:
235                 return true;
236             default:
237                 return false;
238         }
239
240     }
241
242     /**
243      * Converts the supplied string to a day of the week.
244      *
245      * @param s  a string representing the day of the week.
246      *
247      * @return <code>-1</code> if the string is not convertable, the day of
248      *         the week otherwise.
249      */
250     public static int stringToWeekdayCode(String s) {
251
252         final String[] shortWeekdayNames
253             = DATE_FORMAT_SYMBOLS.getShortWeekdays();
254         final String[] weekDayNames = DATE_FORMAT_SYMBOLS.getWeekdays();
255
256         int result = -1;
257         s = s.trim();
258         for (int i = 0; i < weekDayNames.length; i++) {
259             if (s.equals(shortWeekdayNames[i])) {
260                 result = i;
261                 break;
262             }
263             if (s.equals(weekDayNames[i])) {
264                 result = i;
265                 break;
266             }
267         }
268         return result;
269
270     }
271
272     /**
273      * Returns a string representing the supplied day-of-the-week.
274      * <P>
275      * Need to find a better approach.
276      *
277      * @param weekday  the day of the week.
278      *
279      * @return a string representing the supplied day-of-the-week.
280      */
281     public static String weekdayCodeToString(final int weekday) {
282
283         final String[] weekdays = DATE_FORMAT_SYMBOLS.getWeekdays();
284         return weekdays[weekday];
285
286     }
287
288     /**
```

Listing B-1 (continued)
SerialDate.Java

```
289      * Returns an array of month names.
290      *
291      * @return an array of month names.
292      */
293     public static String[] getMonths() {
294
295         return getMonths(false);
296
297     }
298
299     /**
300      * Returns an array of month names.
301      *
302      * @param shortened  a flag indicating that shortened month names should
303      *                   be returned.
304      *
305      * @return an array of month names.
306      */
307     public static String[] getMonths(final boolean shortened) {
308
309         if (shortened) {
310             return DATE_FORMAT_SYMBOLS.getShortMonths();
311         }
312         else {
313             return DATE_FORMAT_SYMBOLS.getMonths();
314         }
315
316     }
317
318     /**
319      * Returns true if the supplied integer code represents a valid month.
320      *
321      * @param code  the code being checked for validity.
322      *
323      * @return <code>true</code> if the supplied integer code represents a
324      *         valid month.
325      */
326     public static boolean isValidMonthCode(final int code) {
327
328         switch(code) {
329             case JANUARY:
330             case FEBRUARY:
331             case MARCH:
332             case APRIL:
333             case MAY:
334             case JUNE:
335             case JULY:
336             case AUGUST:
337             case SEPTEMBER:
338             case OCTOBER:
339             case NOVEMBER:
340             case DECEMBER:
341                 return true;
342             default:
343                 return false;
344         }
345
346     }
347
348     /**
349      * Returns the quarter for the specified month.
350      *
```

Listing B-1 (continued)
`SerialDate.Java`

```
351      * @param code  the month code (1-12).
352      *
353      * @return the quarter that the month belongs to.
354      * @throws java.lang.IllegalArgumentException
355      */
356     public static int monthCodeToQuarter(final int code) {
357
358         switch(code) {
359             case JANUARY:
360             case FEBRUARY:
361             case MARCH: return 1;
362             case APRIL:
363             case MAY:
364             case JUNE: return 2;
365             case JULY:
366             case AUGUST:
367             case SEPTEMBER: return 3;
368             case OCTOBER:
369             case NOVEMBER:
370             case DECEMBER: return 4;
371             default: throw new IllegalArgumentException(
372                 "SerialDate.monthCodeToQuarter: invalid month code.");
373         }
374
375     }
376
377     /**
378      * Returns a string representing the supplied month.
379      * <P>
380      * The string returned is the long form of the month name taken from the
381      * default locale.
382      *
383      * @param month  the month.
384      *
385      * @return a string representing the supplied month.
386      */
387     public static String monthCodeToString(final int month) {
388
389         return monthCodeToString(month, false);
390
391     }
392
393     /**
394      * Returns a string representing the supplied month.
395      * <P>
396      * The string returned is the long or short form of the month name taken
397      * from the default locale.
398      *
399      * @param month  the month.
400      * @param shortened  if <code>true</code> return the abbreviation of the
401      *                   month.
402      *
403      * @return a string representing the supplied month.
404      * @throws java.lang.IllegalArgumentException
405      */
406     public static String monthCodeToString(final int month,
407                                            final boolean shortened) {
408
409         // check arguments...
410         if (!isValidMonthCode(month)) {
411             throw new IllegalArgumentException(
412                 "SerialDate.monthCodeToString: month outside valid range.");
```

Listing B-1 (continued)
`SerialDate.Java`

```
413         }
414
415         final String[] months;
416
417         if (shortened) {
418             months = DATE_FORMAT_SYMBOLS.getShortMonths();
419         }
420         else {
421             months = DATE_FORMAT_SYMBOLS.getMonths();
422         }
423
424         return months[month - 1];
425
426     }
427
428     /**
429      * Converts a string to a month code.
430      * <P>
431      * This method will return one of the constants JANUARY, FEBRUARY, ...,
432      * DECEMBER that corresponds to the string.  If the string is not
433      * recognised, this method returns -1.
434      *
435      * @param s  the string to parse.
436      *
437      * @return <code>-1</code> if the string is not parseable, the month of the
438      *         year otherwise.
439      */
440     public static int stringToMonthCode(String s) {
441
442         final String[] shortMonthNames = DATE_FORMAT_SYMBOLS.getShortMonths();
443         final String[] monthNames = DATE_FORMAT_SYMBOLS.getMonths();
444
445         int result = -1;
446         s = s.trim();
447
448         // first try parsing the string as an integer (1-12)...
449         try {
450             result = Integer.parseInt(s);
451         }
452         catch (NumberFormatException e) {
453             // suppress
454         }
455
456         // now search through the month names...
457         if ((result < 1) || (result > 12)) {
458             for (int i = 0; i < monthNames.length; i++) {
459                 if (s.equals(shortMonthNames[i])) {
460                     result = i + 1;
461                     break;
462                 }
463                 if (s.equals(monthNames[i])) {
464                     result = i + 1;
465                     break;
466                 }
467             }
468         }
469
470         return result;
471
472     }
473
474     /**
```

Listing B-1 (continued)
`SerialDate.Java`

```
475      * Returns true if the supplied integer code represents a valid
476      * week-in-the-month, and false otherwise.
477      *
478      * @param code  the code being checked for validity.
479      * @return <code>true</code> if the supplied integer code represents a
480      *         valid week-in-the-month.
481      */
482     public static boolean isValidWeekInMonthCode(final int code) {
483
484         switch(code) {
485             case FIRST_WEEK_IN_MONTH:
486             case SECOND_WEEK_IN_MONTH:
487             case THIRD_WEEK_IN_MONTH:
488             case FOURTH_WEEK_IN_MONTH:
489             case LAST_WEEK_IN_MONTH: return true;
490             default: return false;
491         }
492
493     }
494
495     /**
496      * Determines whether or not the specified year is a leap year.
497      *
498      * @param yyyy  the year (in the range 1900 to 9999).
499      *
500      * @return <code>true</code> if the specified year is a leap year.
501      */
502     public static boolean isLeapYear(final int yyyy) {
503
504         if ((yyyy % 4) != 0) {
505             return false;
506         }
507         else if ((yyyy % 400) == 0) {
508             return true;
509         }
510         else if ((yyyy % 100) == 0) {
511             return false;
512         }
513         else {
514             return true;
515         }
516
517     }
518
519     /**
520      * Returns the number of leap years from 1900 to the specified year
521      * INCLUSIVE.
522      * <P>
523      * Note that 1900 is not a leap year.
524      *
525      * @param yyyy  the year (in the range 1900 to 9999).
526      *
527      * @return the number of leap years from 1900 to the specified year.
528      */
529     public static int leapYearCount(final int yyyy) {
530
531         final int leap4 = (yyyy - 1896) / 4;
532         final int leap100 = (yyyy - 1800) / 100;
533         final int leap400 = (yyyy - 1600) / 400;
534         return leap4 - leap100 + leap400;
535
536     }
```

Listing B-1 (continued)
SerialDate.Java

```
537
538     /**
539      * Returns the number of the last day of the month, taking into account
540      * leap years.
541      *
542      * @param month  the month.
543      * @param yyyy  the year (in the range 1900 to 9999).
544      *
545      * @return the number of the last day of the month.
546      */
547     public static int lastDayOfMonth(final int month, final int yyyy) {
548
549         final int result = LAST_DAY_OF_MONTH[month];
550         if (month != FEBRUARY) {
551             return result;
552         }
553         else if (isLeapYear(yyyy)) {
554             return result + 1;
555         }
556         else {
557             return result;
558         }
559
560     }
561
562     /**
563      * Creates a new date by adding the specified number of days to the base
564      * date.
565      *
566      * @param days  the number of days to add (can be negative).
567      * @param base  the base date.
568      *
569      * @return a new date.
570      */
571     public static SerialDate addDays(final int days, final SerialDate base) {
572
573         final int serialDayNumber = base.toSerial() + days;
574         return SerialDate.createInstance(serialDayNumber);
575
576     }
577
578     /**
579      * Creates a new date by adding the specified number of months to the base
580      * date.
581      * <P>
582      * If the base date is close to the end of the month, the day on the result
583      * may be adjusted slightly:  31 May + 1 month = 30 June.
584      *
585      * @param months  the number of months to add (can be negative).
586      * @param base  the base date.
587      *
588      * @return a new date.
589      */
590     public static SerialDate addMonths(final int months,
591                                        final SerialDate base) {
592
593         final int yy = (12 * base.getYYYY() + base.getMonth() + months - 1)
594                        / 12;
595         final int mm = (12 * base.getYYYY() + base.getMonth() + months - 1)
596                        % 12 + 1;
597         final int dd = Math.min(
598             base.getDayOfMonth(), SerialDate.lastDayOfMonth(mm, yy)
```

Listing B-1 (continued)
`SerialDate.Java`

```
599         );
600         return SerialDate.createInstance(dd, mm, yy);
601
602     }
603
604     /**
605      * Creates a new date by adding the specified number of years to the base
606      * date.
607      *
608      * @param years  the number of years to add (can be negative).
609      * @param base  the base date.
610      *
611      * @return A new date.
612      */
613     public static SerialDate addYears(final int years, final SerialDate base) {
614
615         final int baseY = base.getYYYY();
616         final int baseM = base.getMonth();
617         final int baseD = base.getDayOfMonth();
618
619         final int targetY = baseY + years;
620         final int targetD = Math.min(
621             baseD, SerialDate.lastDayOfMonth(baseM, targetY)
622         );
623
624         return SerialDate.createInstance(targetD, baseM, targetY);
625
626     }
627
628     /**
629      * Returns the latest date that falls on the specified day-of-the-week and
630      * is BEFORE the base date.
631      *
632      * @param targetWeekday  a code for the target day-of-the-week.
633      * @param base  the base date.
634      *
635      * @return the latest date that falls on the specified day-of-the-week and
636      *         is BEFORE the base date.
637      */
638     public static SerialDate getPreviousDayOfWeek(final int targetWeekday,
639                                                    final SerialDate base) {
640
641         // check arguments...
642         if (!SerialDate.isValidWeekdayCode(targetWeekday)) {
643             throw new IllegalArgumentException(
644                 "Invalid day-of-the-week code."
645             );
646         }
647
648         // find the date...
649         final int adjust;
650         final int baseDOW = base.getDayOfWeek();
651         if (baseDOW > targetWeekday) {
652             adjust = Math.min(0, targetWeekday - baseDOW);
653         }
654         else {
655             adjust = -7 + Math.max(0, targetWeekday - baseDOW);
656         }
657
658         return SerialDate.addDays(adjust, base);
659
660     }
```

Listing B-1 (continued)
SerialDate.Java

```
661
662     /**
663      * Returns the earliest date that falls on the specified day-of-the-week
664      * and is AFTER the base date.
665      *
666      * @param targetWeekday  a code for the target day-of-the-week.
667      * @param base  the base date.
668      *
669      * @return the earliest date that falls on the specified day-of-the-week
670      *         and is AFTER the base date.
671      */
672     public static SerialDate getFollowingDayOfWeek(final int targetWeekday,
673                                                     final SerialDate base) {
674
675         // check arguments...
676         if (!SerialDate.isValidWeekdayCode(targetWeekday)) {
677             throw new IllegalArgumentException(
678                 "Invalid day-of-the-week code."
679             );
680         }
681
682         // find the date...
683         final int adjust;
684         final int baseDOW = base.getDayOfWeek();
685         if (baseDOW > targetWeekday) {
686             adjust = 7 + Math.min(0, targetWeekday - baseDOW);
687         }
688         else {
689             adjust = Math.max(0, targetWeekday - baseDOW);
690         }
691
692         return SerialDate.addDays(adjust, base);
693     }
694
695     /**
696      * Returns the date that falls on the specified day-of-the-week and is
697      * CLOSEST to the base date.
698      *
699      * @param targetDOW  a code for the target day-of-the-week.
700      * @param base  the base date.
701      *
702      * @return the date that falls on the specified day-of-the-week and is
703      *         CLOSEST to the base date.
704      */
705     public static SerialDate getNearestDayOfWeek(final int targetDOW,
706                                                   final SerialDate base) {
707
708         // check arguments...
709         if (!SerialDate.isValidWeekdayCode(targetDOW)) {
710             throw new IllegalArgumentException(
711                 "Invalid day-of-the-week code."
712             );
713         }
714
715         // find the date...
716         final int baseDOW = base.getDayOfWeek();
717         int adjust = -Math.abs(targetDOW - baseDOW);
718         if (adjust >= 4) {
719             adjust = 7 - adjust;
720         }
721         if (adjust <= -4) {
722             adjust = 7 + adjust;
```

Listing B-1 (continued)
`SerialDate.Java`

```
723         }
724         return SerialDate.addDays(adjust, base);
725
726     }
727
728     /**
729      * Rolls the date forward to the last day of the month.
730      *
731      * @param base  the base date.
732      *
733      * @return a new serial date.
734      */
735     public SerialDate getEndOfCurrentMonth(final SerialDate base) {
736         final int last = SerialDate.lastDayOfMonth(
737             base.getMonth(), base.getYYYY()
738         );
739         return SerialDate.createInstance(last, base.getMonth(), base.getYYYY());
740     }
741
742     /**
743      * Returns a string corresponding to the week-in-the-month code.
744      * <P>
745      * Need to find a better approach.
746      *
747      * @param count  an integer code representing the week-in-the-month.
748      *
749      * @return a string corresponding to the week-in-the-month code.
750      */
751     public static String weekInMonthToString(final int count) {
752
753         switch (count) {
754             case SerialDate.FIRST_WEEK_IN_MONTH : return "First";
755             case SerialDate.SECOND_WEEK_IN_MONTH : return "Second";
756             case SerialDate.THIRD_WEEK_IN_MONTH : return "Third";
757             case SerialDate.FOURTH_WEEK_IN_MONTH : return "Fourth";
758             case SerialDate.LAST_WEEK_IN_MONTH : return "Last";
759             default :
760                 return "SerialDate.weekInMonthToString(): invalid code.";
761         }
762
763     }
764
765     /**
766      * Returns a string representing the supplied 'relative'.
767      * <P>
768      * Need to find a better approach.
769      *
770      * @param relative  a constant representing the 'relative'.
771      *
772      * @return a string representing the supplied 'relative'.
773      */
774     public static String relativeToString(final int relative) {
775
776         switch (relative) {
777             case SerialDate.PRECEDING : return "Preceding";
778             case SerialDate.NEAREST : return "Nearest";
779             case SerialDate.FOLLOWING : return "Following";
780             default : return "ERROR : Relative To String";
781         }
782
783     }
784
```

Listing B-1 (continued)
SerialDate.Java

```
785     /**
786      * Factory method that returns an instance of some concrete subclass of
787      * {@link SerialDate}.
788      *
789      * @param day  the day (1-31).
790      * @param month  the month (1-12).
791      * @param yyyy  the year (in the range 1900 to 9999).
792      *
793      * @return An instance of {@link SerialDate}.
794      */
795     public static SerialDate createInstance(final int day, final int month,
796                                             final int yyyy) {
797         return new SpreadsheetDate(day, month, yyyy);
798     }
799
800     /**
801      * Factory method that returns an instance of some concrete subclass of
802      * {@link SerialDate}.
803      *
804      * @param serial  the serial number for the day (1 January 1900 = 2).
805      *
806      * @return a instance of SerialDate.
807      */
808     public static SerialDate createInstance(final int serial) {
809         return new SpreadsheetDate(serial);
810     }
811
812     /**
813      * Factory method that returns an instance of a subclass of SerialDate.
814      *
815      * @param date  A Java date object.
816      *
817      * @return a instance of SerialDate.
818      */
819     public static SerialDate createInstance(final java.util.Date date) {
820
821         final GregorianCalendar calendar = new GregorianCalendar();
822         calendar.setTime(date);
823         return new SpreadsheetDate(calendar.get(Calendar.DATE),
824                                    calendar.get(Calendar.MONTH) + 1,
825                                    calendar.get(Calendar.YEAR));
826
827     }
828
829     /**
830      * Returns the serial number for the date, where 1 January 1900 = 2 (this
831      * corresponds, almost, to the numbering system used in Microsoft Excel for
832      * Windows and Lotus 1-2-3).
833      *
834      * @return the serial number for the date.
835      */
836     public abstract int toSerial();
837
838     /**
839      * Returns a java.util.Date.  Since java.util.Date has more precision than
840      * SerialDate, we need to define a convention for the 'time of day'.
841      *
842      * @return this as <code>java.util.Date</code>.
843      */
844     public abstract java.util.Date toDate();
845
846     /**
```

Listing B-1 (continued)
`SerialDate.Java`

```
847      * Returns a description of the date.
848      *
849      * @return a description of the date.
850      */
851     public String getDescription() {
852         return this.description;
853     }
854
855     /**
856      * Sets the description for the date.
857      *
858      * @param description  the new description for the date.
859      */
860     public void setDescription(final String description) {
861         this.description = description;
862     }
863
864     /**
865      * Converts the date to a string.
866      *
867      * @return  a string representation of the date.
868      */
869     public String toString() {
870         return getDayOfMonth() + "-" + SerialDate.monthCodeToString(getMonth())
871                                + "-" + getYYYY();
872     }
873
874     /**
875      * Returns the year (assume a valid range of 1900 to 9999).
876      *
877      * @return the year.
878      */
879     public abstract int getYYYY();
880
881     /**
882      * Returns the month (January = 1, February = 2, March = 3).
883      *
884      * @return the month of the year.
885      */
886     public abstract int getMonth();
887
888     /**
889      * Returns the day of the month.
890      *
891      * @return the day of the month.
892      */
893     public abstract int getDayOfMonth();
894
895     /**
896      * Returns the day of the week.
897      *
898      * @return the day of the week.
899      */
900     public abstract int getDayOfWeek();
901
902     /**
903      * Returns the difference (in days) between this date and the specified
904      * 'other' date.
905      * <P>
906      * The result is positive if this date is after the 'other' date and
907      * negative if it is before the 'other' date.
908      *
```

Listing B-1 (continued)
SerialDate.Java

```
909      * @param other  the date being compared to.
910      *
911      * @return the difference between this and the other date.
912      */
913     public abstract int compare(SerialDate other);
914
915     /**
916      * Returns true if this SerialDate represents the same date as the
917      * specified SerialDate.
918      *
919      * @param other  the date being compared to.
920      *
921      * @return <code>true</code> if this SerialDate represents the same date as
922      *         the specified SerialDate.
923      */
924     public abstract boolean isOn(SerialDate other);
925
926     /**
927      * Returns true if this SerialDate represents an earlier date compared to
928      * the specified SerialDate.
929      *
930      * @param other  The date being compared to.
931      *
932      * @return <code>true</code> if this SerialDate represents an earlier date
933      *         compared to the specified SerialDate.
934      */
935     public abstract boolean isBefore(SerialDate other);
936
937     /**
938      * Returns true if this SerialDate represents the same date as the
939      * specified SerialDate.
940      *
941      * @param other  the date being compared to.
942      *
943      * @return <code>true<code> if this SerialDate represents the same date
944      *         as the specified SerialDate.
945      */
946     public abstract boolean isOnOrBefore(SerialDate other);
947
948     /**
949      * Returns true if this SerialDate represents the same date as the
950      * specified SerialDate.
951      *
952      * @param other  the date being compared to.
953      *
954      * @return <code>true</code> if this SerialDate represents the same date
955      *         as the specified SerialDate.
956      */
957     public abstract boolean isAfter(SerialDate other);
958
959     /**
960      * Returns true if this SerialDate represents the same date as the
961      * specified SerialDate.
962      *
963      * @param other  the date being compared to.
964      *
965      * @return <code>true</code> if this SerialDate represents the same date
966      *         as the specified SerialDate.
967      */
968     public abstract boolean isOnOrAfter(SerialDate other);
969
970     /**
971      * Returns <code>true</code> if this {@link SerialDate} is within the
```

Listing B-1 (continued)
`SerialDate.Java`

```
     * specified range (INCLUSIVE).  The date order of d1 and d2 is not
     * important.
     *
     * @param d1  a boundary date for the range.
     * @param d2  the other boundary date for the range.
     *
     * @return A boolean.
     */
    public abstract boolean isInRange(SerialDate d1, SerialDate d2);

    /**
     * Returns <code>true</code> if this {@link SerialDate} is within the
     * specified range (caller specifies whether or not the end-points are
     * included).  The date order of d1 and d2 is not important.
     *
     * @param d1  a boundary date for the range.
     * @param d2  the other boundary date for the range.
     * @param include  a code that controls whether or not the start and end
     *                 dates are included in the range.
     *
     * @return A boolean.
     */
    public abstract boolean isInRange(SerialDate d1, SerialDate d2,
                                      int include);

    /**
     * Returns the latest date that falls on the specified day-of-the-week and
     * is BEFORE this date.
     *
     * @param targetDOW  a code for the target day-of-the-week.
     *
     * @return the latest date that falls on the specified day-of-the-week and
     *         is BEFORE this date.
     */
    public SerialDate getPreviousDayOfWeek(final int targetDOW) {
        return getPreviousDayOfWeek(targetDOW, this);
    }

    /**
     * Returns the earliest date that falls on the specified day-of-the-week
     * and is AFTER this date.
     *
     * @param targetDOW  a code for the target day-of-the-week.
     *
     * @return the earliest date that falls on the specified day-of-the-week
     *         and is AFTER this date.
     */
    public SerialDate getFollowingDayOfWeek(final int targetDOW) {
        return getFollowingDayOfWeek(targetDOW, this);
    }

    /**
     * Returns the nearest date that falls on the specified day-of-the-week.
     *
     * @param targetDOW  a code for the target day-of-the-week.
     *
     * @return the nearest date that falls on the specified day-of-the-week.
     */
    public SerialDate getNearestDayOfWeek(final int targetDOW) {
        return getNearestDayOfWeek(targetDOW, this);
    }

}
```

Listing B-2
`SerialDateTest.java`

```
/* ===========================================================================
 * JCommon : a free general purpose class library for the Java(tm) platform
 * ===========================================================================
 *
 * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 *
 * Project Info:  http://www.jfree.org/jcommon/index.html
 *
 * This library is free software; you can redistribute it and/or modify it
 * under the terms of the GNU Lesser General Public License as published by
 * the Free Software Foundation; either version 2.1 of the License, or
 * (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
 * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
 * License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
 * USA.
 *
 * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
 * in the United States and other countries.]
 *
 * --------------------
 * SerialDateTests.java
 * --------------------
 * (C) Copyright 2001-2005, by Object Refinery Limited.
 *
 * Original Author:  David Gilbert (for Object Refinery Limited);
 * Contributor(s):   -;
 *
 * $Id: SerialDateTests.java,v 1.6 2005/11/16 15:58:40 taqua Exp $
 *
 * Changes
 * -------
 * 15-Nov-2001 : Version 1 (DG);
 * 25-Jun-2002 : Removed unnecessary import (DG);
 * 24-Oct-2002 : Fixed errors reported by Checkstyle (DG);
 * 13-Mar-2003 : Added serialization test (DG);
 * 05-Jan-2005 : Added test for bug report 1096282 (DG);
 *
 */

package org.jfree.date.junit;

import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.ObjectInput;
import java.io.ObjectInputStream;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;

import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

import org.jfree.date.MonthConstants;
import org.jfree.date.SerialDate;

```

Listing B-2 (continued)
`SerialDateTest.java`

```
 63 /**
 64  * Some JUnit tests for the {@link SerialDate} class.
 65  */
 66 public class SerialDateTests extends TestCase {
 67
 68     /** Date representing November 9. */
 69     private SerialDate nov9Y2001;
 70
 71     /**
 72      * Creates a new test case.
 73      *
 74      * @param name  the name.
 75      */
 76     public SerialDateTests(final String name) {
 77         super(name);
 78     }
 79
 80     /**
 81      * Returns a test suite for the JUnit test runner.
 82      *
 83      * @return The test suite.
 84      */
 85     public static Test suite() {
 86         return new TestSuite(SerialDateTests.class);
 87     }
 88
 89     /**
 90      * Problem set up.
 91      */
 92     protected void setUp() {
 93         this.nov9Y2001 = SerialDate.createInstance(9, MonthConstants.NOVEMBER, 2001);
 94     }
 95
 96     /**
 97      * 9 Nov 2001 plus two months should be 9 Jan 2002.
 98      */
 99     public void testAddMonthsTo9Nov2001() {
100         final SerialDate jan9Y2002 = SerialDate.addMonths(2, this.nov9Y2001);
101         final SerialDate answer = SerialDate.createInstance(9, 1, 2002);
102         assertEquals(answer, jan9Y2002);
103     }
104
105     /**
106      * A test case for a reported bug, now fixed.
107      */
108     public void testAddMonthsTo5Oct2003() {
109         final SerialDate d1 = SerialDate.createInstance(5, MonthConstants.OCTOBER, 2003);
110         final SerialDate d2 = SerialDate.addMonths(2, d1);
111         assertEquals(d2, SerialDate.createInstance(5, MonthConstants.DECEMBER, 2003));
112     }
113
114     /**
115      * A test case for a reported bug, now fixed.
116      */
117     public void testAddMonthsTo1Jan2003() {
118         final SerialDate d1 = SerialDate.createInstance(1, MonthConstants.JANUARY, 2003);
119         final SerialDate d2 = SerialDate.addMonths(0, d1);
120         assertEquals(d2, d1);
121     }
122
123     /**
124      * Monday preceding Friday 9 November 2001 should be 5 November.
```

Listing B-2 (continued)
SerialDateTest.java

```
125      */
126     public void testMondayPrecedingFriday9Nov2001() {
127         SerialDate mondayBefore = SerialDate.getPreviousDayOfWeek(
128             SerialDate.MONDAY, this.nov9Y2001
129         );
130         assertEquals(5, mondayBefore.getDayOfMonth());
131     }
132 
133     /**
134      * Monday following Friday 9 November 2001 should be 12 November.
135      */
136     public void testMondayFollowingFriday9Nov2001() {
137         SerialDate mondayAfter = SerialDate.getFollowingDayOfWeek(
138             SerialDate.MONDAY, this.nov9Y2001
139         );
140         assertEquals(12, mondayAfter.getDayOfMonth());
141     }
142 
143     /**
144      * Monday nearest Friday 9 November 2001 should be 12 November.
145      */
146     public void testMondayNearestFriday9Nov2001() {
147         SerialDate mondayNearest = SerialDate.getNearestDayOfWeek(
148             SerialDate.MONDAY, this.nov9Y2001
149         );
150         assertEquals(12, mondayNearest.getDayOfMonth());
151     }
152 
153     /**
154      * The Monday nearest to 22nd January 1970 falls on the 19th.
155      */
156     public void testMondayNearest22Jan1970() {
157         SerialDate jan22Y1970 = SerialDate.createInstance(22, MonthConstants.JANUARY, 1970);
158         SerialDate mondayNearest=SerialDate.getNearestDayOfWeek(SerialDate.MONDAY, jan22Y1970);
159         assertEquals(19, mondayNearest.getDayOfMonth());
160     }
161 
162     /**
163      * Problem that the conversion of days to strings returns the right result.  Actually, this
164      * result depends on the Locale so this test needs to be modified.
165      */
166     public void testWeekdayCodeToString() {
167 
168         final String test = SerialDate.weekdayCodeToString(SerialDate.SATURDAY);
169         assertEquals("Saturday", test);
170 
171     }
172 
173     /**
174      * Test the conversion of a string to a weekday.  Note that this test will fail if the
175      * default locale doesn't use English weekday names...devise a better test!
176      */
177     public void testStringToWeekday() {
178 
179         int weekday = SerialDate.stringToWeekdayCode("Wednesday");
180         assertEquals(SerialDate.WEDNESDAY, weekday);
181 
182         weekday = SerialDate.stringToWeekdayCode(" Wednesday ");
183         assertEquals(SerialDate.WEDNESDAY, weekday);
184 
```

Listing B-2 (continued)
`SerialDateTest.java`

```
185         weekday = SerialDate.stringToWeekdayCode("Wed");
186         assertEquals(SerialDate.WEDNESDAY, weekday);
187
188     }
189
190     /**
191      * Test the conversion of a string to a month.  Note that this test will fail if the
192      * default locale doesn't use English month names...devise a better test!
193      */
194     public void testStringToMonthCode() {
195
196         int m = SerialDate.stringToMonthCode("January");
197         assertEquals(MonthConstants.JANUARY, m);
198
199         m = SerialDate.stringToMonthCode(" January ");
200         assertEquals(MonthConstants.JANUARY, m);
201
202         m = SerialDate.stringToMonthCode("Jan");
203         assertEquals(MonthConstants.JANUARY, m);
204
205     }
206
207     /**
208      * Tests the conversion of a month code to a string.
209      */
210     public void testMonthCodeToStringCode() {
211
212         final String test = SerialDate.monthCodeToString(MonthConstants.DECEMBER);
213         assertEquals("December", test);
214
215     }
216
217     /**
218      * 1900 is not a leap year.
219      */
220     public void testIsNotLeapYear1900() {
221         assertTrue(!SerialDate.isLeapYear(1900));
222     }
223
224     /**
225      * 2000 is a leap year.
226      */
227     public void testIsLeapYear2000() {
228         assertTrue(SerialDate.isLeapYear(2000));
229     }
230
231     /**
232      * The number of leap years from 1900 up-to-and-including 1899 is 0.
233      */
234     public void testLeapYearCount1899() {
235         assertEquals(SerialDate.leapYearCount(1899), 0);
236     }
237
238     /**
239      * The number of leap years from 1900 up-to-and-including 1903 is 0.
240      */
241     public void testLeapYearCount1903() {
242         assertEquals(SerialDate.leapYearCount(1903), 0);
243     }
244
245     /**
246      * The number of leap years from 1900 up-to-and-including 1904 is 1.
247      */
```

Listing B-2 (continued)
SerialDateTest.java

```
248     public void testLeapYearCount1904() {
249         assertEquals(SerialDate.leapYearCount(1904), 1);
250     }
251
252     /**
253      * The number of leap years from 1900 up-to-and-including 1999 is 24.
254      */
255     public void testLeapYearCount1999() {
256         assertEquals(SerialDate.leapYearCount(1999), 24);
257     }
258
259     /**
260      * The number of leap years from 1900 up-to-and-including 2000 is 25.
261      */
262     public void testLeapYearCount2000() {
263         assertEquals(SerialDate.leapYearCount(2000), 25);
264     }
265
266     /**
267      * Serialize an instance, restore it, and check for equality.
268      */
269     public void testSerialization() {
270
271         SerialDate d1 = SerialDate.createInstance(15, 4, 2000);
272         SerialDate d2 = null;
273
274         try {
275             ByteArrayOutputStream buffer = new ByteArrayOutputStream();
276             ObjectOutput out = new ObjectOutputStream(buffer);
277             out.writeObject(d1);
278             out.close();
279
280             ObjectInput in = new ObjectInputStream(
                                    new ByteArrayInputStream(buffer.toByteArray()));
281             d2 = (SerialDate) in.readObject();
282             in.close();
283         }
284         catch (Exception e) {
285             System.out.println(e.toString());
286         }
287         assertEquals(d1, d2);
288
289     }
290
291     /**
292      * A test for bug report 1096282 (now fixed).
293      */
294     public void test1096282() {
295         SerialDate d = SerialDate.createInstance(29, 2, 2004);
296         d = SerialDate.addYears(1, d);
297         SerialDate expected = SerialDate.createInstance(28, 2, 2005);
298         assertTrue(d.isOn(expected));
299     }
300
301     /**
302      * Miscellaneous tests for the addMonths() method.
303      */
304     public void testAddMonths() {
305         SerialDate d1 = SerialDate.createInstance(31, 5, 2004);
306
```

Listing B-2 (continued)
`SerialDateTest.java`

```
307         SerialDate d2 = SerialDate.addMonths(1, d1);
308         assertEquals(30, d2.getDayOfMonth());
309         assertEquals(6, d2.getMonth());
310         assertEquals(2004, d2.getYYYY());
311
312         SerialDate d3 = SerialDate.addMonths(2, d1);
313         assertEquals(31, d3.getDayOfMonth());
314         assertEquals(7, d3.getMonth());
315         assertEquals(2004, d3.getYYYY());
316
317         SerialDate d4 = SerialDate.addMonths(1, SerialDate.addMonths(1, d1));
318         assertEquals(30, d4.getDayOfMonth());
319         assertEquals(7, d4.getMonth());
320         assertEquals(2004, d4.getYYYY());
321     }
322 }
```

Listing B-3
MonthConstants.java

```
/* ===========================================================================
 * JCommon : a free general purpose class library for the Java(tm) platform
 * ===========================================================================
 *
 * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 *
 * Project Info:  http://www.jfree.org/jcommon/index.html
 *
 * This library is free software; you can redistribute it and/or modify it
 * under the terms of the GNU Lesser General Public License as published by
 * the Free Software Foundation; either version 2.1 of the License, or
 * (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
 * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
 * License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
 * USA.
 *
 * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
 * in the United States and other countries.]
 *
 * -------------------
 * MonthConstants.java
 * -------------------
 * (C) Copyright 2002, 2003, by Object Refinery Limited.
 *
 * Original Author:  David Gilbert (for Object Refinery Limited);
 * Contributor(s):   -;
 *
 * $Id: MonthConstants.java,v 1.4 2005/11/16 15:58:40 taqua Exp $
 *
 * Changes
 * -------
 * 29-May-2002 : Version 1 (code moved from SerialDate class) (DG);
 *
 */

package org.jfree.date;

/**
 * Useful constants for months.  Note that these are NOT equivalent to the
 * constants defined by java.util.Calendar (where JANUARY=0 and DECEMBER=11).
 * <P>
 * Used by the SerialDate and RegularTimePeriod classes.
 *
 * @author David Gilbert
 */
public interface MonthConstants {

    /** Constant for January. */
    public static final int JANUARY = 1;

    /** Constant for February. */
    public static final int FEBRUARY = 2;

```

Listing B-3 (continued)
`MonthConstants.java`

```
61     /** Constant for March. */
62     public static final int MARCH = 3;
63
64     /** Constant for April. */
65     public static final int APRIL = 4;
66
67     /** Constant for May. */
68     public static final int MAY = 5;
69
70     /** Constant for June. */
71     public static final int JUNE = 6;
72
73     /** Constant for July. */
74     public static final int JULY = 7;
75
76     /** Constant for August. */
77     public static final int AUGUST = 8;
78
79     /** Constant for September. */
80     public static final int SEPTEMBER = 9;
81
82     /** Constant for October. */
83     public static final int OCTOBER = 10;
84
85     /** Constant for November. */
86     public static final int NOVEMBER = 11;
87
88     /** Constant for December. */
89     public static final int DECEMBER = 12;
90
91 }
```

Listing B-4
BobsSerialDateTest.java

```
 1 package org.jfree.date.junit;
 2
 3 import junit.framework.TestCase;
 4 import org.jfree.date.*;
 5 import static org.jfree.date.SerialDate.*;
 6
 7 import java.util.*;
 8
 9 public class BobsSerialDateTest extends TestCase {
10
11   public void testIsValidWeekdayCode() throws Exception {
12     for (int day = 1; day <= 7; day++)
13       assertTrue(isValidWeekdayCode(day));
14     assertFalse(isValidWeekdayCode(0));
15     assertFalse(isValidWeekdayCode(8));
16   }
17
18   public void testStringToWeekdayCode() throws Exception {
19
20     assertEquals(-1, stringToWeekdayCode("Hello"));
21     assertEquals(MONDAY, stringToWeekdayCode("Monday"));
22     assertEquals(MONDAY, stringToWeekdayCode("Mon"));
23 //todo    assertEquals(MONDAY,stringToWeekdayCode("monday"));
24 //    assertEquals(MONDAY,stringToWeekdayCode("MONDAY"));
25 //    assertEquals(MONDAY, stringToWeekdayCode("mon"));
26
27     assertEquals(TUESDAY, stringToWeekdayCode("Tuesday"));
28     assertEquals(TUESDAY, stringToWeekdayCode("Tue"));
29 //    assertEquals(TUESDAY,stringToWeekdayCode("tuesday"));
30 //    assertEquals(TUESDAY,stringToWeekdayCode("TUESDAY"));
31 //    assertEquals(TUESDAY, stringToWeekdayCode("tue"));
32 //    assertEquals(TUESDAY, stringToWeekdayCode("tues"));
33
34     assertEquals(WEDNESDAY, stringToWeekdayCode("Wednesday"));
35     assertEquals(WEDNESDAY, stringToWeekdayCode("Wed"));
36 //    assertEquals(WEDNESDAY,stringToWeekdayCode("wednesday"));
37 //    assertEquals(WEDNESDAY,stringToWeekdayCode("WEDNESDAY"));
38 //    assertEquals(WEDNESDAY, stringToWeekdayCode("wed"));
39
40     assertEquals(THURSDAY, stringToWeekdayCode("Thursday"));
41     assertEquals(THURSDAY, stringToWeekdayCode("Thu"));
42 //    assertEquals(THURSDAY,stringToWeekdayCode("thursday"));
43 //    assertEquals(THURSDAY,stringToWeekdayCode("THURSDAY"));
44 //    assertEquals(THURSDAY, stringToWeekdayCode("thu"));
45 //    assertEquals(THURSDAY, stringToWeekdayCode("thurs"));
46
47     assertEquals(FRIDAY, stringToWeekdayCode("Friday"));
48     assertEquals(FRIDAY, stringToWeekdayCode("Fri"));
49 //    assertEquals(FRIDAY,stringToWeekdayCode("friday"));
50 //    assertEquals(FRIDAY,stringToWeekdayCode("FRIDAY"));
51 //    assertEquals(FRIDAY, stringToWeekdayCode("fri"));
52
53     assertEquals(SATURDAY, stringToWeekdayCode("Saturday"));
54     assertEquals(SATURDAY, stringToWeekdayCode("Sat"));
55 //    assertEquals(SATURDAY,stringToWeekdayCode("saturday"));
56 //    assertEquals(SATURDAY,stringToWeekdayCode("SATURDAY"));
57 //    assertEquals(SATURDAY, stringToWeekdayCode("sat"));
58
59     assertEquals(SUNDAY, stringToWeekdayCode("Sunday"));
60     assertEquals(SUNDAY, stringToWeekdayCode("Sun"));
61 //    assertEquals(SUNDAY,stringToWeekdayCode("sunday"));
62 //    assertEquals(SUNDAY,stringToWeekdayCode("SUNDAY"));
63 //    assertEquals(SUNDAY, stringToWeekdayCode("sun"));
64   }
65
```

Listing B-4 (continued)
`BobsSerialDateTest.java`

```
 66   public void testWeekdayCodeToString() throws Exception {
 67     assertEquals("Sunday", weekdayCodeToString(SUNDAY));
 68     assertEquals("Monday", weekdayCodeToString(MONDAY));
 69     assertEquals("Tuesday", weekdayCodeToString(TUESDAY));
 70     assertEquals("Wednesday", weekdayCodeToString(WEDNESDAY));
 71     assertEquals("Thursday", weekdayCodeToString(THURSDAY));
 72     assertEquals("Friday", weekdayCodeToString(FRIDAY));
 73     assertEquals("Saturday", weekdayCodeToString(SATURDAY));
 74   }
 75
 76   public void testIsValidMonthCode() throws Exception {
 77     for (int i = 1; i <= 12; i++)
 78       assertTrue(isValidMonthCode(i));
 79     assertFalse(isValidMonthCode(0));
 80     assertFalse(isValidMonthCode(13));
 81   }
 82
 83   public void testMonthToQuarter() throws Exception {
 84     assertEquals(1, monthCodeToQuarter(JANUARY));
 85     assertEquals(1, monthCodeToQuarter(FEBRUARY));
 86     assertEquals(1, monthCodeToQuarter(MARCH));
 87     assertEquals(2, monthCodeToQuarter(APRIL));
 88     assertEquals(2, monthCodeToQuarter(MAY));
 89     assertEquals(2, monthCodeToQuarter(JUNE));
 90     assertEquals(3, monthCodeToQuarter(JULY));
 91     assertEquals(3, monthCodeToQuarter(AUGUST));
 92     assertEquals(3, monthCodeToQuarter(SEPTEMBER));
 93     assertEquals(4, monthCodeToQuarter(OCTOBER));
 94     assertEquals(4, monthCodeToQuarter(NOVEMBER));
 95     assertEquals(4, monthCodeToQuarter(DECEMBER));
 96
 97     try {
 98       monthCodeToQuarter(-1);
 99       fail("Invalid Month Code should throw exception");
100     } catch (IllegalArgumentException e) {
101     }
102   }
103
104   public void testMonthCodeToString() throws Exception {
105     assertEquals("January", monthCodeToString(JANUARY));
106     assertEquals("February", monthCodeToString(FEBRUARY));
107     assertEquals("March", monthCodeToString(MARCH));
108     assertEquals("April", monthCodeToString(APRIL));
109     assertEquals("May", monthCodeToString(MAY));
110     assertEquals("June", monthCodeToString(JUNE));
111     assertEquals("July", monthCodeToString(JULY));
112     assertEquals("August", monthCodeToString(AUGUST));
113     assertEquals("September", monthCodeToString(SEPTEMBER));
114     assertEquals("October", monthCodeToString(OCTOBER));
115     assertEquals("November", monthCodeToString(NOVEMBER));
116     assertEquals("December", monthCodeToString(DECEMBER));
117
118     assertEquals("Jan", monthCodeToString(JANUARY, true));
119     assertEquals("Feb", monthCodeToString(FEBRUARY, true));
120     assertEquals("Mar", monthCodeToString(MARCH, true));
121     assertEquals("Apr", monthCodeToString(APRIL, true));
122     assertEquals("May", monthCodeToString(MAY, true));
123     assertEquals("Jun", monthCodeToString(JUNE, true));
124     assertEquals("Jul", monthCodeToString(JULY, true));
125     assertEquals("Aug", monthCodeToString(AUGUST, true));
126     assertEquals("Sep", monthCodeToString(SEPTEMBER, true));
127     assertEquals("Oct", monthCodeToString(OCTOBER, true));
```

Listing B-4 (continued)
BobsSerialDateTest.java

```
128     assertEquals("Nov", monthCodeToString(NOVEMBER, true));
129     assertEquals("Dec", monthCodeToString(DECEMBER, true));
130
131     try {
132       monthCodeToString(-1);
133       fail("Invalid month code should throw exception");
134     } catch (IllegalArgumentException e) {
135     }
136
137   }
138
139   public void testStringToMonthCode() throws Exception {
140     assertEquals(JANUARY,stringToMonthCode("1"));
141     assertEquals(FEBRUARY,stringToMonthCode("2"));
142     assertEquals(MARCH,stringToMonthCode("3"));
143     assertEquals(APRIL,stringToMonthCode("4"));
144     assertEquals(MAY,stringToMonthCode("5"));
145     assertEquals(JUNE,stringToMonthCode("6"));
146     assertEquals(JULY,stringToMonthCode("7"));
147     assertEquals(AUGUST,stringToMonthCode("8"));
148     assertEquals(SEPTEMBER,stringToMonthCode("9"));
149     assertEquals(OCTOBER,stringToMonthCode("10"));
150     assertEquals(NOVEMBER, stringToMonthCode("11"));
151     assertEquals(DECEMBER,stringToMonthCode("12"));
152
153 //todo    assertEquals(-1, stringToMonthCode("0"));
154 //    assertEquals(-1, stringToMonthCode("13"));
155
156     assertEquals(-1,stringToMonthCode("Hello"));
157
158     for (int m = 1; m <= 12; m++) {
159       assertEquals(m, stringToMonthCode(monthCodeToString(m, false)));
160       assertEquals(m, stringToMonthCode(monthCodeToString(m, true)));
161     }
162
163 //    assertEquals(1,stringToMonthCode("jan"));
164 //    assertEquals(2,stringToMonthCode("feb"));
165 //    assertEquals(3,stringToMonthCode("mar"));
166 //    assertEquals(4,stringToMonthCode("apr"));
167 //    assertEquals(5,stringToMonthCode("may"));
168 //    assertEquals(6,stringToMonthCode("jun"));
169 //    assertEquals(7,stringToMonthCode("jul"));
170 //    assertEquals(8,stringToMonthCode("aug"));
171 //    assertEquals(9,stringToMonthCode("sep"));
172 //    assertEquals(10,stringToMonthCode("oct"));
173 //    assertEquals(11,stringToMonthCode("nov"));
174 //    assertEquals(12,stringToMonthCode("dec"));
175
176 //    assertEquals(1,stringToMonthCode("JAN"));
177 //    assertEquals(2,stringToMonthCode("FEB"));
178 //    assertEquals(3,stringToMonthCode("MAR"));
179 //    assertEquals(4,stringToMonthCode("APR"));
180 //    assertEquals(5,stringToMonthCode("MAY"));
181 //    assertEquals(6,stringToMonthCode("JUN"));
182 //    assertEquals(7,stringToMonthCode("JUL"));
183 //    assertEquals(8,stringToMonthCode("AUG"));
184 //    assertEquals(9,stringToMonthCode("SEP"));
185 //    assertEquals(10,stringToMonthCode("OCT"));
186 //    assertEquals(11,stringToMonthCode("NOV"));
187 //    assertEquals(12,stringToMonthCode("DEC"));
188
189 //    assertEquals(1,stringToMonthCode("january"));
190 //    assertEquals(2,stringToMonthCode("february"));
```

Listing B-4 (continued)
`BobsSerialDateTest.java`

```
191 //    assertEquals(3,stringToMonthCode("march"));
192 //    assertEquals(4,stringToMonthCode("april"));
193 //    assertEquals(5,stringToMonthCode("may"));
194 //    assertEquals(6,stringToMonthCode("june"));
195 //    assertEquals(7,stringToMonthCode("july"));
196 //    assertEquals(8,stringToMonthCode("august"));
197 //    assertEquals(9,stringToMonthCode("september"));
198 //    assertEquals(10,stringToMonthCode("october"));
199 //    assertEquals(11,stringToMonthCode("november"));
200 //    assertEquals(12,stringToMonthCode("december"));
201
202 //    assertEquals(1,stringToMonthCode("JANUARY"));
203 //    assertEquals(2,stringToMonthCode("FEBRUARY"));
204 //    assertEquals(3,stringToMonthCode("MAR"));
205 //    assertEquals(4,stringToMonthCode("APRIL"));
206 //    assertEquals(5,stringToMonthCode("MAY"));
207 //    assertEquals(6,stringToMonthCode("JUNE"));
208 //    assertEquals(7,stringToMonthCode("JULY"));
209 //    assertEquals(8,stringToMonthCode("AUGUST"));
210 //    assertEquals(9,stringToMonthCode("SEPTEMBER"));
211 //    assertEquals(10,stringToMonthCode("OCTOBER"));
212 //    assertEquals(11,stringToMonthCode("NOVEMBER"));
213 //    assertEquals(12,stringToMonthCode("DECEMBER"));
214   }
215
216   public void testIsValidWeekInMonthCode() throws Exception {
217     for (int w = 0; w <= 4; w++) {
218       assertTrue(isValidWeekInMonthCode(w));
219     }
220     assertFalse(isValidWeekInMonthCode(5));
221   }
222
223   public void testIsLeapYear() throws Exception {
224     assertFalse(isLeapYear(1900));
225     assertFalse(isLeapYear(1901));
226     assertFalse(isLeapYear(1902));
227     assertFalse(isLeapYear(1903));
228     assertTrue(isLeapYear(1904));
229     assertTrue(isLeapYear(1908));
230     assertFalse(isLeapYear(1955));
231     assertTrue(isLeapYear(1964));
232     assertTrue(isLeapYear(1980));
233     assertTrue(isLeapYear(2000));
234     assertFalse(isLeapYear(2001));
235     assertFalse(isLeapYear(2100));
236   }
237
238   public void testLeapYearCount() throws Exception {
239     assertEquals(0, leapYearCount(1900));
240     assertEquals(0, leapYearCount(1901));
241     assertEquals(0, leapYearCount(1902));
242     assertEquals(0, leapYearCount(1903));
243     assertEquals(1, leapYearCount(1904));
244     assertEquals(1, leapYearCount(1905));
245     assertEquals(1, leapYearCount(1906));
246     assertEquals(1, leapYearCount(1907));
247     assertEquals(2, leapYearCount(1908));
248     assertEquals(24, leapYearCount(1999));
249     assertEquals(25, leapYearCount(2001));
250     assertEquals(49, leapYearCount(2101));
251     assertEquals(73, leapYearCount(2201));
```

Listing B-4 (continued)
BobsSerialDateTest.java

```
252     assertEquals(97, leapYearCount(2301));
253     assertEquals(122, leapYearCount(2401));
254   }
255
256   public void testLastDayOfMonth() throws Exception {
257     assertEquals(31, lastDayOfMonth(JANUARY, 1901));
258     assertEquals(28, lastDayOfMonth(FEBRUARY, 1901));
259     assertEquals(31, lastDayOfMonth(MARCH, 1901));
260     assertEquals(30, lastDayOfMonth(APRIL, 1901));
261     assertEquals(31, lastDayOfMonth(MAY, 1901));
262     assertEquals(30, lastDayOfMonth(JUNE, 1901));
263     assertEquals(31, lastDayOfMonth(JULY, 1901));
264     assertEquals(31, lastDayOfMonth(AUGUST, 1901));
265     assertEquals(30, lastDayOfMonth(SEPTEMBER, 1901));
266     assertEquals(31, lastDayOfMonth(OCTOBER, 1901));
267     assertEquals(30, lastDayOfMonth(NOVEMBER, 1901));
268     assertEquals(31, lastDayOfMonth(DECEMBER, 1901));
269     assertEquals(29, lastDayOfMonth(FEBRUARY, 1904));
270   }
271
272   public void testAddDays() throws Exception {
273     SerialDate newYears = d(1, JANUARY, 1900);
274     assertEquals(d(2, JANUARY, 1900), addDays(1, newYears));
275     assertEquals(d(1, FEBRUARY, 1900), addDays(31, newYears));
276     assertEquals(d(1, JANUARY, 1901), addDays(365, newYears));
277     assertEquals(d(31, DECEMBER, 1904), addDays(5 * 365, newYears));
278   }
279
280   private static SpreadsheetDate d(int day, int month, int year) {return new
SpreadsheetDate(day, month, year);}
281
282   public void testAddMonths() throws Exception {
283     assertEquals(d(1, FEBRUARY, 1900), addMonths(1, d(1, JANUARY, 1900)));
284     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(31, JANUARY, 1900)));
285     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(30, JANUARY, 1900)));
286     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(29, JANUARY, 1900)));
287     assertEquals(d(28, FEBRUARY, 1900), addMonths(1, d(28, JANUARY, 1900)));
288     assertEquals(d(27, FEBRUARY, 1900), addMonths(1, d(27, JANUARY, 1900)));
289
290     assertEquals(d(30, JUNE, 1900), addMonths(5, d(31, JANUARY, 1900)));
291     assertEquals(d(30, JUNE, 1901), addMonths(17, d(31, JANUARY, 1900)));
292
293     assertEquals(d(29, FEBRUARY, 1904), addMonths(49, d(31, JANUARY, 1900)));
294
295   }
296
297   public void testAddYears() throws Exception {
298     assertEquals(d(1, JANUARY, 1901), addYears(1, d(1, JANUARY, 1900)));
299     assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(29, FEBRUARY, 1904)));
300     assertEquals(d(28, FEBRUARY, 1905), addYears(1, d(28, FEBRUARY, 1904)));
301     assertEquals(d(28, FEBRUARY, 1904), addYears(1, d(28, FEBRUARY, 1903)));
302   }
303
304   public void testGetPreviousDayOfWeek() throws Exception {
305     assertEquals(d(24, FEBRUARY, 2006), getPreviousDayOfWeek(FRIDAY, d(1, MARCH, 2006)));
306     assertEquals(d(22, FEBRUARY, 2006), getPreviousDayOfWeek(WEDNESDAY, d(1, MARCH, 2006)));
307     assertEquals(d(29, FEBRUARY, 2004), getPreviousDayOfWeek(SUNDAY, d(3, MARCH, 2004)));
308     assertEquals(d(29, DECEMBER, 2004), getPreviousDayOfWeek(WEDNESDAY, d(5, JANUARY, 2005)));
309
310     try {
311       getPreviousDayOfWeek(-1, d(1, JANUARY, 2006));
312       fail("Invalid day of week code should throw exception");
```

Listing B-4 (continued)
`BobsSerialDateTest.java`

```
313     } catch (IllegalArgumentException e) {
314     }
315   }
316
317   public void testGetFollowingDayOfWeek() throws Exception {
318 //    assertEquals(d(1, JANUARY, 2005),getFollowingDayOfWeek(SATURDAY, d(25, DECEMBER, 2004)));
319     assertEquals(d(1, JANUARY, 2005), getFollowingDayOfWeek(SATURDAY, d(26, DECEMBER, 2004)));
320     assertEquals(d(3, MARCH, 2004), getFollowingDayOfWeek(WEDNESDAY, d(28, FEBRUARY, 2004)));
321
322     try {
323       getFollowingDayOfWeek(-1, d(1, JANUARY, 2006));
324       fail("Invalid day of week code should throw exception");
325     } catch (IllegalArgumentException e) {
326     }
327   }
328
329   public void testGetNearestDayOfWeek() throws Exception {
330     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(16, APRIL, 2006)));
331     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(17, APRIL, 2006)));
332     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(18, APRIL, 2006)));
333     assertEquals(d(16, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(19, APRIL, 2006)));
334     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(20, APRIL, 2006)));
335     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(21, APRIL, 2006)));
336     assertEquals(d(23, APRIL, 2006), getNearestDayOfWeek(SUNDAY, d(22, APRIL, 2006)));
337
338 //todo    assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(16, APRIL, 2006)));
339     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(17, APRIL, 2006)));
340     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(18, APRIL, 2006)));
341     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(19, APRIL, 2006)));
342     assertEquals(d(17, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(20, APRIL, 2006)));
343     assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(21, APRIL, 2006)));
344     assertEquals(d(24, APRIL, 2006), getNearestDayOfWeek(MONDAY, d(22, APRIL, 2006)));
345
346 //    assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(16, APRIL, 2006)));
347 //    assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(17, APRIL, 2006)));
348     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(18, APRIL, 2006)));
349     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(19, APRIL, 2006)));
350     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(20, APRIL, 2006)));
351     assertEquals(d(18, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(21, APRIL, 2006)));
352     assertEquals(d(25, APRIL, 2006), getNearestDayOfWeek(TUESDAY, d(22, APRIL, 2006)));
353
354 //    assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(16, APRIL, 2006)));
355 //    assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(17, APRIL, 2006)));
356 //    assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(18, APRIL, 2006)));
357     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(19, APRIL, 2006)));
358     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(20, APRIL, 2006)));
359     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(21, APRIL, 2006)));
360     assertEquals(d(19, APRIL, 2006), getNearestDayOfWeek(WEDNESDAY, d(22, APRIL, 2006)));
361
362 //    assertEquals(d(13, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(16, APRIL, 2006)));
363 //    assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(17, APRIL, 2006)));
364 //    assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(18, APRIL, 2006)));
365 //    assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(19, APRIL, 2006)));
366     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(20, APRIL, 2006)));
367     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(21, APRIL, 2006)));
368     assertEquals(d(20, APRIL, 2006), getNearestDayOfWeek(THURSDAY, d(22, APRIL, 2006)));
369
370 //    assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(16, APRIL, 2006)));
371 //    assertEquals(d(14, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(17, APRIL, 2006)));
372 //    assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(18, APRIL, 2006)));
373 //    assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(19, APRIL, 2006)));
374 //    assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(20, APRIL, 2006)));
```

Listing B-4 (continued)
`BobsSerialDateTest.java`

```
375     assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(21, APRIL, 2006)));
376     assertEquals(d(21, APRIL, 2006), getNearestDayOfWeek(FRIDAY, d(22, APRIL, 2006)));
377
378 //    assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(16, APRIL, 2006)));
379 //    assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(17, APRIL, 2006)));
380 //    assertEquals(d(15, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(18, APRIL, 2006)));
381 //    assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(19, APRIL, 2006)));
382 //    assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(20, APRIL, 2006)));
383 //    assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(21, APRIL, 2006)));
384     assertEquals(d(22, APRIL, 2006), getNearestDayOfWeek(SATURDAY, d(22, APRIL, 2006)));
385
386     try {
387       getNearestDayOfWeek(-1, d(1, JANUARY, 2006));
388       fail("Invalid day of week code should throw exception");
389     } catch (IllegalArgumentException e) {
390     }
391   }
392
393   public void testEndOfCurrentMonth() throws Exception {
394     SerialDate d = SerialDate.createInstance(2);
395     assertEquals(d(31, JANUARY, 2006), d.getEndOfCurrentMonth(d(1, JANUARY, 2006)));
396     assertEquals(d(28, FEBRUARY, 2006), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2006)));
397     assertEquals(d(31, MARCH, 2006), d.getEndOfCurrentMonth(d(1, MARCH, 2006)));
398     assertEquals(d(30, APRIL, 2006), d.getEndOfCurrentMonth(d(1, APRIL, 2006)));
399     assertEquals(d(31, MAY, 2006), d.getEndOfCurrentMonth(d(1, MAY, 2006)));
400     assertEquals(d(30, JUNE, 2006), d.getEndOfCurrentMonth(d(1, JUNE, 2006)));
401     assertEquals(d(31, JULY, 2006), d.getEndOfCurrentMonth(d(1, JULY, 2006)));
402     assertEquals(d(31, AUGUST, 2006), d.getEndOfCurrentMonth(d(1, AUGUST, 2006)));
403     assertEquals(d(30, SEPTEMBER, 2006), d.getEndOfCurrentMonth(d(1, SEPTEMBER, 2006)));
404     assertEquals(d(31, OCTOBER, 2006), d.getEndOfCurrentMonth(d(1, OCTOBER, 2006)));
405     assertEquals(d(30, NOVEMBER, 2006), d.getEndOfCurrentMonth(d(1, NOVEMBER, 2006)));
406     assertEquals(d(31, DECEMBER, 2006), d.getEndOfCurrentMonth(d(1, DECEMBER, 2006)));
407     assertEquals(d(29, FEBRUARY, 2008), d.getEndOfCurrentMonth(d(1, FEBRUARY, 2008)));
408   }
409
410   public void testWeekInMonthToString() throws Exception {
411     assertEquals("First",weekInMonthToString(FIRST_WEEK_IN_MONTH));
412     assertEquals("Second",weekInMonthToString(SECOND_WEEK_IN_MONTH));
413     assertEquals("Third",weekInMonthToString(THIRD_WEEK_IN_MONTH));
414     assertEquals("Fourth",weekInMonthToString(FOURTH_WEEK_IN_MONTH));
415     assertEquals("Last",weekInMonthToString(LAST_WEEK_IN_MONTH));
416
417 //todo    try {
418 //      weekInMonthToString(-1);
419 //      fail("Invalid week code should throw exception");
420 //    } catch (IllegalArgumentException e) {
421 //    }
422   }
423
424   public void testRelativeToString() throws Exception {
425     assertEquals("Preceding",relativeToString(PRECEDING));
426     assertEquals("Nearest",relativeToString(NEAREST));
427     assertEquals("Following",relativeToString(FOLLOWING));
428
429 //todo    try {
430 //      relativeToString(-1000);
431 //      fail("Invalid relative code should throw exception");
432 //    } catch (IllegalArgumentException e) {
433 //    }
434   }
435
```

Listing B-4 (continued)
`BobsSerialDateTest.java`

```
436   public void testCreateInstanceFromDDMMYYY() throws Exception {
437     SerialDate date = createInstance(1, JANUARY, 1900);
438     assertEquals(1,date.getDayOfMonth());
439     assertEquals(JANUARY,date.getMonth());
440     assertEquals(1900,date.getYYYY());
441     assertEquals(2,date.toSerial());
442   }
443
444   public void testCreateInstanceFromSerial() throws Exception {
445     assertEquals(d(1, JANUARY, 1900),createInstance(2));
446     assertEquals(d(1, JANUARY, 1901), createInstance(367));
447   }
448
449   public void testCreateInstanceFromJavaDate() throws Exception {
450     assertEquals(d(1, JANUARY, 1900),
                    createInstance(new GregorianCalendar(1900,0,1).getTime()));
451     assertEquals(d(1, JANUARY, 2006),
                    createInstance(new GregorianCalendar(2006,0,1).getTime()));
452   }
453
454   public static void main(String[] args) {
455     junit.textui.TestRunner.run(BobsSerialDateTest.class);
456   }
457 }
```

Listing B-5
SpreadsheetDate.java

```
/* ===========================================================================
 * JCommon : a free general purpose class library for the Java(tm) platform
 * ===========================================================================
 *
 * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 *
 * Project Info:  http://www.jfree.org/jcommon/index.html
 *
 * This library is free software; you can redistribute it and/or modify it
 * under the terms of the GNU Lesser General Public License as published by
 * the Free Software Foundation; either version 2.1 of the License, or
 * (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
 * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
 * License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
 * USA.
 *
 * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
 * in the United States and other countries.]
 *
 * --------------------
 * SpreadsheetDate.java
 * --------------------
 * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 *
 * Original Author:  David Gilbert (for Object Refinery Limited);
 * Contributor(s):   -;
 *
 * $Id: SpreadsheetDate.java,v 1.8 2005/11/03 09:25:39 mungady Exp $
 *
 * Changes
 * -------
 * 11-Oct-2001 : Version 1 (DG);
 * 05-Nov-2001 : Added getDescription() and setDescription() methods (DG);
 * 12-Nov-2001 : Changed name from ExcelDate.java to SpreadsheetDate.java (DG);
 *               Fixed a bug in calculating day, month and year from serial
 *               number (DG);
 * 24-Jan-2002 : Fixed a bug in calculating the serial number from the day,
 *               month and year.  Thanks to Trevor Hills for the report (DG);
 * 29-May-2002 : Added equals(Object) method (SourceForge ID 558850) (DG);
 * 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
 * 13-Mar-2003 : Implemented Serializable (DG);
 * 04-Sep-2003 : Completed isInRange() methods (DG);
 * 05-Sep-2003 : Implemented Comparable (DG);
 * 21-Oct-2003 : Added hashCode() method (DG);
 *
 */

package org.jfree.date;

import java.util.Calendar;
import java.util.Date;

/**
 * Represents a date using an integer, in a similar fashion to the
 * implementation in Microsoft Excel.  The range of dates supported is
```

Listing B-5 (continued)
`SpreadsheetDate.java`

```
 * 1-Jan-1900 to 31-Dec-9999.
 * <P>
 * Be aware that there is a deliberate bug in Excel that recognises the year
 * 1900 as a leap year when in fact it is not a leap year. You can find more
 * information on the Microsoft website in article Q181370:
 * <P>
 * http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
 * <P>
 * Excel uses the convention that 1-Jan-1900 = 1.  This class uses the
 * convention 1-Jan-1900 = 2.
 * The result is that the day number in this class will be different to the
 * Excel figure for January and February 1900...but then Excel adds in an extra
 * day (29-Feb-1900 which does not actually exist!) and from that point forward
 * the day numbers will match.
 *
 * @author David Gilbert
 */
public class SpreadsheetDate extends SerialDate {

    /** For serialization. */
    private static final long serialVersionUID = -2039586705374454461L;

    /**
     * The day number (1-Jan-1900 = 2, 2-Jan-1900 = 3, ..., 31-Dec-9999 =
     * 2958465).
     */
    private int serial;

    /** The day of the month (1 to 28, 29, 30 or 31 depending on the month). */
    private int day;

    /** The month of the year (1 to 12). */
    private int month;

    /** The year (1900 to 9999). */
    private int year;

    /** An optional description for the date. */
    private String description;

    /**
     * Creates a new date instance.
     *
     * @param day  the day (in the range 1 to 28/29/30/31).
     * @param month  the month (in the range 1 to 12).
     * @param year  the year (in the range 1900 to 9999).
     */
    public SpreadsheetDate(final int day, final int month, final int year) {

        if ((year >= 1900) && (year <= 9999)) {
            this.year = year;
        }
        else {
            throw new IllegalArgumentException(
                "The 'year' argument must be in range 1900 to 9999."
            );
        }

        if ((month >= MonthConstants.JANUARY)
                && (month <= MonthConstants.DECEMBER)) {
            this.month = month;
        }
```

Listing B-5 (continued)
SpreadsheetDate.java

```
125         else {
126             throw new IllegalArgumentException(
127                 "The 'month' argument must be in the range 1 to 12."
128             );
129         }
130
131         if ((day >= 1) && (day <= SerialDate.lastDayOfMonth(month, year))) {
132             this.day = day;
133         }
134         else {
135             throw new IllegalArgumentException("Invalid 'day' argument.");
136         }
137
138         // the serial number needs to be synchronised with the day-month-year...
139         this.serial = calcSerial(day, month, year);
140
141         this.description = null;
142
143     }
144
145     /**
146      * Standard constructor - creates a new date object representing the
147      * specified day number (which should be in the range 2 to 2958465.
148      *
149      * @param serial  the serial number for the day (range: 2 to 2958465).
150      */
151     public SpreadsheetDate(final int serial) {
152
153         if ((serial >= SERIAL_LOWER_BOUND) && (serial <= SERIAL_UPPER_BOUND)) {
154             this.serial = serial;
155         }
156         else {
157             throw new IllegalArgumentException(
158                 "SpreadsheetDate: Serial must be in range 2 to 2958465.");
159         }
160
161         // the day-month-year needs to be synchronised with the serial number...
162         calcDayMonthYear();
163
164     }
165
166     /**
167      * Returns the description that is attached to the date.  It is not
168      * required that a date have a description, but for some applications it
169      * is useful.
170      *
171      * @return The description that is attached to the date.
172      */
173     public String getDescription() {
174         return this.description;
175     }
176
177     /**
178      * Sets the description for the date.
179      *
180      * @param description  the description for this date (<code>null</code>
181      *                     permitted).
182      */
183     public void setDescription(final String description) {
184         this.description = description;
185     }
186
```

Listing B-5 (continued)
`SpreadsheetDate.java`

```
187     /**
188      * Returns the serial number for the date, where 1 January 1900 = 2
189      * (this corresponds, almost, to the numbering system used in Microsoft
190      * Excel for Windows and Lotus 1-2-3).
191      *
192      * @return The serial number of this date.
193      */
194     public int toSerial() {
195         return this.serial;
196     }
197
198     /**
199      * Returns a <code>java.util.Date</code> equivalent to this date.
200      *
201      * @return The date.
202      */
203     public Date toDate() {
204         final Calendar calendar = Calendar.getInstance();
205         calendar.set(getYYYY(), getMonth() - 1, getDayOfMonth(), 0, 0, 0);
206         return calendar.getTime();
207     }
208
209     /**
210      * Returns the year (assume a valid range of 1900 to 9999).
211      *
212      * @return The year.
213      */
214     public int getYYYY() {
215         return this.year;
216     }
217
218     /**
219      * Returns the month (January = 1, February = 2, March = 3).
220      *
221      * @return The month of the year.
222      */
223     public int getMonth() {
224         return this.month;
225     }
226
227     /**
228      * Returns the day of the month.
229      *
230      * @return The day of the month.
231      */
232     public int getDayOfMonth() {
233         return this.day;
234     }
235
236     /**
237      * Returns a code representing the day of the week.
238      * <P>
239      * The codes are defined in the {@link SerialDate} class as:
240      * <code>SUNDAY</code>, <code>MONDAY</code>, <code>TUESDAY</code>,
241      * <code>WEDNESDAY</code>, <code>THURSDAY</code>, <code>FRIDAY</code>, and
242      * <code>SATURDAY</code>.
243      *
244      * @return A code representing the day of the week.
245      */
246     public int getDayOfWeek() {
247         return (this.serial + 6) % 7 + 1;
248     }
```

Listing B-5 (continued)
`SpreadsheetDate.java`

```
249
250     /**
251      * Tests the equality of this date with an arbitrary object.
252      * <P>
253      * This method will return true ONLY if the object is an instance of the
254      * {@link SerialDate} base class, and it represents the same day as this
255      * {@link SpreadsheetDate}.
256      *
257      * @param object  the object to compare (<code>null</code> permitted).
258      *
259      * @return A boolean.
260      */
261     public boolean equals(final Object object) {
262
263         if (object instanceof SerialDate) {
264             final SerialDate s = (SerialDate) object;
265             return (s.toSerial() == this.toSerial());
266         }
267         else {
268             return false;
269         }
270
271     }
272
273     /**
274      * Returns a hash code for this object instance.
275      *
276      * @return A hash code.
277      */
278     public int hashCode() {
279         return toSerial();
280     }
281
282     /**
283      * Returns the difference (in days) between this date and the specified
284      * 'other' date.
285      *
286      * @param other  the date being compared to.
287      *
288      * @return The difference (in days) between this date and the specified
289      *         'other' date.
290      */
291     public int compare(final SerialDate other) {
292         return this.serial - other.toSerial();
293     }
294
295     /**
296      * Implements the method required by the Comparable interface.
297      *
298      * @param other  the other object (usually another SerialDate).
299      *
300      * @return A negative integer, zero, or a positive integer as this object
301      *         is less than, equal to, or greater than the specified object.
302      */
303     public int compareTo(final Object other) {
304         return compare((SerialDate) other);
305     }
306
307     /**
308      * Returns true if this SerialDate represents the same date as the
309      * specified SerialDate.
310      *
```

Listing B-5 (continued)
`SpreadsheetDate.java`

```
311      * @param other  the date being compared to.
312      *
313      * @return <code>true</code> if this SerialDate represents the same date as
314      *         the specified SerialDate.
315      */
316     public boolean isOn(final SerialDate other) {
317         return (this.serial == other.toSerial());
318     }
319
320     /**
321      * Returns true if this SerialDate represents an earlier date compared to
322      * the specified SerialDate.
323      *
324      * @param other  the date being compared to.
325      *
326      * @return <code>true</code> if this SerialDate represents an earlier date
327      *         compared to the specified SerialDate.
328      */
329     public boolean isBefore(final SerialDate other) {
330         return (this.serial < other.toSerial());
331     }
332
333     /**
334      * Returns true if this SerialDate represents the same date as the
335      * specified SerialDate.
336      *
337      * @param other  the date being compared to.
338      *
339      * @return <code>true</code> if this SerialDate represents the same date
340      *         as the specified SerialDate.
341      */
342     public boolean isOnOrBefore(final SerialDate other) {
343         return (this.serial <= other.toSerial());
344     }
345
346     /**
347      * Returns true if this SerialDate represents the same date as the
348      * specified SerialDate.
349      *
350      * @param other  the date being compared to.
351      *
352      * @return <code>true</code> if this SerialDate represents the same date
353      *         as the specified SerialDate.
354      */
355     public boolean isAfter(final SerialDate other) {
356         return (this.serial > other.toSerial());
357     }
358
359     /**
360      * Returns true if this SerialDate represents the same date as the
361      * specified SerialDate.
362      *
363      * @param other  the date being compared to.
364      *
365      * @return <code>true</code> if this SerialDate represents the same date as
366      *         the specified SerialDate.
367      */
368     public boolean isOnOrAfter(final SerialDate other) {
369         return (this.serial >= other.toSerial());
370     }
371
372     /**
373      * Returns <code>true</code> if this {@link SerialDate} is within the
```

Listing B-5 (continued)
`SpreadsheetDate.java`

```
374      * specified range (INCLUSIVE).  The date order of d1 and d2 is not
375      * important.
376      *
377      * @param d1  a boundary date for the range.
378      * @param d2  the other boundary date for the range.
379      *
380      * @return A boolean.
381      */
382     public boolean isInRange(final SerialDate d1, final SerialDate d2) {
383         return isInRange(d1, d2, SerialDate.INCLUDE_BOTH);
384     }
385
386     /**
387      * Returns true if this SerialDate is within the specified range (caller
388      * specifies whether or not the end-points are included).  The order of d1
389      * and d2 is not important.
390      *
391      * @param d1  one boundary date for the range.
392      * @param d2  a second boundary date for the range.
393      * @param include  a code that controls whether or not the start and end
394      *                 dates are included in the range.
395      *
396      * @return <code>true</code> if this SerialDate is within the specified
397      *         range.
398      */
399     public boolean isInRange(final SerialDate d1, final SerialDate d2,
400                              final int include) {
401         final int s1 = d1.toSerial();
402         final int s2 = d2.toSerial();
403         final int start = Math.min(s1, s2);
404         final int end = Math.max(s1, s2);
405
406         final int s = toSerial();
407         if (include == SerialDate.INCLUDE_BOTH) {
408             return (s >= start && s <= end);
409         }
410         else if (include == SerialDate.INCLUDE_FIRST) {
411             return (s >= start && s < end);
412         }
413         else if (include == SerialDate.INCLUDE_SECOND) {
414             return (s > start && s <= end);
415         }
416         else {
417             return (s > start && s < end);
418         }
419     }
420
421     /**
422      * Calculate the serial number from the day, month and year.
423      * <P>
424      * 1-Jan-1900 = 2.
425      *
426      * @param d  the day.
427      * @param m  the month.
428      * @param y  the year.
429      *
430      * @return the serial number from the day, month and year.
431      */
432     private int calcSerial(final int d, final int m, final int y) {
433         final int yy = ((y - 1900) * 365) + SerialDate.leapYearCount(y - 1);
434         int mm = SerialDate.AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[m];
435         if (m > MonthConstants.FEBRUARY) {
```

Listing B-5 (continued)
`SpreadsheetDate.java`

```
436             if (SerialDate.isLeapYear(y)) {
437                 mm = mm + 1;
438             }
439         }
440         final int dd = d;
441         return yy + mm + dd + 1;
442     }
443
444     /**
445      * Calculate the day, month and year from the serial number.
446      */
447     private void calcDayMonthYear() {
448
449         // get the year from the serial date
450         final int days = this.serial - SERIAL_LOWER_BOUND;
451         // overestimated because we ignored leap days
452         final int overestimatedYYYY = 1900 + (days / 365);
453         final int leaps = SerialDate.leapYearCount(overestimatedYYYY);
454         final int nonleapdays = days - leaps;
455         // underestimated because we overestimated years
456         int underestimatedYYYY = 1900 + (nonleapdays / 365);
457
458         if (underestimatedYYYY == overestimatedYYYY) {
459             this.year = underestimatedYYYY;
460         }
461         else {
462             int ss1 = calcSerial(1, 1, underestimatedYYYY);
463             while (ss1 <= this.serial) {
464                 underestimatedYYYY = underestimatedYYYY + 1;
465                 ss1 = calcSerial(1, 1, underestimatedYYYY);
466             }
467             this.year = underestimatedYYYY - 1;
468         }
469
470         final int ss2 = calcSerial(1, 1, this.year);
471
472         int[] daysToEndOfPrecedingMonth
473             = AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;
474
475         if (isLeapYear(this.year)) {
476             daysToEndOfPrecedingMonth
477                 = LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH;
478         }
479
480         // get the month from the serial date
481         int mm = 1;
482         int sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1;
483         while (sss < this.serial) {
484             mm = mm + 1;
485             sss = ss2 + daysToEndOfPrecedingMonth[mm] - 1;
486         }
487         this.month = mm - 1;
488
489         // what's left is d(+1);
490         this.day = this.serial - ss2
491                    - daysToEndOfPrecedingMonth[this.month] + 1;
492
493     }
494
495 }
```

Listing B-6
RelativeDayOfWeekRule.java

```
/* ===========================================================================
 * JCommon : a free general purpose class library for the Java(tm) platform
 * ===========================================================================
 *
 * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
 *
 * Project Info:  http://www.jfree.org/jcommon/index.html
 *
 * This library is free software; you can redistribute it and/or modify it
 * under the terms of the GNU Lesser General Public License as published by
 * the Free Software Foundation; either version 2.1 of the License, or
 * (at your option) any later version.
 *
 * This library is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY
 * or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public
 * License for more details.
 *
 * You should have received a copy of the GNU Lesser General Public
 * License along with this library; if not, write to the Free Software
 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301,
 * USA.
 *
 * [Java is a trademark or registered trademark of Sun Microsystems, Inc.
 * in the United States and other countries.]
 *
 * --------------------------
 * RelativeDayOfWeekRule.java
 * --------------------------
 * (C) Copyright 2000-2003, by Object Refinery Limited and Contributors.
 *
 * Original Author:  David Gilbert (for Object Refinery Limited);
 * Contributor(s):   -;
 *
 * $Id: RelativeDayOfWeekRule.java,v 1.6 2005/11/16 15:58:40 taqua Exp $
 *
 * Changes (from 26-Oct-2001)
 * --------------------------
 * 26-Oct-2001 : Changed package to com.jrefinery.date.*;
 * 03-Oct-2002 : Fixed errors reported by Checkstyle (DG);
 *
 */

package org.jfree.date;

/**
 * An annual date rule that returns a date for each year based on (a) a
 * reference rule; (b) a day of the week; and (c) a selection parameter
 * (SerialDate.PRECEDING, SerialDate.NEAREST, SerialDate.FOLLOWING).
 * <P>
 * For example, Good Friday can be specified as 'the Friday PRECEDING Easter
 * Sunday'.
 *
 * @author David Gilbert
 */
public class RelativeDayOfWeekRule extends AnnualDateRule {

    /** A reference to the annual date rule on which this rule is based. */
    private AnnualDateRule subrule;

    /**
     * The day of the week (SerialDate.MONDAY, SerialDate.TUESDAY, and so on).
```

Listing B-6 (continued)
`RelativeDayOfWeekRule.java`

```
63      */
64     private int dayOfWeek;
65
66     /** Specifies which day of the week (PRECEDING, NEAREST or FOLLOWING). */
67     private int relative;
68
69     /**
70      * Default constructor - builds a rule for the Monday following 1 January.
71      */
72     public RelativeDayOfWeekRule() {
73         this(new DayAndMonthRule(), SerialDate.MONDAY, SerialDate.FOLLOWING);
74     }
75
76     /**
77      * Standard constructor - builds rule based on the supplied sub-rule.
78      *
79      * @param subrule  the rule that determines the reference date.
80      * @param dayOfWeek  the day-of-the-week relative to the reference date.
81      * @param relative  indicates *which* day-of-the-week (preceding, nearest
82      *                  or following).
83      */
84     public RelativeDayOfWeekRule(final AnnualDateRule subrule,
85             final int dayOfWeek, final int relative) {
86         this.subrule = subrule;
87         this.dayOfWeek = dayOfWeek;
88         this.relative = relative;
89     }
90
91     /**
92      * Returns the sub-rule (also called the reference rule).
93      *
94      * @return The annual date rule that determines the reference date for this
95      *         rule.
96      */
97     public AnnualDateRule getSubrule() {
98         return this.subrule;
99     }
100
101    /**
102     * Sets the sub-rule.
103     *
104     * @param subrule  the annual date rule that determines the reference date
105     *                 for this rule.
106     */
107    public void setSubrule(final AnnualDateRule subrule) {
108        this.subrule = subrule;
109    }
110
111    /**
112     * Returns the day-of-the-week for this rule.
113     *
114     * @return the day-of-the-week for this rule.
115     */
116    public int getDayOfWeek() {
117        return this.dayOfWeek;
118    }
119
120    /**
121     * Sets the day-of-the-week for this rule.
122     *
123     * @param dayOfWeek  the day-of-the-week (SerialDate.MONDAY,
124     *                   SerialDate.TUESDAY, and so on).
```

Listing B-6 (continued)
`RelativeDayOfWeekRule.java`

```
125      */
126     public void setDayOfWeek(final int dayOfWeek) {
127         this.dayOfWeek = dayOfWeek;
128     }
129
130     /**
131      * Returns the 'relative' attribute, that determines *which*
132      * day-of-the-week we are interested in (SerialDate.PRECEDING,
133      * SerialDate.NEAREST or SerialDate.FOLLOWING).
134      *
135      * @return The 'relative' attribute.
136      */
137     public int getRelative() {
138         return this.relative;
139     }
140
141     /**
142      * Sets the 'relative' attribute (SerialDate.PRECEDING, SerialDate.NEAREST,
143      * SerialDate.FOLLOWING).
144      *
145      * @param relative  determines *which* day-of-the-week is selected by this
146      *                  rule.
147      */
148     public void setRelative(final int relative) {
149         this.relative = relative;
150     }
151
152     /**
153      * Creates a clone of this rule.
154      *
155      * @return a clone of this rule.
156      *
157      * @throws CloneNotSupportedException this should never happen.
158      */
159     public Object clone() throws CloneNotSupportedException {
160         final RelativeDayOfWeekRule duplicate
161             = (RelativeDayOfWeekRule) super.clone();
162         duplicate.subrule = (AnnualDateRule) duplicate.getSubrule().clone();
163         return duplicate;
164     }
165
166     /**
167      * Returns the date generated by this rule, for the specified year.
168      *
169      * @param year  the year (1900 &amp;lt;= year &amp;lt;= 9999).
170      *
171      * @return The date generated by the rule for the given year (possibly
172      *         <code>null</code>).
173      */
174     public SerialDate getDate(final int year) {
175
176         // check argument...
177         if ((year < SerialDate.MINIMUM_YEAR_SUPPORTED)
178             || (year > SerialDate.MAXIMUM_YEAR_SUPPORTED)) {
179             throw new IllegalArgumentException(
180                 "RelativeDayOfWeekRule.getDate(): year outside valid range.");
181         }
182
183         // calculate the date...
184         SerialDate result = null;
185         final SerialDate base = this.subrule.getDate(year);
186
```

Listing B-6 (continued)
`RelativeDayOfWeekRule.java`

```
187         if (base != null) {
188             switch (this.relative) {
189                 case(SerialDate.PRECEDING):
190                     result = SerialDate.getPreviousDayOfWeek(this.dayOfWeek,
191                             base);
192                     break;
193                 case(SerialDate.NEAREST):
194                     result = SerialDate.getNearestDayOfWeek(this.dayOfWeek,
195                             base);
196                     break;
197                 case(SerialDate.FOLLOWING):
198                     result = SerialDate.getFollowingDayOfWeek(this.dayOfWeek,
199                             base);
200                     break;
201                 default:
202                     break;
203             }
204         }
205         return result;
206
207     }
208
209 }
```

Listing B-7
DayDate.java (Final)

```
/* ===========================================================================
 * JCommon : a free general purpose class library for the Java(tm) platform
 * ===========================================================================
 *
 * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
...
 */
package org.jfree.date;

import java.io.Serializable;
import java.util.*;

/**
 * An abstract class that represents immutable dates with a precision of
 * one day.  The implementation will map each date to an integer that
 * represents an ordinal number of days from some fixed origin.
 *
 * Why not just use java.util.Date?  We will, when it makes sense.  At times,
 * java.util.Date can be *too* precise - it represents an instant in time,
 * accurate to 1/1000th of a second (with the date itself depending on the
 * time-zone).  Sometimes we just want to represent a particular day (e.g. 21
 * January 2015) without concerning ourselves about the time of day, or the
 * time-zone, or anything else.  That's what we've defined DayDate for.
 *
 * Use DayDateFactory.makeDate to create an instance.
 *
 * @author David Gilbert
 * @author Robert C. Martin did a lot of refactoring.
 */

public abstract class DayDate implements Comparable, Serializable {
  public abstract int getOrdinalDay();
  public abstract int getYear();
  public abstract Month getMonth();
  public abstract int getDayOfMonth();

  protected abstract Day getDayOfWeekForOrdinalZero();

  public DayDate plusDays(int days) {
    return DayDateFactory.makeDate(getOrdinalDay() + days);
  }

  public DayDate plusMonths(int months) {
    int thisMonthAsOrdinal = getMonth().toInt() - Month.JANUARY.toInt();
    int thisMonthAndYearAsOrdinal = 12 * getYear() + thisMonthAsOrdinal;
    int resultMonthAndYearAsOrdinal = thisMonthAndYearAsOrdinal + months;
    int resultYear = resultMonthAndYearAsOrdinal / 12;
    int resultMonthAsOrdinal = resultMonthAndYearAsOrdinal % 12 + Month.JANUARY.toInt();
    Month resultMonth = Month.fromInt(resultMonthAsOrdinal);
    int resultDay = correctLastDayOfMonth(getDayOfMonth(), resultMonth, resultYear);
    return DayDateFactory.makeDate(resultDay, resultMonth, resultYear);
  }

  public DayDate plusYears(int years) {
    int resultYear = getYear() + years;
    int resultDay = correctLastDayOfMonth(getDayOfMonth(), getMonth(), resultYear);
    return DayDateFactory.makeDate(resultDay, getMonth(), resultYear);
  }

  private int correctLastDayOfMonth(int day, Month month, int year) {
    int lastDayOfMonth = DateUtil.lastDayOfMonth(month, year);
    if (day > lastDayOfMonth)
```

Listing B-7 (continued)
`DayDate.java (Final)`

```
92          day = lastDayOfMonth;
93      return day;
94    }
95
96    public DayDate getPreviousDayOfWeek(Day targetDayOfWeek) {
97      int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
98      if (offsetToTarget >= 0)
99        offsetToTarget -= 7;
100     return plusDays(offsetToTarget);
101   }
102
103   public DayDate getFollowingDayOfWeek(Day targetDayOfWeek) {
104     int offsetToTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
105     if (offsetToTarget <= 0)
106       offsetToTarget += 7;
107     return plusDays(offsetToTarget);
108   }
109
110   public DayDate getNearestDayOfWeek(Day targetDayOfWeek) {
111     int offsetToThisWeeksTarget = targetDayOfWeek.toInt() - getDayOfWeek().toInt();
112     int offsetToFutureTarget = (offsetToThisWeeksTarget + 7) % 7;
113     int offsetToPreviousTarget = offsetToFutureTarget - 7;
114
115     if (offsetToFutureTarget > 3)
116       return plusDays(offsetToPreviousTarget);
117     else
118       return plusDays(offsetToFutureTarget);
119   }
120
121   public DayDate getEndOfMonth() {
122     Month month = getMonth();
123     int year = getYear();
124     int lastDay = DateUtil.lastDayOfMonth(month, year);
125     return DayDateFactory.makeDate(lastDay, month, year);
126   }
127
128   public Date toDate() {
129     final Calendar calendar = Calendar.getInstance();
130     int ordinalMonth = getMonth().toInt() - Month.JANUARY.toInt();
131     calendar.set(getYear(), ordinalMonth, getDayOfMonth(), 0, 0, 0);
132     return calendar.getTime();
133   }
134
135   public String toString() {
136     return String.format("%02d-%s-%d", getDayOfMonth(), getMonth(), getYear());
137   }
138
139   public Day getDayOfWeek() {
140     Day startingDay = getDayOfWeekForOrdinalZero();
141     int startingOffset = startingDay.toInt() - Day.SUNDAY.toInt();
142     int ordinalOfDayOfWeek = (getOrdinalDay() + startingOffset) % 7;
143     return Day.fromInt(ordinalOfDayOfWeek + Day.SUNDAY.toInt());
144   }
145
146   public int daysSince(DayDate date) {
147     return getOrdinalDay() - date.getOrdinalDay();
148   }
149
150   public boolean isOn(DayDate other) {
151     return getOrdinalDay() == other.getOrdinalDay();
152   }
153
```

Listing B-7 (continued)
DayDate.java (Final)

```
154   public boolean isBefore(DayDate other) {
155     return getOrdinalDay() < other.getOrdinalDay();
156   }
157
158   public boolean isOnOrBefore(DayDate other) {
159     return getOrdinalDay() <= other.getOrdinalDay();
160   }
161
162   public boolean isAfter(DayDate other) {
163     return getOrdinalDay() > other.getOrdinalDay();
164   }
165
166   public boolean isOnOrAfter(DayDate other) {
167     return getOrdinalDay() >= other.getOrdinalDay();
168   }
169
170   public boolean isInRange(DayDate d1, DayDate d2) {
171     return isInRange(d1, d2, DateInterval.CLOSED);
172   }
173
174   public boolean isInRange(DayDate d1, DayDate d2, DateInterval interval) {
175     int left = Math.min(d1.getOrdinalDay(), d2.getOrdinalDay());
176     int right = Math.max(d1.getOrdinalDay(), d2.getOrdinalDay());
177     return interval.isIn(getOrdinalDay(), left, right);
178   }
179 }
```

Listing B-8
`Month.java (Final)`

```
1 package org.jfree.date;
2
3 import java.text.DateFormatSymbols;
4
5 public enum Month {
6   JANUARY(1), FEBRUARY(2), MARCH(3),
7   APRIL(4),   MAY(5),      JUNE(6),
8   JULY(7),    AUGUST(8),   SEPTEMBER(9),
9   OCTOBER(10),NOVEMBER(11),DECEMBER(12);
10  private static DateFormatSymbols dateFormatSymbols = new DateFormatSymbols();
11  private static final int[] LAST_DAY_OF_MONTH =
12    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
13
14  private int index;
15
16  Month(int index) {
17    this.index = index;
18  }
19
20  public static Month fromInt(int monthIndex) {
21    for (Month m : Month.values()) {
22      if (m.index == monthIndex)
23        return m;
24    }
25    throw new IllegalArgumentException("Invalid month index " + monthIndex);
26  }
27
28  public int lastDay() {
29    return LAST_DAY_OF_MONTH[index];
30  }
31
32  public int quarter() {
33    return 1 + (index - 1) / 3;
34  }
35
36  public String toString() {
37    return dateFormatSymbols.getMonths()[index - 1];
38  }
39
40  public String toShortString() {
41    return dateFormatSymbols.getShortMonths()[index - 1];
42  }
43
44  public static Month parse(String s) {
45    s = s.trim();
46    for (Month m : Month.values())
47      if (m.matches(s))
48        return m;
49
50    try {
51      return fromInt(Integer.parseInt(s));
52    }
53    catch (NumberFormatException e) {}
54    throw new IllegalArgumentException("Invalid month " + s);
55  }
56
57  private boolean matches(String s) {
58    return s.equalsIgnoreCase(toString()) ||
59           s.equalsIgnoreCase(toShortString());
60  }
61
62  public int toInt() {
63    return index;
64  }
65 }
```

Listing B-9
Day.java (Final)

```
 1 package org.jfree.date;
 2
 3 import java.util.Calendar;
 4 import java.text.DateFormatSymbols;
 5
 6 public enum Day {
 7   MONDAY(Calendar.MONDAY),
 8   TUESDAY(Calendar.TUESDAY),
 9   WEDNESDAY(Calendar.WEDNESDAY),
10   THURSDAY(Calendar.THURSDAY),
11   FRIDAY(Calendar.FRIDAY),
12   SATURDAY(Calendar.SATURDAY),
13   SUNDAY(Calendar.SUNDAY);
14
15   private final int index;
16   private static DateFormatSymbols dateSymbols = new DateFormatSymbols();
17
18   Day(int day) {
19     index = day;
20   }
21
22   public static Day fromInt(int index) throws IllegalArgumentException {
23     for (Day d : Day.values())
24       if (d.index == index)
25         return d;
26     throw new IllegalArgumentException(
27       String.format("Illegal day index: %d.", index));
28   }
29
30   public static Day parse(String s) throws IllegalArgumentException {
31     String[] shortWeekdayNames =
32       dateSymbols.getShortWeekdays();
33     String[] weekDayNames =
34       dateSymbols.getWeekdays();
35
36     s = s.trim();
37     for (Day day : Day.values()) {
38       if (s.equalsIgnoreCase(shortWeekdayNames[day.index]) ||
39           s.equalsIgnoreCase(weekDayNames[day.index])) {
40         return day;
41       }
42     }
43     throw new IllegalArgumentException(
44       String.format("%s is not a valid weekday string", s));
45   }
46
47   public String toString() {
48     return dateSymbols.getWeekdays()[index];
49   }
50
51   public int toInt() {
52     return index;
53   }
54 }
```

Listing B-10
`DateInterval.java (Final)`

```
 1 package org.jfree.date;
 2
 3 public enum DateInterval {
 4   OPEN {
 5     public boolean isIn(int d, int left, int right) {
 6       return d > left && d < right;
 7     }
 8   },
 9   CLOSED_LEFT {
10     public boolean isIn(int d, int left, int right) {
11       return d >= left && d < right;
12     }
13   },
14   CLOSED_RIGHT {
15     public boolean isIn(int d, int left, int right) {
16       return d > left && d <= right;
17     }
18   },
19   CLOSED {
20     public boolean isIn(int d, int left, int right) {
21       return d >= left && d <= right;
22     }
23   };
24
25   public abstract boolean isIn(int d, int left, int right);
26 }
```

Listing B-11
`WeekInMonth.java (Final)`

```
 1 package org.jfree.date;
 2
 3 public enum WeekInMonth {
 4   FIRST(1), SECOND(2), THIRD(3), FOURTH(4), LAST(0);
 5   private final int index;
 6
 7   WeekInMonth(int index) {
 8     this.index = index;
 9   }
10
11   public int toInt() {
12     return index;
13   }
14 }
```

Listing B-12
`WeekdayRange.java (Final)`

```
1 package org.jfree.date;
2
3 public enum WeekdayRange {
4   LAST, NEAREST, NEXT
5 }
```

Listing B-13
`DateUtil.java (Final)`

```
 1 package org.jfree.date;
 2
 3 import java.text.DateFormatSymbols;
 4
 5 public class DateUtil {
 6   private static DateFormatSymbols dateFormatSymbols = new DateFormatSymbols();
 7
 8   public static String[] getMonthNames() {
 9     return dateFormatSymbols.getMonths();
10   }
11
12   public static boolean isLeapYear(int year) {
13     boolean fourth = year % 4 == 0;
14     boolean hundredth = year % 100 == 0;
15     boolean fourHundredth = year % 400 == 0;
16     return fourth && (!hundredth || fourHundredth);
17   }
18
19   public static int lastDayOfMonth(Month month, int year) {
20     if (month == Month.FEBRUARY && isLeapYear(year))
21       return month.lastDay() + 1;
22     else
23       return month.lastDay();
24   }
25
26   public static int leapYearCount(int year) {
27     int leap4 = (year - 1896) / 4;
28     int leap100 = (year - 1800) / 100;
29     int leap400 = (year - 1600) / 400;
30     return leap4 - leap100 + leap400;
31   }
32 }
```

Listing B-14
`DayDateFactory.java (Final)`

```
 1 package org.jfree.date;
 2
 3 public abstract class DayDateFactory {
 4   private static DayDateFactory factory = new SpreadsheetDateFactory();
 5   public static void setInstance(DayDateFactory factory) {
 6     DayDateFactory.factory = factory;
 7   }
 8
 9   protected abstract DayDate _makeDate(int ordinal);
10   protected abstract DayDate _makeDate(int day, Month month, int year);
11   protected abstract DayDate _makeDate(int day, int month, int year);
12   protected abstract DayDate _makeDate(java.util.Date date);
13   protected abstract int _getMinimumYear();
14   protected abstract int _getMaximumYear();
15
16   public static DayDate makeDate(int ordinal) {
17     return factory._makeDate(ordinal);
18   }
19
20   public static DayDate makeDate(int day, Month month, int year) {
21     return factory._makeDate(day, month, year);
22   }
23
24   public static DayDate makeDate(int day, int month, int year) {
25     return factory._makeDate(day, month, year);
26   }
27
28   public static DayDate makeDate(java.util.Date date) {
29     return factory._makeDate(date);
30   }
31
32   public static int getMinimumYear() {
33     return factory._getMinimumYear();
34   }
35
36   public static int getMaximumYear() {
37     return factory._getMaximumYear();
38   }
39 }
```

Listing B-15
`SpreadsheetDateFactory.java (Final)`

```
 1 package org.jfree.date;
 2
 3 import java.util.*;
 4
 5 public class SpreadsheetDateFactory extends DayDateFactory {
 6   public DayDate _makeDate(int ordinal) {
 7     return new SpreadsheetDate(ordinal);
 8   }
 9
10   public DayDate _makeDate(int day, Month month, int year) {
11     return new SpreadsheetDate(day, month, year);
12   }
13
14   public DayDate _makeDate(int day, int month, int year) {
15     return new SpreadsheetDate(day, month, year);
16   }
17
18   public DayDate _makeDate(Date date) {
19     final GregorianCalendar calendar = new GregorianCalendar();
20     calendar.setTime(date);
21     return new SpreadsheetDate(
22       calendar.get(Calendar.DATE),
23       Month.fromInt(calendar.get(Calendar.MONTH) + 1),
24       calendar.get(Calendar.YEAR));
25   }
26
27   protected int _getMinimumYear() {
28     return SpreadsheetDate.MINIMUM_YEAR_SUPPORTED;
29   }
30
31   protected int _getMaximumYear() {
32     return SpreadsheetDate.MAXIMUM_YEAR_SUPPORTED;
33   }
34 }
```

Listing B-16
`SpreadsheetDate.java (Final)`

```
  1 /* ===========================================================================
  2  * JCommon : a free general purpose class library for the Java(tm) platform
  3  * ===========================================================================
  4  *
  5  * (C) Copyright 2000-2005, by Object Refinery Limited and Contributors.
  6  *
...
 52  *
 53  */
 54
 55 package org.jfree.date;
 56
 57 import static org.jfree.date.Month.FEBRUARY;
 58
 59 import java.util.*;
 60
 61 /**
 62  * Represents a date using an integer, in a similar fashion to the
 63  * implementation in Microsoft Excel.  The range of dates supported is
 64  * 1-Jan-1900 to 31-Dec-9999.
 65  * <p/>
 66  * Be aware that there is a deliberate bug in Excel that recognises the year
 67  * 1900 as a leap year when in fact it is not a leap year. You can find more
 68  * information on the Microsoft website in article Q181370:
 69  * <p/>
 70  * http://support.microsoft.com/support/kb/articles/Q181/3/70.asp
 71  * <p/>
 72  * Excel uses the convention that 1-Jan-1900 = 1.  This class uses the
 73  * convention 1-Jan-1900 = 2.
 74  * The result is that the day number in this class will be different to the
 75  * Excel figure for January and February 1900...but then Excel adds in an extra
 76  * day (29-Feb-1900 which does not actually exist!) and from that point forward
 77  * the day numbers will match.
 78  *
 79  * @author David Gilbert
 80  */
 81 public class SpreadsheetDate extends DayDate {
 82   public static final int EARLIEST_DATE_ORDINAL = 2;     // 1/1/1900
 83   public static final int LATEST_DATE_ORDINAL = 2958465; // 12/31/9999
 84   public static final int MINIMUM_YEAR_SUPPORTED = 1900;
 85   public static final int MAXIMUM_YEAR_SUPPORTED = 9999;
 86   static final int[] AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
 87     {0, 0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365};
 88   static final int[] LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH =
 89     {0, 0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366};
 90
 91   private int ordinalDay;
 92   private int day;
 93   private Month month;
 94   private int year;
 95
 96   public SpreadsheetDate(int day, Month month, int year) {
 97     if (year < MINIMUM_YEAR_SUPPORTED || year > MAXIMUM_YEAR_SUPPORTED)
 98       throw new IllegalArgumentException(
 99         "The 'year' argument must be in range " +
100         MINIMUM_YEAR_SUPPORTED + " to " + MAXIMUM_YEAR_SUPPORTED + ".");
101     if (day < 1 || day > DateUtil.lastDayOfMonth(month, year))
102       throw new IllegalArgumentException("Invalid 'day' argument.");
103
104     this.year = year;
105     this.month = month;
```

Listing B-16 (continued)
SpreadsheetDate.java (Final)

```
106     this.day = day;
107     ordinalDay = calcOrdinal(day, month, year);
108   }
109
110   public SpreadsheetDate(int day, int month, int year) {
111     this(day, Month.fromInt(month), year);
112   }
113
114   public SpreadsheetDate(int serial) {
115     if (serial < EARLIEST_DATE_ORDINAL || serial > LATEST_DATE_ORDINAL)
116       throw new IllegalArgumentException(
117         "SpreadsheetDate: Serial must be in range 2 to 2958465.");
118
119     ordinalDay = serial;
120     calcDayMonthYear();
121   }
122
123   public int getOrdinalDay() {
124     return ordinalDay;
125   }
126
127   public int getYear() {
128     return year;
129   }
130
131   public Month getMonth() {
132     return month;
133   }
134
135   public int getDayOfMonth() {
136     return day;
137   }
138
139   protected Day getDayOfWeekForOrdinalZero() {return Day.SATURDAY;}
140
141   public boolean equals(Object object) {
142     if (!(object instanceof DayDate))
143       return false;
144
145     DayDate date = (DayDate) object;
146     return date.getOrdinalDay() == getOrdinalDay();
147   }
148
149   public int hashCode() {
150     return getOrdinalDay();
151   }
152
153   public int compareTo(Object other) {
154     return daysSince((DayDate) other);
155   }
156
157   private int calcOrdinal(int day, Month month, int year) {
158     int leapDaysForYear = DateUtil.leapYearCount(year - 1);
159     int daysUpToYear = (year - MINIMUM_YEAR_SUPPORTED) * 365 + leapDaysForYear;
160     int daysUpToMonth = AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[month.toInt()];
161     if (DateUtil.isLeapYear(year) && month.toInt() > FEBRUARY.toInt())
162       daysUpToMonth++;
163     int daysInMonth = day - 1;
164     return daysUpToYear + daysUpToMonth + daysInMonth + EARLIEST_DATE_ORDINAL;
165   }
166
```

Listing B-16 (continued)
`SpreadsheetDate.java (Final)`

```
167   private void calcDayMonthYear() {
168     int days = ordinalDay - EARLIEST_DATE_ORDINAL;
169     int overestimatedYear = MINIMUM_YEAR_SUPPORTED + days / 365;
170     int nonleapdays = days - DateUtil.leapYearCount(overestimatedYear);
171     int underestimatedYear = MINIMUM_YEAR_SUPPORTED + nonleapdays / 365;
172
173     year = huntForYearContaining(ordinalDay, underestimatedYear);
174     int firstOrdinalOfYear = firstOrdinalOfYear(year);
175     month = huntForMonthContaining(ordinalDay, firstOrdinalOfYear);
176     day = ordinalDay - firstOrdinalOfYear - daysBeforeThisMonth(month.toInt());
177   }
178
179   private Month huntForMonthContaining(int anOrdinal, int firstOrdinalOfYear) {
180     int daysIntoThisYear = anOrdinal - firstOrdinalOfYear;
181     int aMonth = 1;
182     while (daysBeforeThisMonth(aMonth) < daysIntoThisYear)
183       aMonth++;
184
185     return Month.fromInt(aMonth - 1);
186   }
187
188   private int daysBeforeThisMonth(int aMonth) {
189     if (DateUtil.isLeapYear(year))
190       return LEAP_YEAR_AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[aMonth] - 1;
191     else
192       return AGGREGATE_DAYS_TO_END_OF_PRECEDING_MONTH[aMonth] - 1;
193   }
194
195   private int huntForYearContaining(int anOrdinalDay, int startingYear) {
196     int aYear = startingYear;
197     while (firstOrdinalOfYear(aYear) <= anOrdinalDay)
198       aYear++;
199
200     return aYear - 1;
201   }
202
203   private int firstOrdinalOfYear(int year) {
204     return calcOrdinal(1, Month.JANUARY, year);
205   }
206
207   public static DayDate createInstance(Date date) {
208     GregorianCalendar calendar = new GregorianCalendar();
209     calendar.setTime(date);
210     return new SpreadsheetDate(calendar.get(Calendar.DATE),
211                                Month.fromInt(calendar.get(Calendar.MONTH) + 1),
212                                calendar.get(Calendar.YEAR));
213
214   }
215 }
```

Appendix C

Cross References of Heuristics

Cross references of Smells and Heuristics. All other cross references can be deleted.

C1 .....16-276, 16-279, 17-292
C2 ..... 16-279, 16-285, 16-295, 17-292
C3 ..... 16-283, 16-285, 16-288, 17-293
C4 17-293
C5 17-293
E1 17-294
E2 17-294
F1 ..... 14-239, 17-295
F2 17-295
F3 17-295
F4 .....14-289, 16-273, 16-285, 16-287, 16-288, 17-295
G1 ..... 16-276, 17-295
G2 .....16-273, 16-274, 17-296
G3 ..... 16-274, 17-296
G4 .....9-31, 16-279, 16-286, 16-291, 17-297
G5 .....9-31, 16-279, 16-286, 16-291, 16-296, 17-297
G6 .....6-106, 16-280, 16-283, 16-284, 16-289, 16-293, 16-294, 16-296, 17-299
G7 .....16-281, 16-283, 17-300
G8 ..... 16-283, 17-301
G9 ..... 16-283, 16-285, 16-286, 16-287, 17-302
G10 ..... 5-86, 15-264, 16-276, 16-284, 17-302
G11 ..... 15-264, 16-284, 16-288, 16-292, 17-302
G12 ..... 16-284, 16-285, 16-286, 16-287, 16-288, 16-295, 17-303
G13 .....16-286, 16-288, 17-303
G14 .....16-288, 16-292, 17-304

G15 ..... 16-288, 17-305
G16 ..... 16-289, 17-306
G17 ..... 16-289, 17-307, 17-312
G18 ..... 16-289, 16-290, 16-291, 17-308
G19 ..... 16-290, 16-291, 16-292, 17-309
G20 ..... 16-290, 17-309
G21 ..... 16-291, 17-310
G22 ..... 16-294, 17-322
G23 ..... ??-44, 14-239, 16-295, 17-313
G24 ..... 16-296, 17-313
G25 ..... 16-296, 17-314
G26 17-316
G27 17-316
G28 ..... 15-262, 17-317
G29 ..... 15-262, 17-317
G30 ..... 15-263, 17-317
G31 ..... 15-264, 17-318
G32 ..... 15-265, 17-319
G33 ..... 15-265, 15-266, 17-320
G34 ..... 1-40, 6-106, 17-321
G35 ..... 5-90, 17-323
G36 ..... 6-103, 17-324
J1 ..... 16-276, 17-325
J2 ..... 16-278, 16-285, 17-326
J3 ..... 16-283, 16-285, 17-327
N1 ..... 15-264, 16-277, 16-279, 16-282, 16-287, 16-288, 16-289, 16-290, 16-294, 16-296, 17-328
N2 ..... 16-277, 17-330
N3 ..... 16-284, 16-288, 17-331
N4 ..... 15-263, 16-291, 17-332
N5 ..... 2-26, 14-221, 15-262, 17-332
N6 ..... 15-261, 17-333
N7 ..... 15-263, 17-333
T1 ..... 16-273, 16-274, 17-334
T2 ..... 16-273, 17-334
T3 ..... 16-274, 17-334
T4 17-334
T5 ..... 16-274, 16-275, 17-335
T6 ..... 16-275, 17-335
T7 ..... 16-275, 17-335
T8 ..... 16-275, 17-335
T9 17-336