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`, `getFollowingDayOfWeek`, 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.

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 `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);
}
 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));
  }

  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 `Spread-sheetDate` (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 {
public final int index;
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` 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);
 }

 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);

      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.