Chapter 15: A Longer Example

Reading individual refactorings is important, because you have to know the moves in order to play the game. Yet reading about refactorings one at a time can also miss the point, for the value of refactoring only comes when combine a long series of refactorings together. Individually each refactoring is a minor change, almost not worth the effort. Yet the result of a series of them is a big change that can do a lot for the long term flexibility of the system.

Chapter 1 provides something of an example of this. This chapter extends on that by giving you a larger example of refactoring. Of course this is still very much an example system. When you refactor in the real world the process is fast and fluid, but reading an account of it is hard work. Even an example like the one here takes a lot of pages to explain, and my reviewers all remarked that it is hard going on the reader. So don't be surprised if you find this chapter a struggle. Indeed I suspect a significant proportion of readers won't find this section too valuable. That's why it's at the end.

The example I chose deals with inheritance. Inheritance is one of the most famous features of object-oriented systems, and so it should come as no surprise that refactoring is a good way to help make the best use of inheritance. I am going to work with a bunch of classes that form a classic inheritance structure. I will start with a program that has four unrelated classes which exhibit similar behavior. During the course of the refactoring I will merge them into a superclass and subclasses. The similar behavior is not that obvious, not simply a case of spotting that there methods with the same name and same body. As such this is quite a long chapter, but it says a lot about the way in which you do refactoring.

The Initial Program

The example problem is that of an electricity utility charging its customers. The utility has several kinds of customers and charges them in different ways. In all cases, however, it wants to know how many dollars it should charge them for the latest reading.

The types of customer are: Residential, Disability, Lifeline, and Business.

In each case the principal task of the class is to calculate the latest charge. The algorithms show quite different code but, as we shall see, there is still a great deal of commonality.

To begin with, however, we shall look at the code. I shall start with the Residential Site class (Figure 15.1).

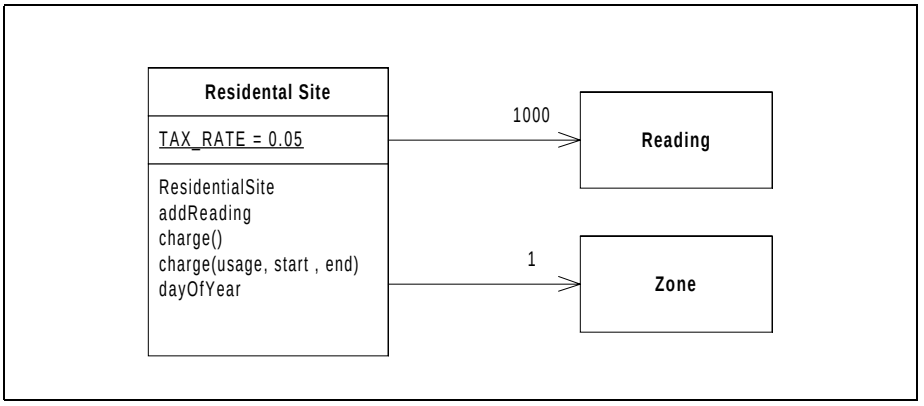

Figure 15.1: The residential site

```
class ResidentialSite {


   private Reading[] _readings = new Reading[1000];
   private static final double TAX_RATE = 0.05;
   private Zone _zone;

   ResidentialSite (Zone zone) {
       _zone = zone;
   }
```

Readings are added to the residential site with

```
public void addReading(Reading newReading)
  {
    // add reading to end of array
    int i = 0;
    while (_readings[i] != null) i++;
    _readings[i] = newReading;
  }
```

Like all the site classes there is a public charge method which calculates the latest charge for the site.

```
public Dollars charge()
  {
    // find last reading
    int i = 0;
    while (_readings[i] != null) i++;

    int usage = _readings[i-1].amount() - _readings[i-2].amount();
    Date end = _readings[i-1].date();
    Date start = _readings[i-2].date();
    start.setDate(start.getDate() + 1);                    //set to begining of period
    return charge(usage, start, end);
  }
```

As you can see, all it does is set up the arguments for another charge method, which is private to the class.

```
private Dollars charge(int usage, Date start, Date end) {

    Dollars result;
    double summerFraction;

    // Find out how much of period is in the summer
    if (start.after(_zone.summerEnd()) || end.before(_zone.summerStart()))
        summerFraction = 0;
    else if (!start.before(_zone.summerStart()) && !start.after(_zone.summerEnd()) &&
              !end.before(_zone.summerStart()) && !end.after(_zone.summerEnd()))
        summerFraction = 1;
    else {    // part in summer part in winter
        double summerDays;
        if (start.before(_zone.summerStart()) || start.after(_zone.summerEnd())) {
           // end is in the summer
           summerDays = dayOfYear(end) - dayOfYear (_zone.summerStart()) + 1;
        } else {
           // start is in summer
           summerDays = dayOfYear(_zone.summerEnd()) - dayOfYear (start) + 1;
        };
        summerFraction = summerDays / (dayOfYear(end) - dayOfYear(start) + 1);
    };
```

```
        result =  new Dollars ((usage * _zone.summerRate() * summerFraction) +
            (usage * _zone.winterRate() * (1 - summerFraction)));


        result = result.plus(new Dollars (result.times(TAX_RATE)));

        Dollars fuel = new Dollars(usage * 0.0175);
        result = result.plus(fuel);

        result = new Dollars (result.plus(fuel.times(TAX_RATE)));

        return result;

    }
```

This is a complicated beast. It makes use of one further method to help with date calculations.

```
    int dayOfYear(Date arg) {
    int result;
    switch (arg.getMonth()) {
        case 0:
           result = 0;
           break;
        case 1:
           result = 31;
           break;
        case 2:
           result = 59;
           break;
        case 3:
           result = 90;
           break;
        case 4:
           result = 120;
           break;
        case 5:
           result = 151;
           break;
        case 6:
           result = 181;
           break;
        case 7:
           result = 212;
           break;
        case 8:
           result = 243;
           break;
        case 9:
           result = 273;
           break;
```

```
            case 10:
               result = 304;
               break;
            case 11:
               result = 334;
               break;
            default :
               throw new IllegalArgumentException();
         };
         result += arg.getDate();

         //check leap year
         if ((arg.getYear()%4 == 0) && ((arg.getYear() % 100 != 0) ||
             ((arg.getYear() + 1900) % 400 == 0)))
         {
             result++;
         };

         return result;
      }
```

That (at least for our purposes) is the residential site class. The residential site uses a number of other classes. The readings and zone classes (Figure 15.2) are just simple encapsulated records.

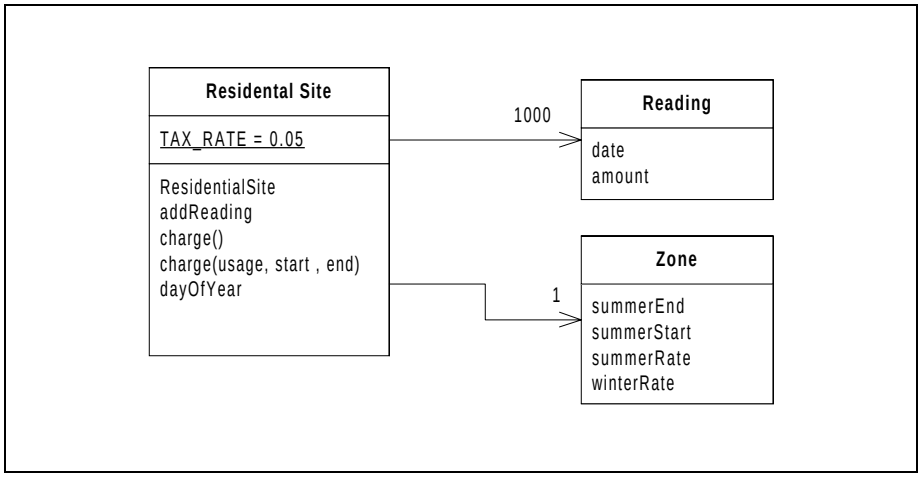

Figure 15.2: Adding the reading and zone classes

```
class Reading {
    public Date date() {
           return _date;
```

```
    }
    public int amount() {
          return _amount;
    }
    public Reading(int amount, Date date) {
          _amount = amount;
          _date = date;
    }

    private Date _date;
    private int _amount;
 }

class Zone {
  public Zone persist()                {
      Registrar.add("Zone", this);
      return this;
  }

  public static Zone get (String name)                      {
      return (Zone) Registrar.get("Zone", name);
  }

  public Date summerEnd() {
      return _summerEnd;
  }

  public Date summerStart() {
      return _summerStart;
  }

  public double winterRate() {
      return _winterRate;
  }

  public double summerRate() {
      return _summerRate;
  }


  Zone (String name, double summerRate, double winterRate,
                    Date summerStart, Date summerEnd) {
      _name = name;
      _summerRate = summerRate;
      _winterRate = winterRate;
      _summerStart = summerStart;
      _summerEnd = summerEnd;
  };

  private Date _summerEnd;
  private Date _summerStart;
```

```
    private double _winterRate;
    private double _summerRate;
    }
```

The dollars class is a use of the Quantity pattern. It combines the notion of an amount and a currency. I'm not going to go into too many details here. Essentially you create dollars objects with a constructor that has a number for the amount. The class supports some basic arithmetic operations.

An important part of the dollars class is the fact that it rounds all numbers to the nearest cent, a behavior which is often very important in financial systems. As my friend Ron Jeffries told me: "Be kind to pennies, and they will be kind to you".

Next up is a class used for calculating charges for people who are disabled (Figure 15.3). Its structure is similar to the residential case except for a few more constants.

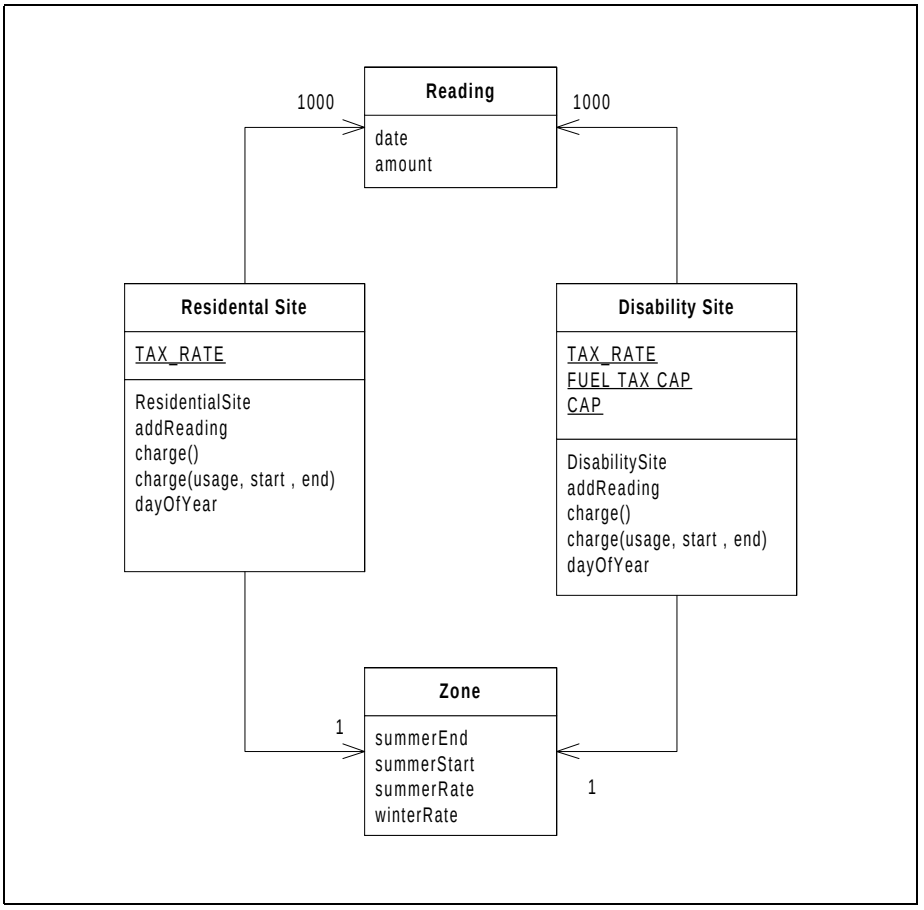

Figure 15.3: Adding the disability site

```
class DisabilitySite {
      private Reading[] _readings = new Reading[1000];
      private static final Dollars FUEL_TAX_CAP = new Dollars (0.10);
      private static final double TAX_RATE = 0.05;
      private Zone _zone;
      private static final int CAP = 200;
```

Again it has a method for adding new readings

```
public void addReading(Reading newReading)
```

```
    {
      int i;
      for (i = 0; _readings[i] != null; i++);
      _readings[i] = newReading;
    }
```

There are also two charge methods. The first is public and takes no arguments.

```
public Dollars charge()
    {
      int i;
      for (i = 0; _readings[i] != null; i++);

      int usage = _readings[i-1].amount() - _readings[i-2].amount();
      Date end = _readings[i-1].date();
      Date start = _readings[i-2].date();
      start.setDate(start.getDate() + 1);                    //set to begining of period
      return charge(usage, start, end);
    }
```

The second is the private, three argument method.

```
private Dollars charge(int fullUsage, Date start, Date end) {
      Dollars result;
      double summerFraction;
      int usage = Math.min(fullUsage, CAP);

      if (start.after(_zone.summerEnd()) || end.before(_zone.summerStart()))
          summerFraction = 0;
      else if (!start.before(_zone.summerStart()) && !start.after(_zone.summerEnd()) &&
                !end.before(_zone.summerStart()) && !end.after(_zone.summerEnd()))
          summerFraction = 1;
      else {
          double summerDays;
          if (start.before(_zone.summerStart()) || start.after(_zone.summerEnd())) {
             // end is in the summer
             summerDays = dayOfYear(end) - dayOfYear (_zone.summerStart()) + 1;
          } else {
             // start is in summer
             summerDays = dayOfYear(_zone.summerEnd()) - dayOfYear (start) + 1;
          };
          summerFraction = summerDays / (dayOfYear(end) - dayOfYear(start) + 1);
      };

      result =  new Dollars ((usage * _zone.summerRate() * summerFraction) +
          (usage * _zone.winterRate() * (1 - summerFraction)));

      result = result.plus(new Dollars (Math.max(fullUsage - usage, 0) * 0.062));
```

```
        result = result.plus(new Dollars (result.times(TAX_RATE)));

        Dollars fuel = new Dollars(fullUsage * 0.0175);
        result = result.plus(fuel);
        result = new Dollars (result.plus(fuel.times(TAX_RATE).min(FUEL_TAX_CAP)));

        return result;
    }
```

This again uses a dayOfYear method which is identical to the one for the residential site.

Our next site is called a lifeline site (Figure 15.4), used for people who claim special state dispensation due to poverty.

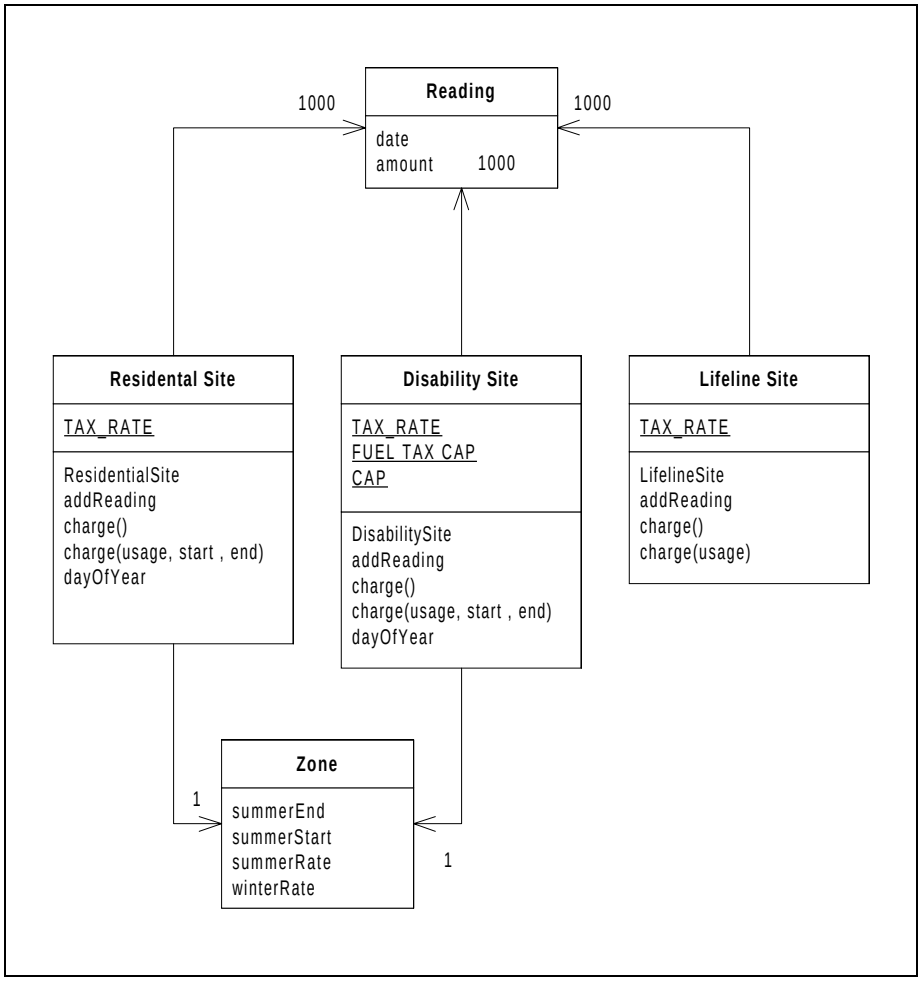

Figure 15.4: Adding the lifeline site

```
public class LifelineSite

  private Reading[] _readings = new Reading[1000];
  private static final double TAX_RATE = 0.05;
```

The method to add a reading looks different this time

```
public void addReading(Reading newReading) {
```

```
        Reading[] newArray = new Reading[_readings.length + 1];
        System.arraycopy(_readings, 0, newArray, 1, _readings.length);
        newArray[0] = newReading;
        _readings = newArray;
    }
```

Again there is a charge method with no parameters.

```
    public Dollars charge()
        {
          int usage = _readings[0].amount() - _readings[1].amount();
          return charge(usage);
        }
```

But this time the private charge method only takes one parameter

```
    private Dollars charge  (int usage) {

        double base = Math.min(usage,100) * 0.03;
        if (usage > 100) {
            base += (Math.min (usage,200) - 100) * 0.05;
        };
        if (usage > 200) {
            base += (usage - 200) * 0.07;
        };

        Dollars result = new Dollars (base);

        Dollars tax = new Dollars (
                  result.minus(new Dollars(8)).max(new Dollars (0)).times(TAX_RATE)
        );
        result = result.plus(tax);

        Dollars fuelCharge = new Dollars (usage * 0.0175);
        result = result.plus (fuelCharge);
        return result.plus (new Dollars (fuelCharge.times(TAX_RATE)));
    }
```

Our last site is the business site

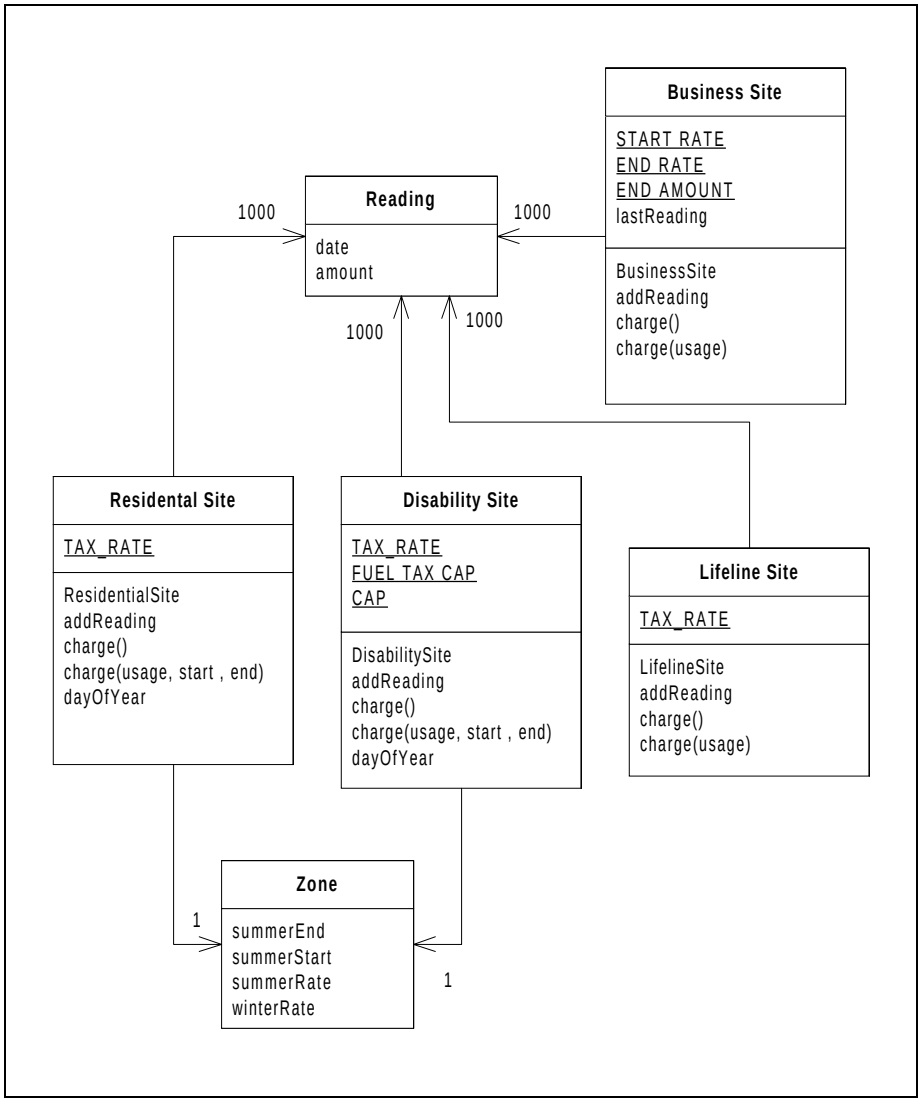

Figure 15.5: Adding the business site

```
class BusinessSite {
  private int lastReading;
  private Reading[] _readings = new Reading[1000];
  private static final double START_RATE = 0.09;
  static final double END_RATE = 0.05;
```

```
  static final int END_AMOUNT = 1000;
```

There is another variation for adding a reading

```
public void addReading(Reading newReading) {
      _readings[++lastReading] = newReading;
}
private int lastReading;
```

Again there is a no-argument charge method

```
public Dollars charge()
    {
      int usage = _readings[lastReading].amount() - _readings[lastReading -1].amount();
      return charge(usage);
    }
```

And a one-argument charge method.

```
private  Dollars charge(int usage) {
      Dollars result;

    if (usage == 0) return new Dollars(0);

  double t1 = START_RATE - ((END_RATE * END_AMOUNT) - START_RATE) / (END_AMOUNT - 1);
  double t2 = ((END_RATE * END_AMOUNT) - START_RATE) * Math.min(END_AMOUNT, usage) /
(END_AMOUNT - 1);
  double t3 = Math.max(usage - END_AMOUNT, 0) * END_RATE;

  result = new Dollars (t1 + t2 + t3);

  result = result.plus(new Dollars (usage * 0.0175));

  Dollars base = new Dollars (result.min(new Dollars (50)).times(0.07));
      if (result.isGreaterThan(new Dollars (50))) {
          base = new Dollars (base.plus(result.min(new Dollars(75)).minus(
             new Dollars(50)).times(0.06)
          ));
      };
      if (result.isGreaterThan(new Dollars (75))) {
          base = new Dollars (base.plus(result.minus(new Dollars(75)).times(0.05)));
      };
  result = result.plus(base);
  return result;

    }
```

The First Step in Refactoring

The first step in refactoring is to write the tests. How do we do this for this example? I shall use JUnit, which I described in Chapter 4.

The key behavior of classes that we are considering here is their ability to calculate charges. Thus my testing approach is based on feeding a site object with readings, calculating a charge, and checking that the charge is correct. As we have four kinds of site, we should have four fixtures that allow us to test the four different sites effectively. To start with we can pick any kind of site, I'll start with the lifeline. I set the class up with facilities to run suites as I discussed in Chapter 4. I also prepare a setUp method

```
class LifelineTester() {

  LifelineSite _subject;

  public void setUp() {
      Registry.add("Unit", new Unit ("USD"));
      new Zone ("A", 0.06, 0.07, new Date ("15 May 1997"), new Date ("10 Sep 1997")).register();
      new Zone ("B", 0.07, 0.06, new Date ("5 Jun 1997"), new Date ("31 Aug 1997")).register();
      new Zone ("C",  0.065,  0.065,  new  Date  ("5  Jun  1997"), new Date ("31 Aug
1997")).register();

      _subject = new LifelineSite();
  }
```

The registry class provides simple function to save and retrieve values in memory. The register method registers objects in the registry. That part of the program is outside the bounds of what I want to concentrate on in this exercise.

Now I have a lifeline site in my fixture, I need to prepare some test cases. It would be useful to know what the requirements are for this charge. If you have them, you should use the business requirements to help come up with the tests. Surprise, surprise the requirements documents for this code are not around (if they were ever written at all.) So here I look at the code for the charge method

```
  private Dollars charge  (int usage) {

      double base = Math.min(usage,100) * 0.03;
      if (usage > 100) {
          base += (Math.min (usage,200) - 100) * 0.05;
```

```
        };
        if (usage > 200) {
            base += (usage - 200) * 0.07;
        };

        Dollars result = new Dollars (base);

        Dollars    tax    = new    Dollars (result.minus(new    Dollars(8)).max(new Dollars
(0)).times(TAX_RATE));
        result = result.plus(tax);

        Dollars fuelCharge = new Dollars (usage * 0.0175);
        result = result.plus (fuelCharge);
        return result.plus (new Dollars (fuelCharge.times(TAX_RATE)));
    }
```

What I'm looking for here is boundary conditions to help me come up with values. Straight away it seems that a zero charge should yield a zero bill.

```
public void testZero() {
    _subject.addReading(new Reading (10, new Date ("1 Jan 1997")));
    _subject.addReading(new Reading (10, new Date ("1 Feb 1997")));
    assertEquals (new Dollars(0), _subject.charge());
}
```

Another boundary clearly lies at around 100.

```
public void test100() {
    _subject.addReading(new Reading (10, new Date ("1 Jan 1997")));
    _subject.addReading(new Reading (110, new Date ("1 Feb 1997")));
    assertEquals (new Dollars(4.84), _subject.charge());
}
```

How did I get the $4.84 figure? If I had the requirements document I would have calculated it independently (using a calculator, spreadsheet, or abacus). Since I don't and I'm refactoring I can do it by running the test, seeing what the answer is, and embedding that answer into the tests. Refactoring should not change the external behavior. I that behavior includes errors, then strictly we don't care. Of course since we are writing these tests we ought to try and verify them with the business. But we don't need to do that in order to refactor.

100 looks like a boundary, we should tackle above and below the boundary.

```
public void test99() {
    _subject.addReading(new Reading (100, new Date ("1 Jan 1997")));
    _subject.addReading(new Reading (199, new Date ("1 Feb 1997")));
```

```
        assertEquals (new Dollars(4.79), _subject.charge());
    }

    public void test101() {
        _subject.addReading(new Reading (1000, new Date ("1 Jan 1997")));
        _subject.addReading(new Reading (1101, new Date ("1 Feb 1997")));
        assertEquals (new Dollars(4.91), _subject.charge());
    }
```

We do the same for 200

```
    public void test199() {
        _subject.addReading(new Reading (10000, new Date ("1 Jan 1997")));
        _subject.addReading(new Reading (10199, new Date ("1 Feb 1997")));
        assertEquals (new Dollars(11.6), _subject.charge());
    }
    public void test200() {
        _subject.addReading(new Reading (0, new Date ("1 Jan 1997")));
        _subject.addReading(new Reading (200, new Date ("1 Feb 1997")));
        assertEquals (new Dollars(11.68), _subject.charge());
    }
    public void test201() {
        _subject.addReading(new Reading (50, new Date ("1 Jan 1997")));
        _subject.addReading(new Reading (251, new Date ("1 Feb 1997")));
        assertEquals (new Dollars(11.77), _subject.charge());
    }
```

Another boundary is the eight dollars for taxes. We seem to be either side of that boundary with our calculations. The question is whether it is worth the effort to get closer to that boundary. The value is the likelihood of finding bugs we otherwise would not catch. A bit of algebra tells me that the $8 cut off for taxes is exactly at the 200 usage boundary, so the tests at the 200 boundary catch both. So I will finish with a large value.

```
    public void testMax() {
        _subject.addReading(new Reading (0, new Date ("1 Jan 1997")));
        _subject.addReading(new Reading (Integer.MAX_VALUE, new Date ("1 Feb 1997")));
        assertEquals (new Dollars(1.9730005336E8), _subject.charge());
    }
```

Another boundary is what happens if there are no readings?

```
    public void testNoReadings() {
        assertEquals (new Dollars(0), _subject.charge());
    }
```

The answer is that I get a null pointer exception. In this situation there are two possibilities. One is that the null pointer exception is a bug. In this case I put the test to one side as bug finding test. I would fix that

bug later on during the refactoring exercise. The other possibility is that it is the expected behavior, in which case I need to test to ensure it continues to happen. This would look like

```
public void testNoReadings() {
    try {
        _subject.charge();
        assert(false);
    } catch (NullPointerException e) {}
}
```

The other sites are dealt with along similar lines. In each case look for the conditions that look like boundaries and concentrate the tests there. This process yields tester classes for each site. I also add an over all class to pull all the tests together.

```
public static Test suite() {
  TestSuite result = new TestSuite();
  result.addTest(LifelineTester.suite());
  result.addTest(BusinessTester.suite());
  result.addTest(DisabilityTester.suite());
  result.addTest(ResidentialTester.suite());
  return result;
}
```

Starting a Hierarchy of Sites

I have no idea which two classes to start with in this case, so I will pick two at random. Well maybe not entirely at random, these two classes do have two fields in common, readings and zone, so there is perhaps a bit more common ground here. Usually I do like to start with those that have the most data in common. But I don't think it matters in this case.

Pulling up the zone and readings fields

The zone field is really very similar. Each class has a field of type Zone that is set in the constructor. At this point I want to know how the zone is used. To do this I do a find for "_zone" in both files and look to see where it used. As I do this I can see that the zone is not modified after the constructor has set it up, and it used quite a lot in the charge method.

From this I feel ready to create a superclass Site and to move the zone field to it. The first step is to declare the new superclass.

```
class Site {}
```

Now I make it the superclass of residential and disability

```
class ResidentialSite extends Site
class DisabilitySite extends Site
```

With that I can use *Pull Up Field (270* to move the zone field up to the superclass

```
class Site...
    protected Zone _zone;
```

By making the zone field protected I can ensure that the subclasses can still work with it as before. Some people feel strongly that fields should not be protected, rather they should be private. If you feel like this you use *Self Encapsulate Field (184)* at this point and provide a *Getting Method* for it. I'm not so concerned and so I'll leave it protected, at least for the moment.

Both constructors use this field so I might as well use*Pull up Constructor Body (273* on those, one at a time

```
class Site...
    Site (Zone zone) {
      _zone = zone;
    }
class ResidentialSite...
  public ResidentialSite (Zone zone) {
      super (zone);
  };
```

Once I've tested it to make sure everything is all right, I do the same for `DisabilitySite`.

```
class DisabilitySite ...
  public DisabilitySite (Zone zone) {
      super (zone);
  }
```

The next field to move is the readings field. Again I do a find to look at how it is used in the two classes. In both cases it is initialized to a 1000 element array. The `addReading` method finds the last reading in the array and adds a reading to the end. The charge method pulls some readings out of the array. I can begin safely using *Pull Up Field (270).*

```
class Site {

    Site (Zone zone) {
      _zone = zone;
    }

    protected Zone _zone;
    protected Reading[] _readings = new Reading[1000];
```

I then remove it from the subclasses and test.

While the zone field did not have much behavior to deal with, the readings field does, and this seems a reasonable next target for my attention. Although one uses a for loop, and the other a while loop, they both add the new reading to the next null spot in the array. I can therefore pick one of them to move to site, and get rid of both of them.

```
class Site {

  public void addReading(Reading newReading) {
      int i = 0;
      while (_readings[i] != null) i++;
      _readings[i] = newReading;
  }
```

That worked fine, but I don't find the `addReading` method clearly indicates what it is doing. It is finding the first non null index in the array, and adding the new reading at that point. I would like the code to say that more clearly. I can do that by creating an *Intention Revealing Method* for finding the first unused index in the readings array.

```
class Site {

  public void addReading(Reading newReading) {
      _readings[ firstUnusedReadingsIndex() ] = newReading;
  }

  private int firstUnusedReadingsIndex () {
      int i = 0;
      while (_readings[i] != null) i++;
      return i;
  }
```

I'm not completely happy about the name "`firstUnusedReadingsIndex`" but the way the method works seems clearer now. Figure 15.6 summarizes where we are at this point. The zone and readings fields have been

pushed up to our new site class, and the `addReadings` method has also been moved up and simplified.

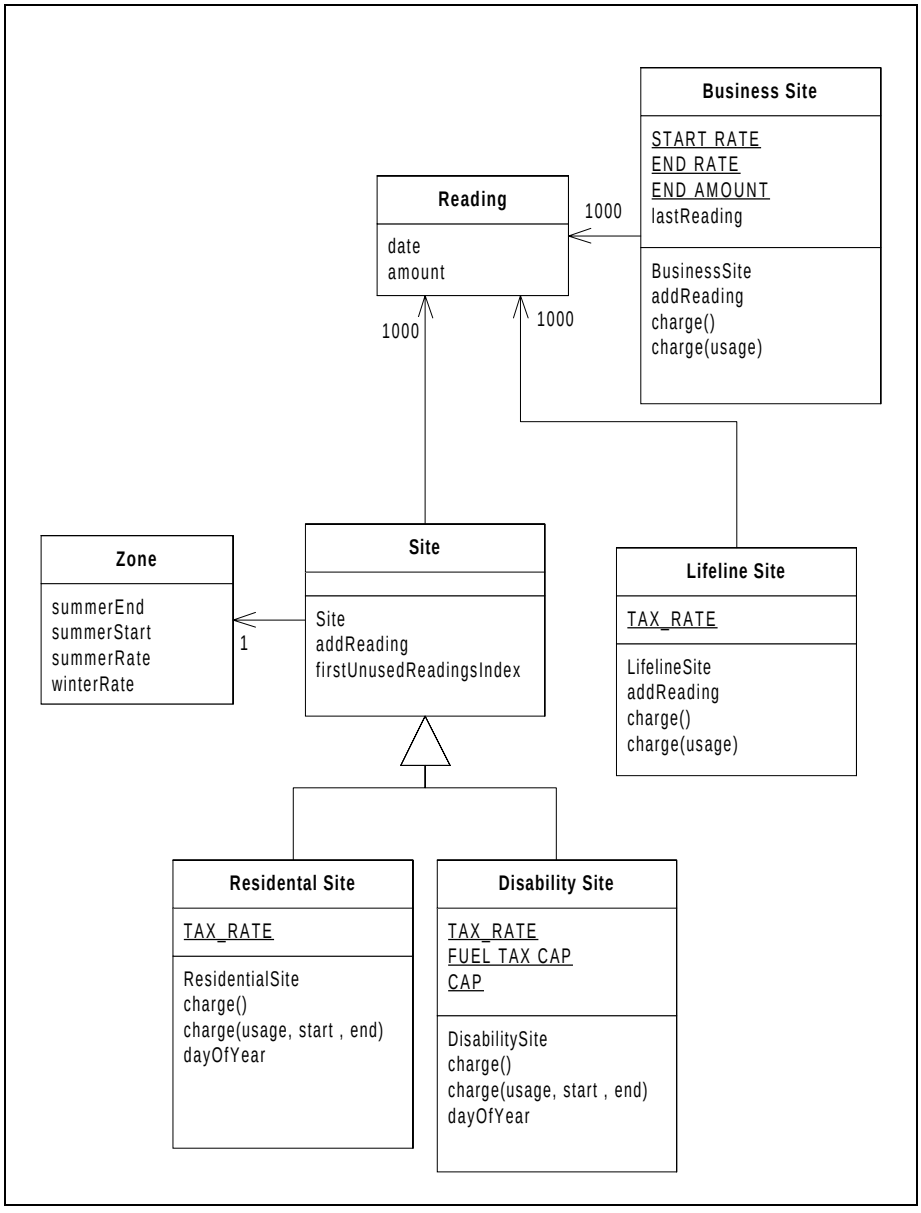

Figure 15.6: After pulling up the data for residential and disability sites

Decomposing the no-arg charge method

While I'm looking at the use of the readings field the public charge method seems a good next item to work on. It looks identical in the two classes (I guess it was cut and pasted between them). So I can move it to site.

```
class Site {

  public Dollars charge() {
      // find last reading
      int i = 0;
      while (_readings[i] != null) i++;

      int usage = _readings[i-1].amount() - _readings[i-2].amount();
      Date end = _readings[i-1].date();
      Date start = _readings[i-2].date();
      start.setDate(start.getDate() + 1);                    //set to begining of period
      return charge(usage, start, end);
  }
```

The highlighted line above causes a problem because it calls a method that is only defined by the subclasses. To get this to work I need to define an abstract method for charge (int, Date, Date) and loosened the access control on charge (int, Date, Date) to protected so that it can be overridden.

```
abstract class Site {

  public Dollars charge() {
      // find last reading
      int i = 0;
      while (_readings[i] != null) i++;

      int usage = _readings[i-1].amount() - _readings[i-2].amount();
      Date end = _readings[i-1].date();
      Date start = _readings[i-2].date();
      start.setDate(start.getDate() + 1);                    //set to begining of period
      return charge(usage, start, end);
  }

  abstract protected Dollars charge(int fullUsage, Date start, Date end);

}
```

The access control has to be loosened on the subclasses too. I've left the variable names in the abstract method declaration even though they are not necessary since I think they help communicate the use of the

method. Since I now have an abstract method the class is abstract, as it should be.

Looking at the method, I can see that the much of the action lies in working with the i-1 and i-2 readings. These are the last reading in the readings array, and the next to last reading. This can be made much clearer. I can start by defining a method to get me the last reading.

```
private Reading lastReading() {
    return _readings[firstUnusedReadingsIndex() - 1];
};
```

This allows me to modify charge to:

```
public Dollars charge() {
      // find last reading
      int i = 0;
      while (_readings[i] != null) i++;

      int usage = lastReading().amount() - _readings[i-2].amount();
      Date end = lastReading().date();
      Date start = _readings[i-2].date();
      start.setDate(start.getDate() + 1);                    //set to begining of period
      return charge(usage, start, end);
    }
```

I can do the same for the previous reading.

```
public Dollars charge() {
    int usage = lastReading().amount() -  previousReading().amount();
    Date end = lastReading().date();
    Date start = previousReading().date();
    start.setDate(start.getDate() + 1);                    //set to begining of period
    return charge(usage, start, end);
}

private Reading previousReading() {
    return _readings[firstUnusedReadingsIndex() - 2];
}
```

The code is now much easier to read. However it will now run slower, because every call to `lastReading` or `previousReading` will cause the array to be traversed through the loop in `firstUnusedReadingIndex`. Since I'm currently refactoring I choose to ignore that. I will fix it with an optimization later if it is important (I can easily cache the `firstUnusedReadingsIndex` in a field).

While I'm on this method there is a couple of other things I will do. I can make the usage calculation into its own method.

```
public Dollars charge() {
    Date end = lastReading().date();
    Date start = previousReading().date();
    start.setDate(start.getDate() + 1);                    //set to begining of period
    return charge( lastUsage() , start, end);
}

private int lastUsage() {
    return lastReading().amount() - previousReading().amount();
};
```

Since how the end is calculated is pretty clear, I could also remove its temp. I'm not sure whether its not clearer, however, to leave it there to indicate the role it is playing the method call. If I was in Smalltalk I would probably remove it because Smalltalk has keywords for its parameters, which communicate the role of each item in the method call. Java has positional parameters, so I find I'm more inclined to use a

temp to show the role of the parameter. In the end these choices come

down to what you and your colleagues find the easiest to understand.

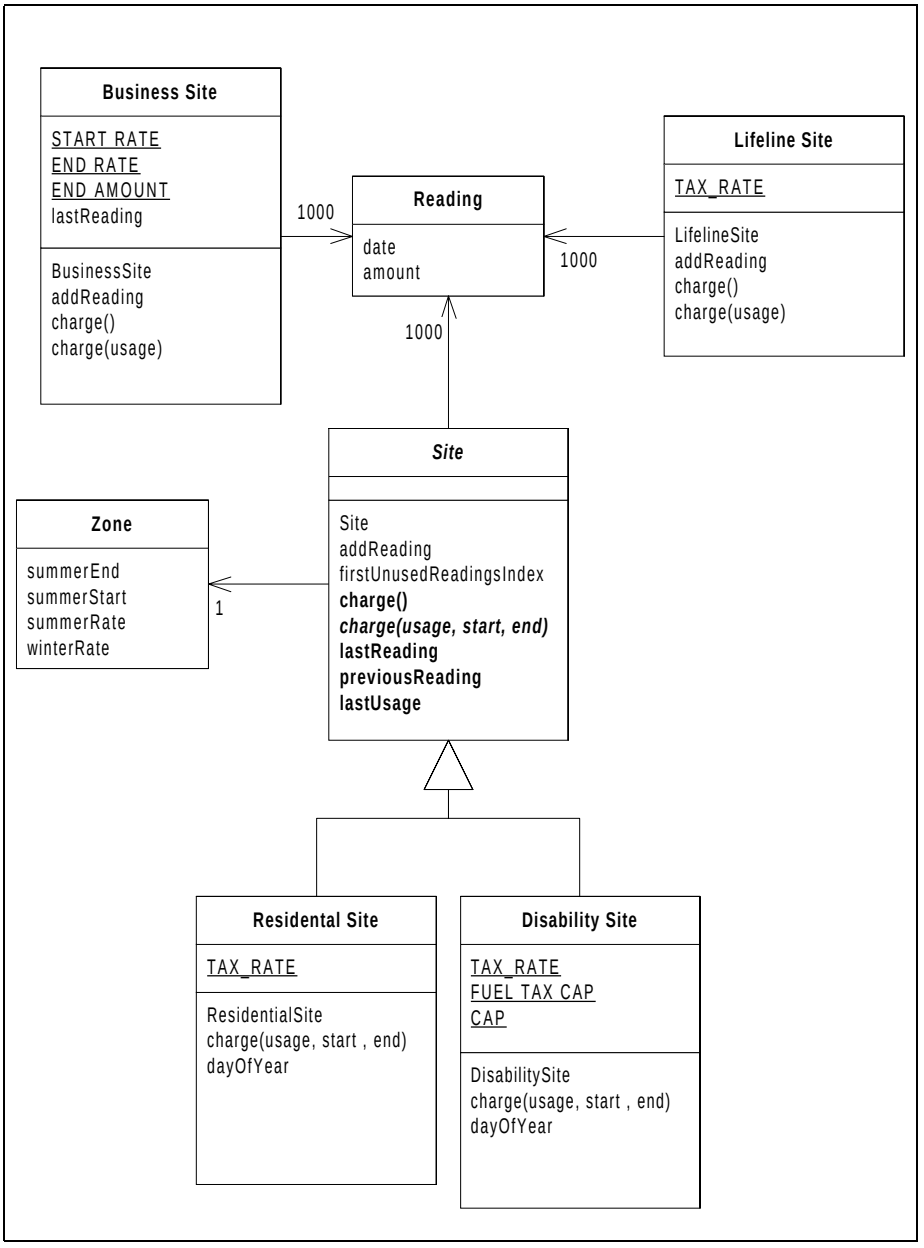

Figure 15.7: After reorganizing the charge field

Figure 15.7 shows where we are now. Decomposing the charge field has added several method to the site.

Extracting a next day method

I would like to do something about the start date. Ideally I would like a statement like `start = previousReading().date()++` , but that isn't legal Java. An alternative would be to do something like `start = previousReading().date().nextDay()`, but there is no method along those lines in the class library (when I did this I was using 1.0 Java). I'm treating the Java class library as a given, something I cannot refactor. So my next step is to use *Create Foreign Method (176* . This will create a method that should be on Date, but instead I put it in my own class. Then I can write `start = nextDay (previousReading().date())` .

The way I would normally come up with a next day is to return a new date object which is one day later. But this is not quite the same as what the current code does:

```
Date start = previousReading().date();
    start.setDate(start.getDate() + 1);                    //set to begining of period
```

The `setDate` method actually changes the day of the current date. To me this as a very odd thing to do. I think of dates as values, things much like integers and reals. Such things are immutable, there is no notion of altering the number 3 to be some other value, instead you alter the variable to point to a different number. This is different to people. If a person has a name and you change her name, she is still the same person. There is nothing you can change about a number in that way.

Make a clear distinction betwee ***Reference Objects*** *(e.g. Person) and* ***Value Objects*** *(e.g. Date). Value objects should always be immutable.*

Quantity is like that too, there is nothing you can change about the quantity $3. Thus in any implementation of Quantity I make the amount and the unit immutable: I set them in the constructor and provide no way of changing them. In Java Date is not like that, it has all these set methods, yet I think of Date as a value object and thus it should not be changed.

The problem with allowing value objects to have mutable data is that it leads to odd bugs. If you say:

```
x = new Quantity (3,"USD");
y = x;
```

you don't expect anything that you can do to y to change x. But with this date that is not true. I can go:

```
x = new Date ("1 Jan 97");
y = x;
y.setDate(2);
```

and x will change as well.

This has actually happened in the program I am refactoring. The tests check that the value returned by charge() is correct for the various cases. There is a side effect of this process, a changing of the dates in the readings. The tests did not catch this.

One of the reasons the tests did not catch this is because the #$%^@ who wrote the tests forgot a useful testing rule. When you test a query method, try testing it a few times in row. If the tester had done that, he would have found the error because subsequent invocations o charge() would have returned a different value. (Authors are supposed to be omniscient, so I won't tell you who wrote the tests.

This yields a couple of further points. One is, of course, that tests don't catch everything and your security in refactoring (as in any development) is only as good as your tests. But it is no good bleating on about how tests don't guarantee correctness, tests are still your best weapon, and you can incrementally improve tests as you go on. The second point is that refactoring can help you find errors, as you spot odd little things like that, much as code inspection does.

Refactoring is an active form of code inspection.

When I spot a bug like that I immediately update the tests so that they catch this problem, and any obvious similar problems. A repeated call to charge() does that nicely.

The next decision is to how to deal with it. Well I still need come up with a `nextDay` method. I would like to do it by copying a date and adding one to the new date with `setDate`, but Date is not cloneable, so I

can't copy it with clone. I can easily get around that by using an appropriate constructor.

```
class Site...
  public Dollars charge() {
      Date end = lastReading().date();
      Date start = nextDay(previousReading().date()) ;
      return charge(lastUsage(), start, end);
  }

  private Date nextDay (Date arg) {
      // foreign method - should be in Date
      Date result = new Date (arg.getTime());
      result.setDate(result.getDate() + 1);
      return result;
  }
```

A general principle I use in Java is not use the set... methods on date, since they can easily lead to bugs of this nature. I do use it in `nextDay`, however, because it is a foreign method, and should really be on Date. Foreign methods have to treated with caution. They are not unreasonable if there only one or two of them, but they can easily get out of control. If you find you have created more than two foreign methods it's time to use *Create Extension (178)* instead — I'll come to that later.

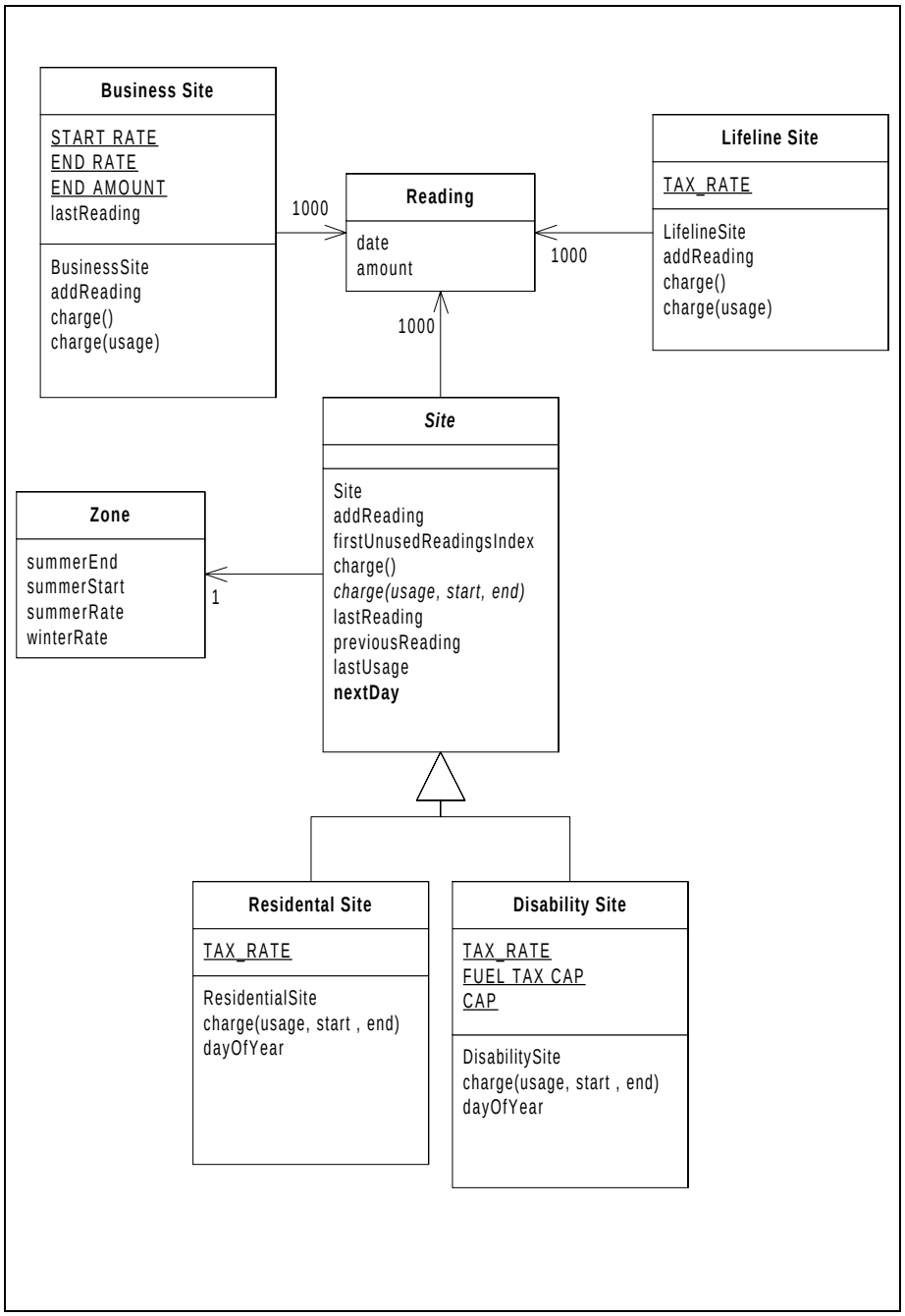

Figure 15.8: After creating the next day method

Figure 15.8 captures our current state of play, with the new foreign method.

Refactoring So Far

1) Starting a Hierarchy of Sites

So far we have create a superclass site and brought residential and dis ability sites under it. We've pulled up the data and some simple behav ior. To really get at the heart of simplifying this program we have to start working on the more complex behavior buried in the charge method. It is good, however, to begin with a few simple things to begin understanding the program. I also like to move around data before I work on behavior, especially complex behavior like these charge methods.

Simplifying the Charge Methods

But now it is time for the big problem: the three-argument charge methods. Let me refresh you on what they look like.

```
class ResidentialSite...
  protected Dollars charge(int usage, Date start, Date end) {

      Dollars result;
      double summerFraction;

      // Find out how much of period is in the summer
      if (start.after(_zone.summerEnd()) || end.before(_zone.summerStart()))
          summerFraction = 0;
      else if (!start.before(_zone.summerStart()) && !start.after(_zone.summerEnd()) &&
                !end.before(_zone.summerStart()) && !end.after(_zone.summerEnd()))
          summerFraction = 1;
      else {    // part in summer part in winter
          double summerDays;
          if (start.before(_zone.summerStart()) || start.after(_zone.summerEnd())) {
             // end is in the summer
             summerDays = dayOfYear(end) - dayOfYear (_zone.summerStart()) + 1;
          } else {
             // start is in summer
             summerDays = dayOfYear(_zone.summerEnd()) - dayOfYear (start) + 1;
          };
```

```
            summerFraction = summerDays / (dayOfYear(end) - dayOfYear(start) + 1);
        };

        result =  new Dollars ((usage * _zone.summerRate() * summerFraction) +
            (usage * _zone.winterRate() * (1 - summerFraction)));


        result = result.plus(new Dollars (result.times(TAX_RATE)));

        Dollars fuel = new Dollars(usage * 0.0175);
        result = result.plus(fuel);

        result = new Dollars (result.plus(fuel.times(TAX_RATE)));

        return result;

    }

class DisabilitySite...
    protected Dollars charge(int fullUsage, Date start, Date end) {
        Dollars result;
        double summerFraction;
        int usage = Math.min(fullUsage, CAP);

        if (start.after(_zone.summerEnd()) || end.before(_zone.summerStart()))
            summerFraction = 0;
        else if (!start.before(_zone.summerStart()) && !start.after(_zone.summerEnd()) &&
                !end.before(_zone.summerStart()) && !end.after(_zone.summerEnd()))
            summerFraction = 1;
        else {
            double summerDays;
            if (start.before(_zone.summerStart()) || start.after(_zone.summerEnd())) {
               // end is in the summer
               summerDays = dayOfYear(end) - dayOfYear (_zone.summerStart()) + 1;
            } else {
               // start is in summer
               summerDays = dayOfYear(_zone.summerEnd()) - dayOfYear (start) + 1;
            };
            summerFraction = summerDays / (dayOfYear(end) - dayOfYear(start) + 1);
        };

        result =  new Dollars ((usage * _zone.summerRate() * summerFraction) +
            (usage * _zone.winterRate() * (1 - summerFraction)));

        result = result.plus(new Dollars (Math.max(fullUsage - usage, 0) * 0.062));


        result = result.plus(new Dollars (result.times(TAX_RATE)));

        Dollars fuel = new Dollars(fullUsage * 0.0175);
        result = result.plus(fuel);
```

```
        result = new Dollars (result.plus(fuel.times(TAX_RATE).min(FUEL_TAX_CAP)));

        return result;
    }
```

Extracting summerFraction

Its easy for my eyes to glaze over with big methods like that, but as I run my eye over them I can see some similarities. The code I've high lighted is identical (I'd bet good beer money that it was cut and pasted). This code is there to determine the value of the temp summerFraction. I can thus use *Extract Method (114)*, and put the extracted method into Site.

The first step in extracting a method is to identify any variables referenced in the code that are local in scope to the routine. You can do this by eye, or by using a find on the local scope variables. The local scope variables are temps and parameters. In this start, end, and summerFraction are the ones used by the candidate extract. Next I need to consider if any of these values are altered. I treat altering a parameter in Java as the height of bad taste, but I check anyway. The only one of the three to be altered is summerFraction. If you have one locally scoped variable that is altered, and that variable is used outside the extracted code, then it makes sense to make the extracted code have a return value. The temp sounds like a good name for the new method: summerFraction

Each locally scoped variable that is used needs to be passed into the new method as a parameter. SummerFraction is not altered between its initialization and the entry into to candidate extraction, so we don't need to pass it in, I can just initialize it within the new method. So the new method will have two parameters for the start and end dates.

I begin with just one class, I picked residential site at random. I create the new method in that class (I'll pull it up to site later). I first copy the candidate extraction, put it into a new method, and compile it. Then I remove the candidate extraction from the source method and replace it with a method that calls the new method. In this case I find all uses o summerFraction and replace them with references to the new method. Then I test to see if I have broken anything.

```
class ResidentialSite...
  protected Dollars charge(int usage, Date start, Date end) {
```

```
        Dollars result;

        result =  new Dollars ((usage * _zone.summerRate() *   summerFraction(start,end) ) +
            (usage * _zone.winterRate() * (1 -   summerFraction(start,end) )));


        result = result.plus(new Dollars (result.times(TAX_RATE)));

        Dollars fuel = new Dollars(usage * 0.0175);
        result = result.plus(fuel);

        result = new Dollars (result.plus(fuel.times(TAX_RATE)));

        return result;

    }


    private double summerFraction(Date start, Date end) {
        double summerFraction;
        if (start.after(_zone.summerEnd()) || end.before(_zone.summerStart()))
            summerFraction = 0;
        else if (!start.before(_zone.summerStart()) && !start.after(_zone.summerEnd()) &&
                  !end.before(_zone.summerStart()) && !end.after(_zone.summerEnd()))
            summerFraction = 1;
        else {    // part in summer part in winter
            double summerDays;
            if (start.before(_zone.summerStart()) || start.after(_zone.summerEnd())) {
               // end is in the summer
               summerDays = dayOfYear(end) - dayOfYear (_zone.summerStart()) + 1;
            } else {
               // start is in summer
               summerDays = dayOfYear(_zone.summerEnd()) - dayOfYear (start) + 1;
            };
            summerFraction = summerDays / (dayOfYear(end) - dayOfYear(start) + 1);
        };
        return summerFraction;
    }
```

All went well, so now I use *Pull Up Method (271)* to move the new method up to site and compile. When I compile it complains that the method dayOfYear is not in site (it was defined on the subclass). I could look into moving that up too, but I will do that later. For now I'll make an appropriate abstract method in site (which means loosening the access control in the subclasses)

```
class Site...
  abstract int dayOfYear(Date arg);
```

I test again and all is good. Now I remove the similar looking code from disability site's charge method and replace with the appropriate call to summerFraction.

```
class DisabilitySite...
  protected Dollars charge(int fullUsage, Date start, Date end) {
      Dollars result;
      int usage = Math.min(fullUsage, CAP);


      result =  new Dollars ((usage * _zone.summerRate() *   summerFraction(start, end) ) +
          (usage * _zone.winterRate() * (1 -  summerFraction(start, end) )));

      result = result.plus(new Dollars (Math.max(fullUsage - usage, 0) * 0.062));


      result = result.plus(new Dollars (result.times(TAX_RATE)));

      Dollars fuel = new Dollars(fullUsage * 0.0175);
      result = result.plus(fuel);
      result = new Dollars (result.plus(fuel.times(TAX_RATE).min(FUEL_TAX_CAP)));

      return result;
    }
```

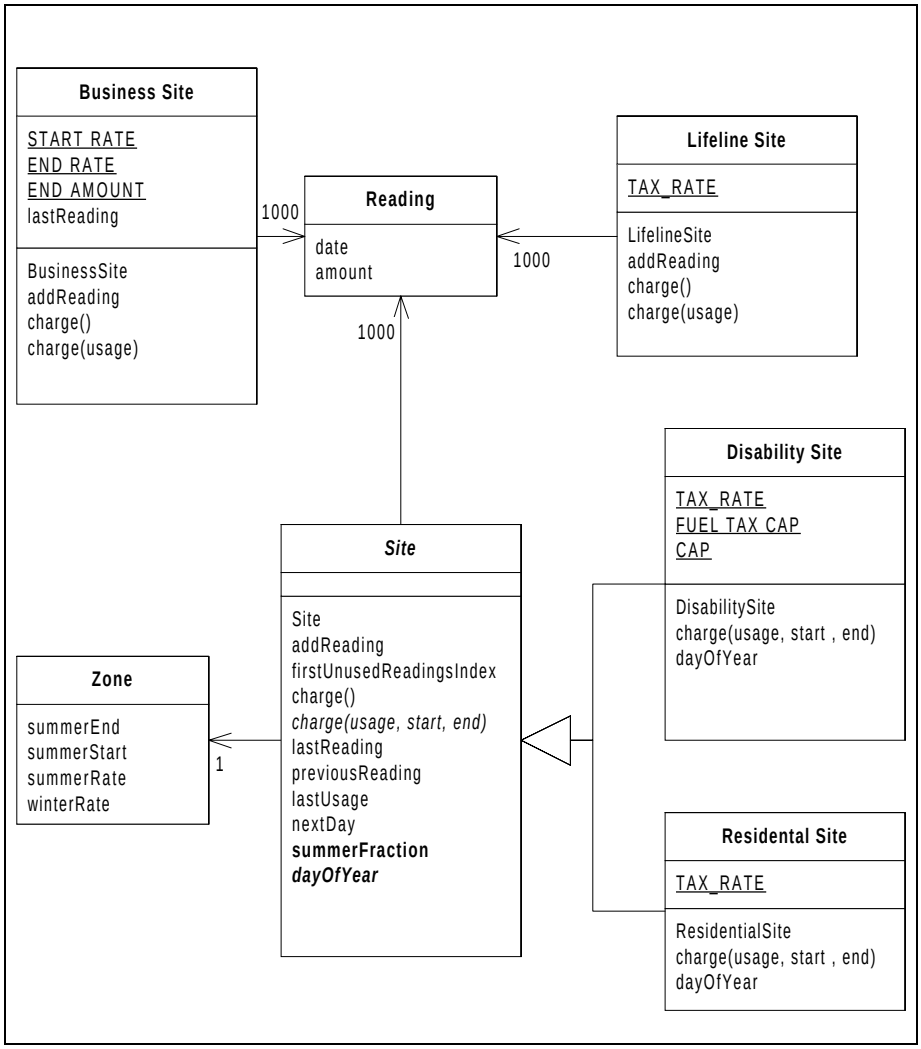

Figure 15.9: After extracting `summerFraction`

Extracting a large method like that really helps the readability of the code. If someone has used cut and paste when they shouldn't, you can often do this.

Figure 15.9 shows the program with the summer fraction moved up to site.

Extracting the fuel and tax calculations

Now it's easier to see the differences between the methods. The dis ability case caps the amount sent in via a parameter. The later sections of code do not use this capped value, so I decided to make my next steps down there. Both methods have the two following lines in common

```
Dollars fuel = new Dollars(fullUsage * 0.0175);
result = result.plus(fuel);
```

I can use *Inline Temp (121)* to remove the temp fuel with a new method.

```
class DisabilitySite
  protected Dollars charge(int fullUsage, Date start, Date end) {
      Dollars result;
      int usage = Math.min(fullUsage, CAP);


      result =  new Dollars ((usage * _zone.summerRate() * summerFraction(start, end)) +
          (usage * _zone.winterRate() * (1 - summerFraction(start, end))));

      result = result.plus(new Dollars (Math.max(fullUsage - usage, 0) * 0.062));


      result = result.plus(new Dollars (result.times(TAX_RATE)));

      result = result.plus(fuelCharge(fullUsage)) ;
      result = new Dollars

                      (result.plus( fuelCharge(fullUsage) .times(TAX_RATE).min(FUEL_TAX_CAP)));

      return result;
  }

class Site...
  protected Dollars fuelCharge() {
      return new Dollars(lastUsage() * FUEL_CHARGE_RATE);
  }



  protected static final double FUEL_CHARGE_RATE = 0.0175;
```

As I did that I also used *Replace Magic Number with Symbolic Constant (210)* to name the fuel charge rate, and then*Pull Up Method (271)* on the `fuelCharge` method and *Pull Up Field (270)* on the new constant.

At first sight it may seem odd to remove the temporary variable. Again all I am doing is causing the `fuelCharge` to be calculated twice

instead of once. Again my argument would be that it really does no harm, I can cache it later when I optimize. (You might say "why bother" and leave the temp in place. Keep that in mind during the next step.)

Now I will go at the next line, which is different in two methods.

```
class ResidentialSite {
  protected Dollars charge(int usage, Date start, Date end) {

      Dollars result;

      result =  new Dollars ((usage * _zone.summerRate() * summerFraction(start,end)) +
          (usage * _zone.winterRate() * (1 - summerFraction(start,end))));


      result = result.plus(new Dollars (result.times(TAX_RATE)));

      result = result.plus(fuelCharge(usage));

      result = new Dollars (result.plus(fuelCharge(usage).times(TAX_RATE)));

      return result;

    }

class DisabilitySite {
  protected Dollars charge(int fullUsage, Date start, Date end) {
      Dollars result;
      int usage = Math.min(fullUsage, CAP);


      result =  new Dollars ((usage * _zone.summerRate() * summerFraction(start, end)) +
          (usage * _zone.winterRate() * (1 - summerFraction(start, end))));

      result = result.plus(new Dollars (Math.max(fullUsage - usage, 0) * 0.062));


      result = result.plus(new Dollars (result.times(TAX_RATE)));

      result = result.plus(fuelCharge(fullUsage));
      result = new Dollars
                 (result.plus(fuelCharge(fullUsage).times(TAX_RATE).min(FUEL_TAX_CAP)));

      return result;
    }
```

Although they are different, they both do a similar thing. They both add the taxes for the fuel charge to the result. In this case I will use

Extract Method (114) again, but this time I will have two methods with the same signature, one in each class.

```
class ResidentialSite {
  protected Dollars charge(int usage, Date start, Date end) {

      Dollars result;

      result =  new Dollars ((usage * _zone.summerRate() * summerFraction(start,end)) +
          (usage * _zone.winterRate() * (1 - summerFraction(start,end))));


      result = result.plus(new Dollars (result.times(TAX_RATE)));

      result = result.plus(fuelCharge(usage));

      result = result.plus( fuelChargeTaxes (usage));

      return result;

  }

  protected Dollars fuelChargeTaxes(int usage) {
      return new Dollars (fuelCharge(usage).times(TAX_RATE));
  }


class DisabilitySite {
  protected Dollars charge(int fullUsage, Date start, Date end) {
      Dollars result;
      int usage = Math.min(fullUsage, CAP);


      result =  new Dollars ((usage * _zone.summerRate() * summerFraction(start, end)) +
          (usage * _zone.winterRate() * (1 - summerFraction(start, end))));

      result = result.plus(new Dollars (Math.max(fullUsage - usage, 0) * 0.062));


      result = result.plus(new Dollars (result.times(TAX_RATE)));

      result = result.plus(fuelCharge(fullUsage));
      result = result.plus( fuelChargeTaxes (fullUsage));

      return result;
  }

  protected Dollars fuelChargeTaxes(int usage) {
      return new Dollars (fuelCharge(usage).times(TAX_RATE).min(FUEL_TAX_CAP));
  }
```

My master plan is now coming apparent to me. Charge looks like it may turn out to be a *Template Method* [Gang of Four], if so I want to decompose it and turn it into a sequence of identical calls to methods that vary polymorphically using *Form Template Method (289)*. I haven't really looked to see if this will work for the whole method yet, but I might as well start that way. If two methods in different classes seems to do the same thing, I might as well give them the same signature.

You might also now see an advantage in why I didn't hang on to the fuel temporary variable earlier. If I had done that then that would be another locally scoped temp that I would have had to pass into the `fuelChargeTaxes` method. All those temps and parameters make life awkward. By replacing the temp with a method, I no longer pass it in, I just call the method. If later I need to cache the value of the method, that cache will work whenever the method is used.

Can you see that I could have done the same thing with the call to the three parameter charge method in the first place? Usage, start, and end can all be replaced with appropriate methods. I will do that later.

(You might wonder, why didn't I do that earlier. Here is a confession that you shouldn't really see in a book like this. I have always followed the habit of eliminating parameters when I refactor, but I never really knew why, it was just a habit. Only in writing this chapter did I realize why I did it.)

For now I will get back to working on this charge method. I have a game plan to make this a template method, where else can I get ready to do this? An obvious place is the taxes.

```
class Site {

  protected Dollars taxes(Dollars arg) {
      return new Dollars (arg.times(TAX_RATE));
  }

  protected static final double TAX_RATE = 0.05;

class ResidentialSite {
  protected Dollars charge(int usage, Date start, Date end) {

      Dollars result;

      result =  new Dollars ((usage * _zone.summerRate() * summerFraction(start,end)) +
          (usage * _zone.winterRate() * (1 - summerFraction(start,end))));
```

```
        result = result.plus( taxes(result));

        result = result.plus(fuelCharge(usage));

        result = result.plus(fuelChargeTaxes(usage));

        return result;

    }
class DisabilitySite {
    protected Dollars charge(int fullUsage, Date start, Date end) {
        Dollars result;
        int usage = Math.min(fullUsage, CAP);


        result =  new Dollars ((usage * _zone.summerRate() * summerFraction(start, end)) +
            (usage * _zone.winterRate() * (1 - summerFraction(start, end))));

        result = result.plus(new Dollars (Math.max(fullUsage - usage, 0) * 0.062));


        result = result.plus( taxes(result));

        result = result.plus(fuelCharge(fullUsage));
        result = result.plus(fuelChargeTaxes(fullUsage));

        return result;
    }
```

Now I'm at the point where I've extracted the calculations of taxes and fuel charges, leaving the position of Figure 15.10.

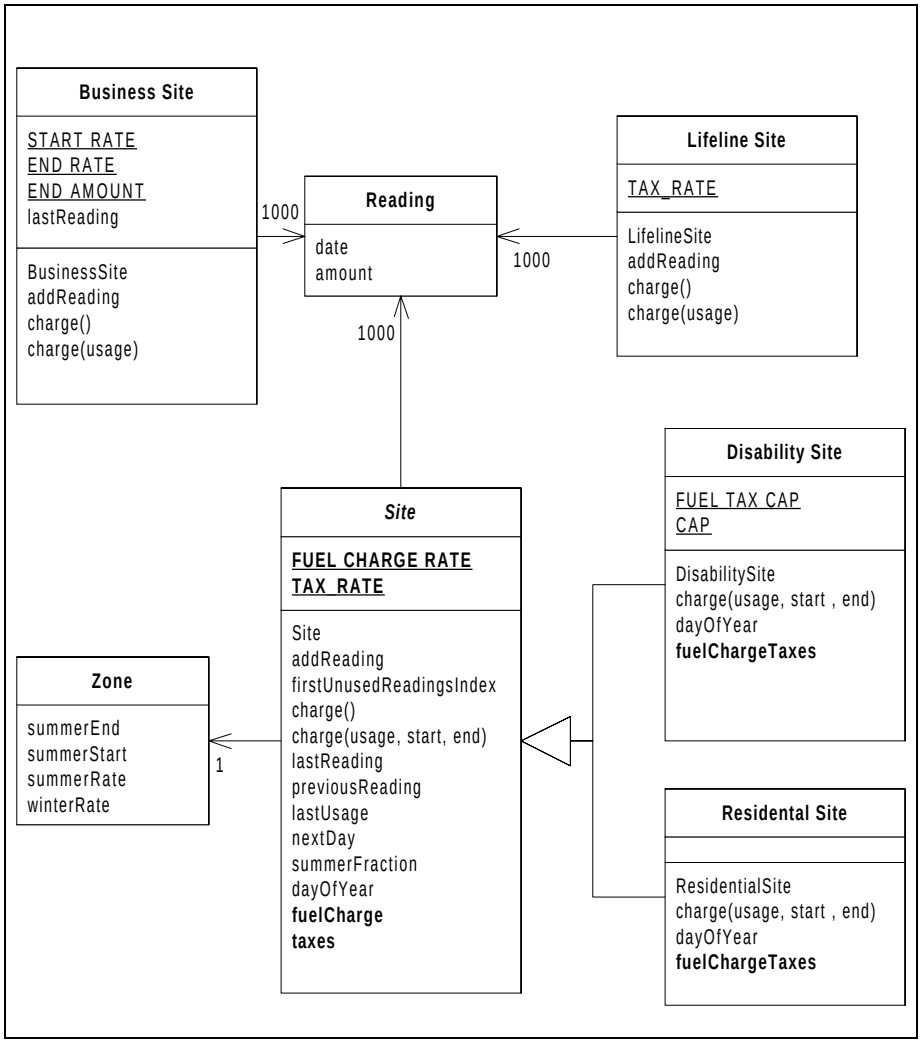

Figure 15.10: After extracting the fuel and tax calculations

Turning the charge method into a template method

The next step takes a little thought. An obvious next move would be to do something along these lines to disability site

```
protected Dollars charge(int fullUsage, Date start, Date end) {
    Dollars result;
```

```
        int usage = Math.min(fullUsage, CAP);


        result =  baseCharge(usage);

        result = result.plus(overCapCharge (fullUsage - usage));


        result = result.plus(taxes(result));

        result = result.plus(fuelCharge(fullUsage));
        result = result.plus(fuelChargeTaxes(fullUsage));

        return result;
    }
```

But if I do that, the charge method will have to be different between the disability site and the residential site, foiling my master plan o making charge a template method. Although I only formed this master plan in the middle of refactoring this method, I still like it and want it to succeed (template methods are good). So I look for a refactoring that fits in with it. Such a move would mean extracting the highlighted code in the following.

```
class DisabilitySite {
  protected Dollars charge(int fullUsage, Date start, Date end) {
      Dollars result;
      int usage = Math.min(fullUsage, CAP);


      result =  new Dollars ((usage * _zone.summerRate() * summerFraction(start, end)) +
          (usage * _zone.winterRate() * (1 - summerFraction(start, end))));

      result = result.plus(new Dollars (Math.max(fullUsage - usage, 0) * 0.062));


      result = result.plus(taxes(result));

      result = result.plus(fuelCharge(fullUsage));
      result = result.plus(fuelChargeTaxes(fullUsage));

      return result;
  }
```

If I do this I could make a single polymorphic method baseCharge. The result looks like this.

```
class DisabilitySite ...
  protected Dollars charge(int fullUsage, Date start, Date end) {
      Dollars result;
```

```
        result =  baseCharge(fullUsage, start, end);
        result = result.plus(taxes(result));
        result = result.plus(fuelCharge(fullUsage));
        result = result.plus(fuelChargeTaxes(fullUsage));

        return result;
    }

    protected Dollars baseCharge(int arg, Date start, Date end) {
        int cappedUsage = Math.min(arg, CAP);

        Dollars result;
        result =  new Dollars (
            (cappedUsage * _zone.summerRate() * summerFraction(start, end)) +
            (cappedUsage * _zone.winterRate() * (1 - summerFraction(start, end)))
        );
        result = result.plus(new Dollars (Math.max(arg - cappedUsage, 0) * 0.062));
        return result;
    }
```

For ResidentialSite it looks like this.

```
class ResidentialSite ...
    protected Dollars charge(int usage, Date start, Date end) {
        Dollars result;
        result = baseCharge(usage, start, end);
        result = result.plus(taxes(result));
        result = result.plus(fuelCharge(usage));
        result = result.plus(fuelChargeTaxes(usage));
        return result;
    }

    protected Dollars baseCharge(int usage, Date start, Date end) {
        return new Dollars ((usage * _zone.summerRate() * summerFraction(start,end)) +
            (usage * _zone.winterRate() * (1 - summerFraction(start,end))));
    }
```

Now the two charge methods are identical, so I use *Pull Up Method (271)* to move them (it?) up to the site class.

```
abstract class Site {
    protected Dollars charge(int usage, Date start, Date end) {
        Dollars result;
        result = baseCharge(usage, start, end);
        result = result.plus(taxes(result));
        result = result.plus(fuelCharge(usage));
        result = result.plus(fuelChargeTaxes(usage));
        return result;
    }
    abstract protected Dollars baseCharge (int usage, Date start, Date end);
    abstract protected Dollars fuelChargeTaxes(int usage);
```

I also have to create abstract methods for `baseCharge` and `fuelChargeTaxes`.

Now I have a statement of how you create a charge that reads like documentation, even without comments. Furthermore the commonality and differences are clear between the two kinds of site, making it eas ier to change them in the future, and also to add new kinds of site. I'll be testing that capability later as I blend the other two kinds of site into the hierarchy. Figure 15.11 shows the classes at this point.

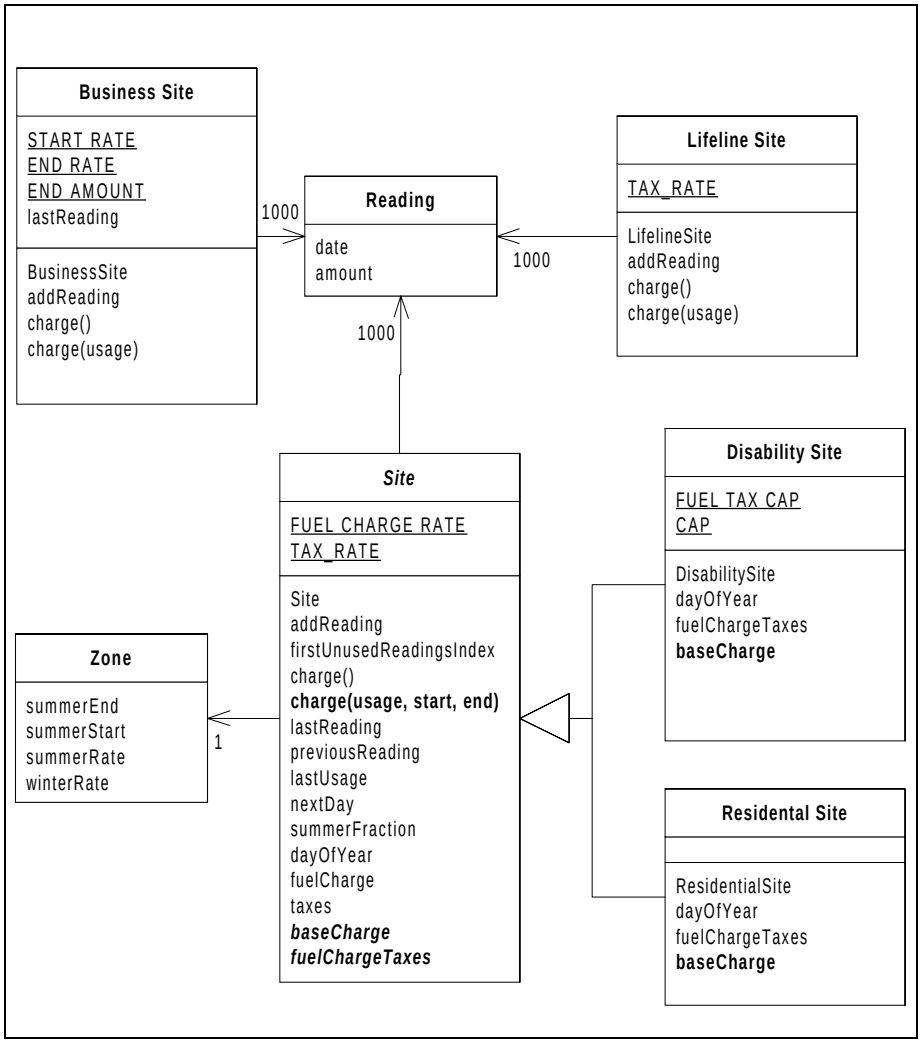

Figure 15.11: After turning charge into a template method

Removing the usage argument

Having two charge methods, is not ideal, especially since the 3-arg charge method is now so simple. Some further refactoring will simplify the logic further. I should be able to reduce the size of the parameter list for the various methods. The first target is the usage argument,

which already has a method for it: lastUsage. I can use *Inline Temp (121)* to get rid of the temp, and then *Replace Parameter with Method (245)*.

To do this I need to find every reference to the string usage. If the refer ence is in a method call or declaration I remove it. If it is in some body code, I replace it with lastUsage. This causes quite a lot of changes, but with the find and replace tools in editors it is actually quite easy to do. Once I'd done all that, the code looked like this.

```
class Site...
  public Dollars charge() {
      Date end = lastReading().date();
      Date start = nextDay(previousReading().date());
      return charge(start, end);
  }
  protected Dollars charge(Date start, Date end) {
      Dollars result;
      result = baseCharge(start, end);
      result = result.plus(taxes(result));
      result = result.plus( fuelCharge() );
      result = result.plus( fuelChargeTaxes() );
      return result;
  }
  abstract protected Dollars  baseCharge (Date start, Date end) ;
  abstract protected Dollars  fuelChargeTaxes() ;
  protected Dollars fuelCharge() {
      return new Dollars( lastUsage() * 0.0175);
  }

class DisabilitySite...
  protected Dollars baseCharge(Date start, Date end) {
      int cappedUsage = Math.min( lastUsage() , CAP);

      Dollars result;
      result =  new Dollars (
          (cappedUsage * _zone.summerRate() * summerFraction(start, end)) +
          (cappedUsage * _zone.winterRate() * (1 - summerFraction(start, end)))
      );

      result = result.plus(new Dollars (Math.max(  lastUsage() - cappedUsage, 0) * 0.062));
      return result;
    }
  protected Dollars fuelChargeTaxes() {
      return new Dollars ( fuelCharge() .times(TAX_RATE).min(FUEL_TAX_CAP));
  }



class ResidentialSite {
  protected Dollars baseCharge(Date start, Date end) {
```

```
        return new Dollars (( lastUsage() * _zone.summerRate() * summerFraction(start,end)) +
            (lastUsage() * _zone.winterRate() * (1 - summerFraction(start,end))));
    }
    protected Dollars fuelChargeTaxes() {
        return new Dollars ( fuelCharge().times(TAX_RATE));
    }
```

Replacing start and end with a date range

The other parameters to work on are the start and end dates. I can do this quite easily as they are only used in determining the summerFraction. I could replace them with methods such as lastPeriodStart and lastPeriodEnd, but there is something else I would like to do.

Starts and ends tend to come in pairs, and that pair has a meaning all of itself. We are talking here about a period of time. The start and the end are a classic data clump and should instead be attributes of a new object, which I can create with *Extract Component (166)*.

The whole object for any pair of values marked start and end is a*Range* [Fowler, AP]. For a pair of dates in a typed language I will use a specific class DateRange. This is a common class for me, and I often have one lying around. In this case I will build it up from scratch as I need it. It starts simply as an encapsulated record.

```
import java.util.Date;
class DateRange {
  public DateRange(Date start, Date end) {
      _start = start;
      _end = end;
  }
  public Date end() {
      return _end;
  }
  public Date start() {
      return _start;
  }
  private Date _end;
  private Date _start;
}
```

Now I have the class I need to put it to use. A good candidate is summerFraction

```
    private double summerFraction(Date start, Date end) {
        double summerFraction;
        if (start.after(_zone.summerEnd()) || end.before(_zone.summerStart()))
```

```
        summerFraction = 0;
    else if (!start.before(_zone.summerStart()) && !start.after(_zone.summerEnd()) &&
              !end.before(_zone.summerStart()) && !end.after(_zone.summerEnd()))
        summerFraction = 1;
    else {    // part in summer part in winter
        double summerDays;
        if (start.before(_zone.summerStart()) || start.after(_zone.summerEnd())) {
           // end is in the summer
           summerDays = dayOfYear(end) - dayOfYear (_zone.summerStart()) + 1;
        } else {
           // start is in summer
           summerDays = dayOfYear(_zone.summerEnd()) - dayOfYear (start) + 1;
        };
        summerFraction = summerDays / (dayOfYear(end) - dayOfYear(start) + 1);
    };
    return summerFraction;
}
```

I can write a method to create the range object

```
class Site...
  public DateRange lastPeriod() {
      return new DateRange (nextDay(previousReading().date()), lastReading().date());
  }
```

and then *Replace Parameter with Method (245).*

```
class Site ...
  protected double summerFraction() {
      double result;          // I also changed this name to make it fit my normal usage
      if (
            lastPeriod().start() .after(_zone.summerEnd()) ||
            lastPeriod().end() .before(_zone.summerStart())
      )
          result = 0;
      else if (
            !lastPeriod().start() .before(_zone.summerStart()) &&
            !lastPeriod().start() .after(_zone.summerEnd()) &&
            !lastPeriod().end() .before(_zone.summerStart()) &&
            !lastPeriod().end() .after(_zone.summerEnd())
      )
          result = 1;
      else {    // part in summer part in winter
          double summerDays;
          if (
                  lastPeriod().start() .before(_zone.summerStart()) ||
                  lastPeriod().start() .after(_zone.summerEnd())
          ) {
             // end is in the summer
             summerDays = dayOfYear( lastPeriod().end() ) -
                                                dayOfYear (_zone.summerStart()) + 1;
          } else {
```

```
            // start is in summer
            summerDays = dayOfYear(_zone.summerEnd()) -
                                                dayOfYear ( lastPeriod().start() ) + 1;
        };
        result = summerDays / (dayOfYear(  lastPeriod().end() ) -
                                        dayOfYear( lastPeriod().start() ) + 1);
    };
    return result;
  }

class ResidentialSite...
  protected Dollars baseCharge() {
    return new Dollars ((lastUsage() * _zone.summerRate() *   summerFraction() ) +
        (lastUsage() * _zone.winterRate() * (1 -  summerFraction() )));
  }
class DisabilitySite...
  protected Dollars baseCharge() {
    int cappedUsage = Math.min(lastUsage(), CAP);

    Dollars result;
    result =  new Dollars ((cappedUsage * _zone.summerRate() *   summerFraction() ) +
        (cappedUsage * _zone.winterRate() * (1 -  summerFraction() )));

    result = result.plus(new Dollars (Math.max(lastUsage() - cappedUsage, 0) * 0.062));
    return result;
  }
```

Merging the charge methods

Now I can merge the two charge methods

```
class Site...
public Dollars charge() {
      Dollars result;
      result = baseCharge();
      result = result.plus(taxes(result));
      result = result.plus(fuelCharge());
      result = result.plus(fuelChargeTaxes());
      return result;
  }
```

The taxes method doesn't really need an argument, so again I *Replace Parameter with Method (245* .

```
  public Dollars charge() {
      Dollars result;
      result = baseCharge();
      result = result.plus(taxes());
      result = result.plus(fuelCharge());
      result = result.plus(fuelChargeTaxes());
      return result;
```

```
}

protected Dollars taxes() {
    return new Dollars (baseCharge().times(TAX_RATE));
}
```

So all we have is a simple sum

```
public Dollars charge() {
    return baseCharge().plus(taxes()).plus(fuelCharge()).plus(fuelChargeTaxes());
}
```

It doesn't look quite as good as it would if we could overload +, but it is still pretty clear.

Refactoring So Far

1) Starting a Hierarchy of Sites

2) Simplifying the Charge Methods

With this I have given the charge calculation a clear structure, one that allows me to separate out the things that are common between the classes from those parts that are different. This makes it much easier to modify the calculations and to add new ones. You can easily see which changes will affect both classes, and understand the steps in building the charge. The true test will come when we bring business and lifeline sites into the structure. I don't expect this structure to match exactly (unless I'm very lucky), but it should be recognizable.

I didn't start this exercise intending to do this. It was only by doing the early decomposition that I understood enough about the method to see that a template would work here.

You do the early refactorings to learn more about the program, these set you up for later ones that really simplify the structure.

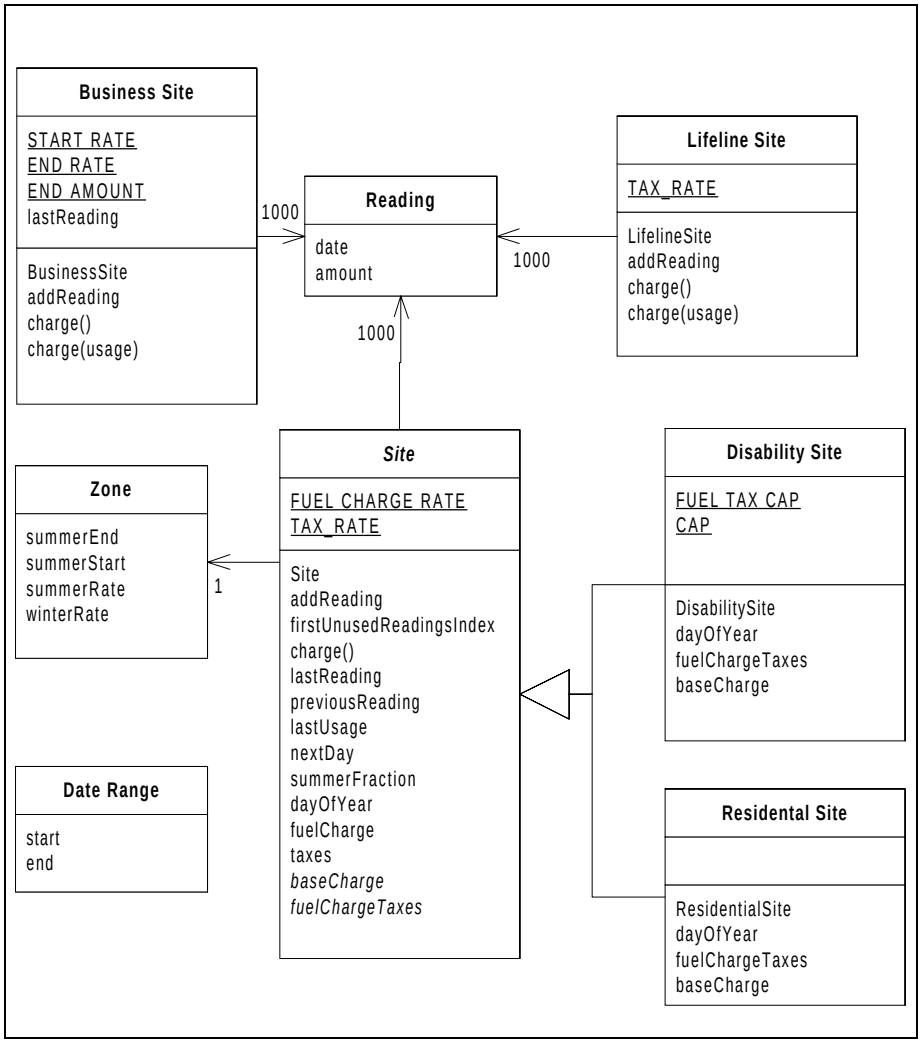

Figure 15.12: After merging the charge methods

Decomposing Site's Long Methods

Now I'm salivating to see if the other two sites will fit the template. However before I do that, I'd like to do some further simplification on

the two classes I have. The more I do this, the more I understand the hierarchy as it currently stands, and the easier it will be to add new classes. Of course this means I'll make decisions which I will have to change when I bring in the other classes. But by making things simpler now I'll make those later changes easier.

> *Don't let fear of the future stop you from doing a refactoring now. Refactorings aren't difficult to change, and learning you gain repays the effort.*

The next thing I do is start looking though the methods of the three classes in the browser. I'm sniffing for bad smells, and get a good whif of over-long methods. The first method that catches my nose is summerFraction

```
class Site ...
  protected double summerFraction() {
      double result;
      if (
          lastPeriod().start().after(_zone.summerEnd()) ||
          lastPeriod().end().before(_zone.summerStart())
          )
          result = 0;
      else if (
          !lastPeriod().start().before(_zone.summerStart()) &&
          !lastPeriod().start().after(_zone.summerEnd()) &&
          !lastPeriod().end().before(_zone.summerStart()) &&
          !lastPeriod().end().after(_zone.summerEnd())
          )
          result = 1;
      else {    // part in summer part in winter
          double summerDays;
          if (
             lastPeriod().start().before(_zone.summerStart()) ||
             lastPeriod().start().after(_zone.summerEnd())
             ) {
             // end is in the summer
             summerDays = dayOfYear(lastPeriod().end()) -
                    dayOfYear (_zone.summerStart()) + 1;
          } else {
             // start is in summer
             summerDays = dayOfYear(_zone.summerEnd()) -
                  dayOfYear (lastPeriod().start()) + 1;
          };
          result = summerDays / (dayOfYear(lastPeriod().end()) -
```

```
                dayOfYear(lastPeriod().start()) + 1);
        };
        return result;
    }
```

Decomposing summerFraction's conditionals

There are a lot of conditionals here, which are hard to fit on a page. I will use *Decompose Conditional (136)*, a useful cure for this disease. I do this by extracting a method from each part of the conditional statement.

```
protected double summerFraction() {
    double result;
    if (isLastPeriodOutsideSummer() )
        result = 0;
    else if (
              !lastPeriod().start().before(_zone.summerStart()) &&
              !lastPeriod().start().after(_zone.summerEnd()) &&
              !lastPeriod().end().before(_zone.summerStart()) &&
              !lastPeriod().end().after(_zone.summerEnd())
            )
        result = 1;
    else {    // part in summer part in winter
        double summerDays;
        if (
           lastPeriod().start().before(_zone.summerStart()) ||
           lastPeriod().start().after(_zone.summerEnd())
        ) {
           // end is in the summer
           summerDays = dayOfYear(lastPeriod().end()) -
                   dayOfYear (_zone.summerStart()) + 1;
        } else {
           // start is in summer
           summerDays = dayOfYear(_zone.summerEnd()) -
                   dayOfYear (lastPeriod().start()) + 1;
        };
        result = summerDays / (dayOfYear(lastPeriod().end()) -
                dayOfYear(lastPeriod().start()) + 1);
    };
    return result;
 }

protected boolean isLastPeriodOutsideSummer() {
    return lastPeriod().start().after(_zone.summerEnd()) ||
        lastPeriod().end().before(_zone.summerStart());
}
```

As I look at the isLastPeriodOutsideSummer method I think that I should be able to simply this considerably by using the periods as whole objects. In this method I really want to know if the lastPeriod has no overlap with the zone's summer. The first step would be to get zone to be able to tell you its summer as a date range.

```
class Zone ...
  public DateRange summer() {
      return new DateRange (_summerStart, _summerEnd);
  }
```

I can then rewrite isLastPeriodOutsideSummer

```
  protected boolean isLastPeriodAllSummer() {
      return lastPeriod().start().after(_zone.summer().end()) ||
          lastPeriod().end().before(_zone.summer().start());
  }
```

As such that is no improvement. But now I can write a disjoint method for DateRange,

```
class DateRange
public boolean disjoint(DateRange arg) {
      return arg.start().after(_end) || arg.end().before(_start);
  }
```

and make isLastPeriodOutsideSummer much simpler

```
  protected boolean isLastPeriodOutsideSummer() {
      return _zone.summer().disjoint(lastPeriod());
  }
```

In fact I don't think the body of isLastPeriodOutsideSummer is any less communicative than the method name, so I will use *Inline Method (120)* to inline it back into summerFraction.

```
  protected double summerFraction() {
      double result;
      if (_zone.summer().disjoint(lastPeriod()) )
          result = 0;
      else if (
                !lastPeriod().start().before(_zone.summerStart()) &&
                !lastPeriod().start().after(_zone.summerEnd()) &&
                !lastPeriod().end().before(_zone.summerStart()) &&
                !lastPeriod().end().after(_zone.summerEnd())
              )
          result = 1;
      else {    // part in summer part in winter
          double summerDays;
          if (
             lastPeriod().start().before(_zone.summerStart()) ||
```

```
            lastPeriod().start().after(_zone.summerEnd())
        ) {
            // end is in the summer
            summerDays = dayOfYear(lastPeriod().end()) -
                    dayOfYear (_zone.summerStart()) + 1;
        } else {
            // start is in summer
            summerDays = dayOfYear(_zone.summerEnd()) -
                    dayOfYear (lastPeriod().start()) + 1;
        };
        result = summerDays / (dayOfYear(lastPeriod().end()) -
                dayOfYear(lastPeriod().start()) + 1);
    };
    return result;
}
```

I get the strong feeling that I can make this whole method into a method on DateRange, I will keep that goal in mind as I continue to decompose the method. I'll do the next conditional. This is pretty complicated and as my brain starts to hurt from figuring out what it is doing I look at the result. The result is effectively saying that the lastPeriod is entirely within the summer. Thus if I write a contains method on DateRange

```
class DateRange ...
  public boolean contains(DateRange arg) {
      return arg.start().after(_start) && arg.end().before(_end);
  }
```

then I can rewrite it as

```
protected double summerFraction() {
    double result;
    if (_zone.summer().disjoint(lastPeriod()))
        result = 0;
    else if (_zone.summer().contains(lastPeriod())  )
        result = 1;
    else {    // part in summer part in winter
        double summerDays;
        if (
            lastPeriod().start().before(_zone.summerStart()) ||
            lastPeriod().start().after(_zone.summerEnd())
        ) {
            // end is in the summer
            summerDays = dayOfYear(lastPeriod().end()) -
                    dayOfYear (_zone.summerStart()) + 1;
        } else {
            // start is in summer
            summerDays = dayOfYear(_zone.summerEnd()) -
                    dayOfYear (lastPeriod().start()) + 1;
```

```
            };
            result = summerDays / (dayOfYear(lastPeriod().end()) -
                    dayOfYear(lastPeriod().start()) + 1);
        };
        return result;
    }
```

As I look at the next bit of code I see that the code is doing one thing i the start is within the summer period, and another if the end is within the summer. What happens if neither are within the summer, that is if the summer is contained within the last period. The code is written with the assumption that that cannot happen (billing periods are monthly, shorter than a summer). Again none of the test cases probe it (I need to find a new tester). I add a test case to probe it, and indeed it fails. As I refactor this code I will fix that bug.

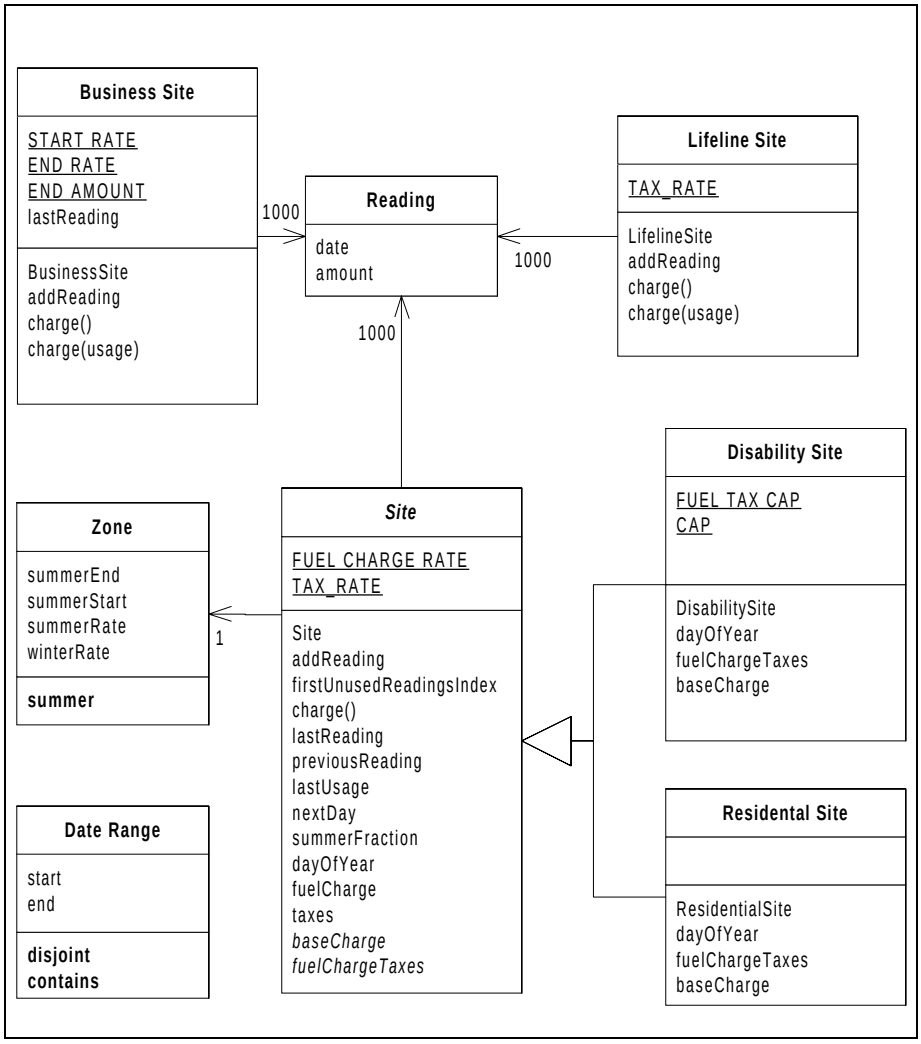

Figure 15.13: After beginning to decompose `summerFraction`

I feel I'm heading in the right direction (Figure 15.13 shows the position so far), but to make further progress I need to step back a bit. If I think about this as a manipulation of date ranges then I can solve it by asking what the length of the last period is, and what the length of the overlap range between the last period and the summer is. I can substi-

tute this algorithm for the existing one. Before I can do this I need to add some further behavior to date range to determine the intersection of two date ranges and to determine the length of a date range.

Extending Date

To determine the length of a date range I need to be able to subtract two dates. Sadly the Date class in java does not give me this feature, hence this use of `dayOfYear` which is really a foreign method. I need to make a new series of foreign methods on date. The amount of foreign methods I will need is too much and will soon get out of hand. What I really need to do is to fix the Date class. Since I cannot get at the Date class, I can deal with it by using *Create Extension (178).* I can do this is two ways, either by subclassing Date or by making a*decorator* [Gang of Four] for Date. The easiest way seems to be to use a subclass. I begin with defining just the suitable constructors.

```
import java.util.Date;
class MfDate extends Date {

  public MfDate (Date arg) {
      super (arg.getTime());
    }

  public MfDate (String dateString) {
      super (dateString);
  };
}
```

Whenever I create an extension I provide a constructor that takes what I'm extending as an argument. This makes it easy for code to switch between the two if needed. I've also provided the constructor for String since I know I use that in my test code.

At this point I have to decide: do I just use the extension where I need it (by converting to `MfDate` within certain methods) or do I use it everywhere. If I only use it where I need it I will be forever jumping back and forth between `Date` and `MfDate` and getting confused. If I change it everywhere I really do need to change it everywhere. Well my code is not too big and I do have a global find capability. So I use the global find to find every occurrence of `Date`.

There is only one point where there is a problem.

```
class Site
```

```
private Date nextDay (Date arg) {
    // foreign method - should be in Date
    Date result = new Date (arg.getTime())  ;
    result.setDate(result.getDate() + 1);
    return result;
}
```

If course I do need to move nextDay to MfDate, now that I have created the extension. But for the moment I will fix the problem, compile and test, and then move it. I don't want too long a gap between tests. I could solve this by defining a constructor that takes a long argument, but what I really want here is a copy. So I would do better by giving MfDate a copy method.

```
class MfDate extends Date implements Cloneable...
  public Object clone() {
      try {
          return super.clone();
      } catch (CloneNotSupportedException e) {
          //should not happen
          throw new InternalError(e.toString());
      }
  }

class Site...
  private MfDate nextDay (MfDate arg) {
      // foreign method - should be on Date
      MfDate result = (MfDate) arg.clone();
      result.setDate(result.getDate() + 1);
      return result;
  }
```

That works, but I don't like the fact that the client of MfDate has to be responsible for the downcast. I can avoid that by using *Encapsulate Downcast (258)*.

```
class MfDate
  public MfDate copy() {
          return (MfDate) clone();
  }

class Site
  private MfDate nextDay (MfDate arg) {
      // foreign method - should be on Date
      MfDate result = arg.copy();
      result.setDate(result.getDate() + 1);
      return result;
  }
```

With that the system recompiles using MfDates everywhere.

Now I might as well move the foreign method to its proper home using *Move Method (160)*. First I create it in the new place, adjusting the code as necessary.

```
class MfDate
  public MfDate nextDay() {
      MfDate result = copy();
      result.setDate(result.getDate() + 1);
      return result;
  }
```

Next I find all the references to the original method and replace them with calls to the new method. There are two ways to do this in Java environments. You can either do a search and replace, or you can remove the old method, and let the compiler find where you need to make changes. There was one reference.

```
class Site
  public DateRange lastPeriod() {
      return new DateRange ( previousReading().date().nextDay()  , lastReading().date());
  }
```

Whichever way you do the replacement make sure you remove the original method before you compile and test. Otherwise if you miss one you can get some funny results. Also look to see if the method is redefined anywhere within the inheritance hierarchy, the compiler won't catch that either.

To do the date subtraction I need to use the `dayOfYear` method currently in residential site and disability site. These methods are identical and are both foreign methods that should really by on MfDate. So I will move them over. First I move `DayOfYear` to `MfDate`.

```
class MfDate {
int dayOfYear() {
      int result;
      switch (getMonth()) {
          case 0:
             result = 0;
             break;
          case 1:
             result = 31;
             break;
          case 2:
             result = 59;
             break;
          case 3:
             result = 90;
```

```
            break;
        case 4:
            result = 120;
            break;
        case 5:
            result = 151;
            break;
        case 6:
            result = 181;
            break;
        case 7:
            result = 212;
            break;
        case 8:
            result = 243;
            break;
        case 9:
            result = 273;
            break;
        case 10:
            result = 304;
            break;
        case 11:
            result = 334;
            break;
        default :
            throw new IllegalArgumentException();
    };
    result += getDate();

    //check leap year
    if ((getYear()%4 == 0) && ((getYear() % 100 != 0) || ((getYear()  + 1900) % 400 == 0))) {
        result++;
    };

    return result;
  }
```

Still an ugly method, but I will deal with that later. Now I find all places that call it, as it turns out they are within summerFraction.

```
protected double summerFraction() {
    double result;
    if (_zone.summer().disjoint(lastPeriod()))
        result = 0;
    else if (_zone.summer().contains(lastPeriod()))
        result = 1;
    else {    // part in summer part in winter
        double summerDays;
        if (
            lastPeriod().start().before(_zone.summerStart()) ||
```

```
            lastPeriod().start().after(_zone.summerEnd())
        ) {
            // end is in the summer
            summerDays = lastPeriod().end().dayOfYear()  -
                    _zone.summerStart().dayOfYear()  + 1;
        } else {
            // start is in summer
            summerDays = _zone.summerEnd().dayOfYear()  -
                    lastPeriod().start().dayOfYear()  + 1;
        };
        result = summerDays / (lastPeriod().end().dayOfYear()  -
                lastPeriod().start().dayOfYear()  + 1);
    };
    return result;
  }
```

Then I remove dayOfYear from ResidentialSite and DisabilitySite, remove the abstract method declaration I had put on site, compile and test.

Now I can get ready to do the date subtraction. I do, however, have a little problem. The date manipulations in summerFraction assume that periods do not cross year boundaries. A date subtraction routine should work with multiple years. Do I implement for the more general case, or just the simpler case present in the system I am refactoring? It seems like a dangerous assumption to me, but for the moment I will leave it as it is (with a suitable check). I can come back and fix that later. (The leap years make it not completely trivial.)

```
class MfDate...
  public int minus(MfDate arg) {
      requireSameYear (arg);                                  // TK fix this to cross years
      return dayOfYear() - arg.dayOfYear();
  }

  private void requireSameYear(MfDate arg) {
      if (getYear() != arg.getYear())
          throw new IllegalArgumentException ("Arguments must be in same year");
  }
```

Now I can define length for aDateRange

```
class DateRange
  public int length() {
      return _end.minus(_start) + 1;
  }
```

The next thing I need to do is to define an intersection method for DateRange.

```
public DateRange intersection(DateRange arg) {
    MfDate newStart = (_start.after(arg.start())) ?
        _start :
        arg.start();
    MfDate newEnd = (_end.before(arg.end())) ?
        _end:
        arg.end();
    return new DateRange(newStart, newEnd);
}
```

With that I am now ready (finally) to work on summerFraction. Figure 15.14 shows the what the date and date range classes look like.

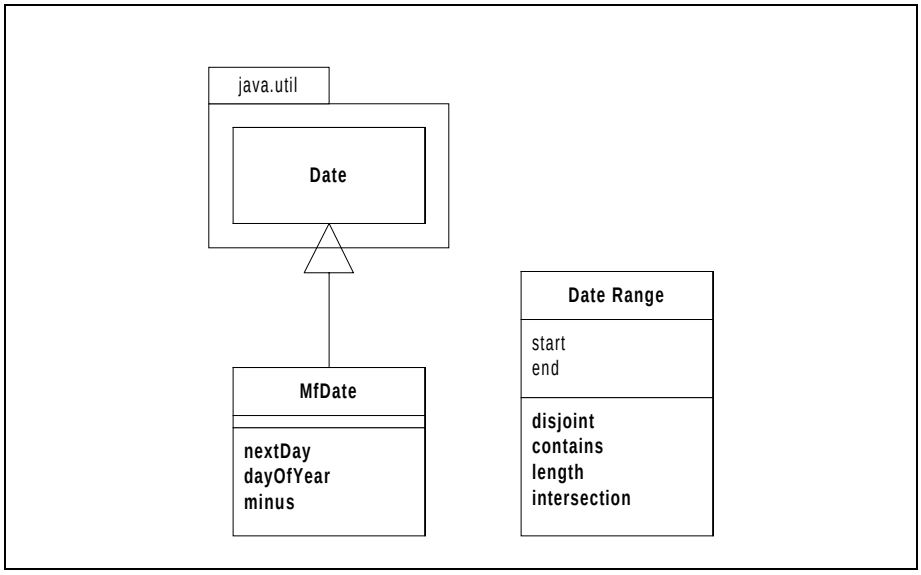

Figure 15.14: Date and Date Range classes

Substituting an algorithm for summerFraction

I shall change summerFraction by using *Substitute Algorithm (132)*. This refactoring can often be a bigger jump than most refactorings. Rather than make small, behavior preserving transformations, I am making a larger transformation which I hope will be behavior preserving. Errors do occur, and I find this refactoring can often lead to debugging.

Here is my first shot at the new summerFraction

```
protected double summerFraction() {
```

```
        DateRange periodInSummer = lastPeriod().intersection(_zone.summer());
        return periodInSummer.length() / lastPeriod().length();
    }
```

It didn't work, several test cases failed. In most refactoring my reaction to such a failure is to back out and try a smaller step, but *Substitute Algorithm (132)* doesn't really lend itself to smaller steps. One thing I can do to help debugging is to reintroduce the old version `summerFraction` under the name `oldSummerFraction`. I can then use it to help me debug by alerting me whenever the result of `summerFraction` and `oldSummerFraction` differ.

```
    protected double summerFraction() {
        DateRange periodInSummer = lastPeriod().intersection(_zone.summer());
        double result = periodInSummer.length() / lastPeriod().length();
        if (result != oldSummerFraction())
            System.out.println ("new sf was " + String.valueOf(result) +
               " old was " + String.valueOf (oldSummerFraction()) + " for " + lastPeriod())
    }
```

A little debugging quickly shows me that I am getting a lot of length zero summer periods when I shouldn't. The problem lies in using inte ger division; I need to use a cast.

```
    protected double summerFraction() {
        DateRange periodInSummer = lastPeriod().intersection(_zone.summer());
        return (double) periodInSummer.length() / lastPeriod().length();
    }
```

Now I get a different series of errors, all involving ranges with negative lengths. These are caused by the intersection method generating ranges whose start is after the end. I could treat these as errors, but in practice I've found that it is all right to allow these, treating them as empty ranges.

```
  class DateRange
    public boolean isEmpty() {
        return _start.after(_end);
    }
```

I need to adjust length to take this into account.

```
  class DateRange
    public int length() {
        if (isEmpty()) return 0;
        return _end.minus(_start) + 1;
    }
```

With that the tests work, including the one I added to probe for the bug that existed in the old version.

Decomposing the Base Charges

The next item to catch my eye is the baseCharge methods on residential and disability sites

```
class ResidentialSite
  protected Dollars baseCharge() {
      return new Dollars ((lastUsage()   * _zone.summerRate() * summerFraction()) +
          (lastUsage() * _zone.winterRate() * (1 - summerFraction())  ));
  }

class DisabilitySite
  protected Dollars baseCharge() {
      int cappedUsage = Math.min(lastUsage(), CAP);

      Dollars result;
      result =  new Dollars ((cappedUsage   * _zone.summerRate() * summerFraction()) +
          (cappedUsage * _zone.winterRate() * (1 - summerFraction())  ));

      result = result.plus(new Dollars (Math.max(lastUsage() - cappedUsage, 0) * 0.062));
      return result;
  }
```

The highlighted section is similar to both methods and can be extracted into the superclass. The difference is that the disability site caps the usage before applying this calculation while the residential site does not. I can deal with this by using *Extract Method (114)* and *Parameterize Method (240)*.

```
class Site
  Dollars residentialBaseCharge (int usage) {
      return new Dollars ((usage * _zone.summerRate() * summerFraction()) +
          (usage * _zone.winterRate() * (1 - summerFraction())));
  }

class ResidentialSite
  protected Dollars baseCharge() {
      return residentialBaseCharge (lastUsage())  ;
  }

class DisabilitySite
  protected Dollars baseCharge() {
      int cappedUsage = Math.min(lastUsage(), CAP);

      Dollars result;
      result =  residentialBaseCharge (cappedUsage)  ;
```

```
        result = result.plus(new Dollars (Math.max(lastUsage() - cappedUsage, 0) * 0.062));
        return result;
    }
```

As I look at this code I wonder whether the zone should not calculate the residentialBaseCharge. To do this I would have to move the behavior for summerFraction over there as well, and send it the usage and the lastPeriod as parameters. I don't have a strong feeling about it at the moment, but it would be useful if we had some zones with different factors than the summer period and the two rates. At the moment the site needs to know a lot about the zone, while if I moved the behavior over to zone it would only need to work with an int and a date range. It would also pull some behavior out of the Site class, which is getting fairly involved now. To be honest it seems six of one and half a dozen of the other to me, and in many cases I would leave it as it is for the moment. Since I'm demonstrating the techniques though, I might as well do it.

To move the residentialBaseCharge over to zone I look at the methods that are called by residentialBaseCharge. For each one I have to decide: do I move over the result of the method as a parameter, or do I move over the whole method. I decide to move over summerFraction and send in the usage and the period as parameters.

I do this by putting the two methods into zone, adjusting them to their new home and compiling. Then I find all the references to the moved methods by commenting them out and seeing where the compiler complains. I fix the problems, compile and test. Then I remove the commented out methods.

```
class Zone...
  double summerFraction(DateRange usagePeriod) {
      DateRange periodInSummer = usagePeriod.intersection(summer());
      return (double) periodInSummer.length() / usagePeriod.length();
  }

  Dollars baseCharge (int usage, DateRange usagePeriod) {
      return new Dollars ((usage * _summerRate * summerFraction(usagePeriod)) +
          (usage * _winterRate * (1 - summerFraction(usagePeriod))));
 }

class ResidentialSite...
  protected Dollars baseCharge() {
      return _zone.baseCharge (lastUsage(), lastPeriod())  ;
```

```
    }

  class DisabilitySite...
    protected Dollars baseCharge() {
        int cappedUsage = Math.min(lastUsage(), CAP);

        Dollars result;
        result =  _zone.baseCharge (cappedUsage, lastPeriod())  ;

        result = result.plus(new Dollars (Math.max(lastUsage() - cappedUsage, 0) * 0.062));
        return result;
    }
```

I can do some more with disability site's `baseCharge` method. First I can use *InlineTemp (121)* on `cappedUsage`.

```
  class DisabilitySite ...
    protected Dollars baseCharge() {
        Dollars result;
        result =  _zone.baseCharge (usageBelowCap(), lastPeriod());
        result = result.plus(new Dollars (Math.max(lastUsage() -
               usageBelowCap(), 0) * 0.062));
        return result;
    }

    protected int usageBelowCap() {
        return Math.min(lastUsage(), CAP);
    }
```

I do similar for the amount above the cap, and *Replace Magic Number with Symbolic Constant (210)*.

```
    protected Dollars baseCharge() {
        Dollars result;
        result =  _zone.baseCharge (usageBelowCap(), lastPeriod());
        result = result.plus(new Dollars ( usageAboveCap() * ABOVE_CAP_RATE ));
        return result;
    }

    protected int usageAboveCap() {
        return Math.max(lastUsage() - usageBelowCap(), 0);
    }

    private static final double ABOVE_CAP_RATE = 0.062;
```

With that Site and its subclasses are pretty well factored (Figure 15.15). All the methods are small and understandable. You might not choose the same refactorings as I did. In the end what counts is what is most understandable to your team. I find very small methods easier to deal with, but you might prefer a larger granularity.

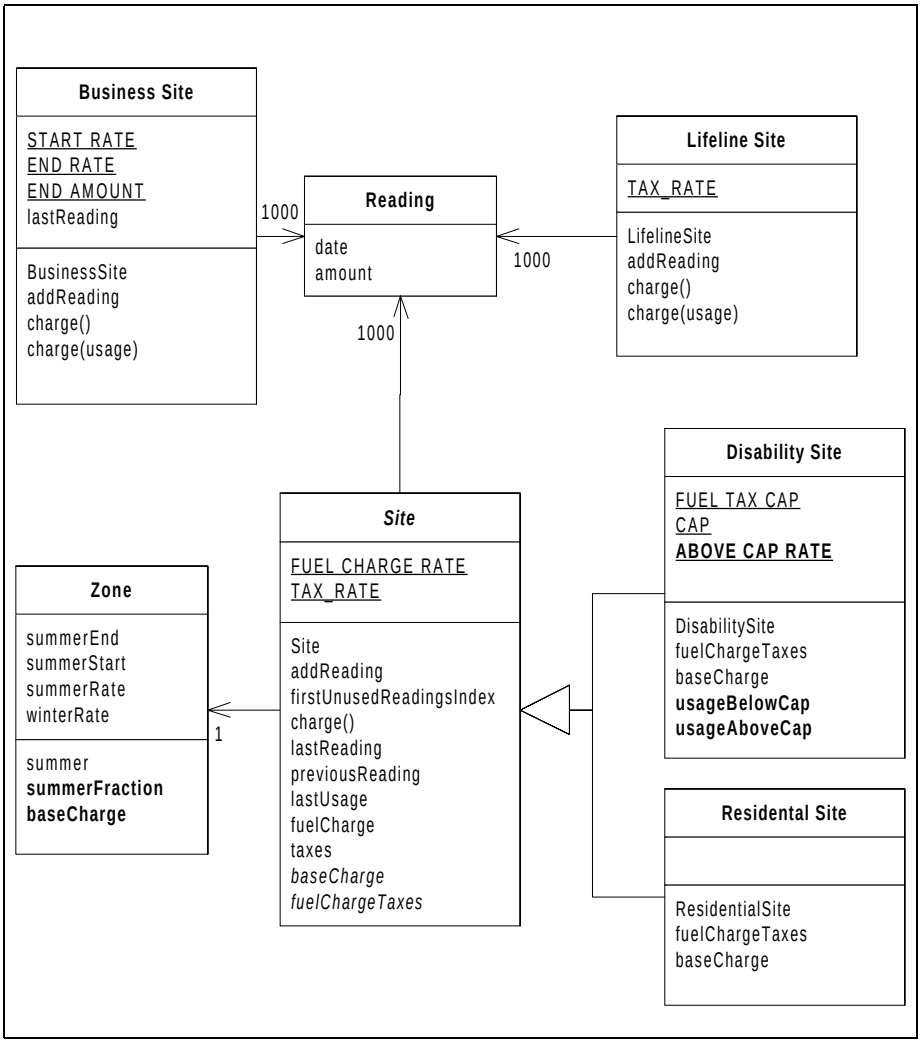

Figure 15.15: After decomposing the base charge methods

Fixing up `MfDate` and Date Range

I do some more looking at the classes I have been working on and spot two other methods to clean up. First is `DateRange`'s intersection

```
public DateRange intersection(DateRange arg) {
    MfDate newStart = (_start.after(arg.start())) ?
```

```
            _start :
            arg.start();
        MfDate newEnd = (_end.before(arg.end())) ?
            _end:
            arg.end();
        return new DateRange(newStart, newEnd);
    }
```

I can improve this by making a latest and earliest for `MfDate`.

```
class DateRange
public DateRange intersection(DateRange arg) {
        return new DateRange(
                MfDate.latest(_start, arg.start()), MfDate.earliest(_end, arg.end()));
      }

class MfDate
    public static MfDate earliest(MfDate arg1, MfDate arg2) {
        return (arg1.before(arg2)) ?
            arg1 :
            arg2;
    }

    public static MfDate latest(MfDate arg1, MfDate arg2) {
        return (arg1.after(arg2)) ?
            arg1 :
            arg2;
    }
```

A bigger looking problem is `dayOfYear`.

```
int dayOfYear() {
        int result;
        switch (getMonth()) {
            case 0:
               result = 0;
               break;
            case 1:
               result = 31;
               break;
            case 2:
               result = 59;
               break;
            case 3:
               result = 90;
               break;
            case 4:
               result = 120;
               break;
            case 5:
               result = 151;
               break;
```

```
            case 6:
               result = 181;
               break;
            case 7:
               result = 212;
               break;
            case 8:
               result = 243;
               break;
            case 9:
               result = 273;
               break;
            case 10:
               result = 304;
               break;
            case 11:
               result = 334;
               break;
            default :
               throw new IllegalArgumentException();
        };
        result += getDate();

        //check leap year
        if ((getYear()%4 == 0) && ((getYear() % 100 != 0) || ((getYear()  + 1900) % 400 == 0))) {
            result++;
        };

        return result;
    }
```

The long case statement takes up a lot of vertical space (and thus scrolls off the bottom of my browser). Another alternative is to use a *searching literal* [Beck].

```
    int dayOfYear() {
        int result;
        int[] monthNumbers = {0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334};
        result = monthNumbers[getMonth()];
        result += getDate();

        //check leap year
        if ((getYear()%4 == 0) && ((getYear() % 100 != 0) || ((getYear()  + 1900) % 400 == 0))) {
            result++;
        };

        return result;
    }
```

I use *Inline Temp (121)*.

```
int dayOfYear() {
    int result;
    result = daysToStartOfMonth() ;
    result += getDate();

    //check leap year
    if ((getYear()%4 == 0) && ((getYear() % 100 != 0) || ((getYear() + 1900) % 400 == 0))) {
        result++;
    };

    return result;
}

private int daysToStartOfMonth() {
    int[] monthNumbers = {0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334};
    return monthNumbers[getMonth()];
}
```

The leap year determination can also be extracted.

```
int dayOfYear() {
    int result = daysToStartOfMonth() + getDate();
    if (isLeapYear()) result++;
    return result;
}

boolean isLeapYear() {
    return (getYear()%4 == 0) && ((getYear() % 100 != 0) || ((getYear() + 1900) % 400 == 0));
}
```

In doing this I noticed another bug, the leap year will add one to the result even if the date is before the 28 Feb. I add a test to confirm and fix the bug.

```
class MfDate
  int dayOfYear() {
      int result = daysToStartOfMonth() + getDate();
      if (isLeapYear() &  this.after(new Date (getYear(), 1, 29)))  result++;
      return result;
  }
```

While I'm working on date I would like to make the date subtraction work across years. I'm too lazy to figure out the algorithm, so I pinch it from the Smalltalk image

```
class MfDate
  public int minus(MfDate arg) {
      return (getYear() == arg.getYear()) ?
          dayOfYear() - arg.dayOfYear() :                    // shortcut
          daysSince1901() - arg.daysSince1901();
  }
```

```
public int daysSince1901(){
      if (getYear() < 1) throw new IllegalArgumentException();
      int result;
      int yearIndex = getYear() - 1;
      result = yearIndex * 365;
      result += yearIndex / 4;                                    // ordinary leap years
      result += (yearIndex + 300) / 400;                          // leap centuries
      result -= yearIndex / 100;                                  // non-leap centuries
      result += dayOfYear() - 1;
      return result;
    }
```

I will leave it not working for the year 1900. I haven't tested to see i any of this works before the Java epoch (Jan 1 1970) in any case. Such dates are outside the current requirements of this program, so there's no need to worry about them just now. Figure 15.16 shows the date and date range classes.

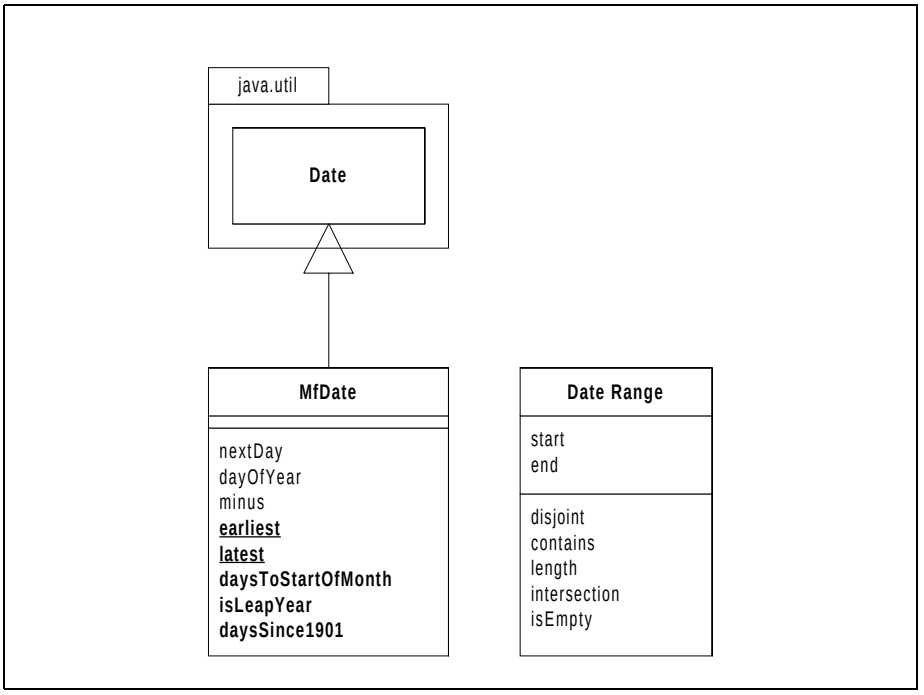

Figure 15.16: `MfDate` *and Date Range after cleaning up*

Refactoring So Far

1) Starting a Hierarchy of Sites
2) Simplifying the Charge Methods
3) Decomposing Site's Long Methods

Now site and its subclasses are pretty nicely factored. I've whittled down some long methods, and as a result moved some behavior around to better places. The date class is doing much more now, and the date range class is something that is already useful (just look at how much simpler `summerFraction` is now), and I'm sure will make modification easier in the future.

Adding Lifeline Site to the Hierarchy

Now its time to take another site and refactor it into the Site hierarchy. I will now work on lifeline site.

I begin by making lifeline site a subclass of site. The compiler complains that I haven't defined site's abstract methods, so I put in some placeholders. It's important to throw exceptions here, I've been bitten before by doing nothing and forgetting that I should have put some real code in here.

```
public class LifelineSite extends Site {
  protected Dollars baseCharge() {
      throw new AbstractMethodError("undefined baseCharge");
  }
  protected Dollars fuelChargeTaxes () {
      throw new AbstractMethodError("undefined fuelChargeTaxes");
  }
```

More seriously it complains because Site does not have a no-argument constructor. Site currently has a Zone but Lifeline Sites do not have zones. For the moment I will create a Lifeline site with a null Zone.

```
class LifelineSite {
  public LifelineSite() {
      super (null);
  };
```

First I will look at how the site handles readings. Lifeline site does it in a different way to that for the two prior sites. It adds the readings to the beginning instead of the end of the array.

```
public void addReading(Reading newReading) {
    Reading[] newArray = new Reading[_readings.length + 1];
    System.arraycopy(_readings, 0, newArray, 1, _readings.length);
    newArray[0] = newReading;
    _readings = newArray;
}
```

The charge method thus also works differently

```
public Dollars charge() {
    int usage = _readings[0].amount() - _readings[1].amount();
    return charge(usage);
}
```

It also does not compute a period for the charge.

As I inspect the methods I think that if I remove the readings array and alter the charge method to use lastReading and previousReading the thing will still work. I can test that hypothesis by changing it and run ning the tests. I remove the field _readings in lifeline site, and remove the addReading method. That way I will inherit the usual behavior fro site. Then I alter the charge method to make use of the new behavior.

```
public Dollars charge()
    {
      int usage = lastReading().amount() - previousReading().amount();
      return charge(usage);
    }
```

The compiler would tell me if anything else was using addReading or _readings and nothing was. The tests tell me if all is still working fine, and they come back OK.

In that step, as with the future ones, I am using the understanding I have gained of the site's behavior through the previous refactoring to help me with this refactoring. I'm expecting lifeline site to show a lot of similarity with the previous sites. I won't be flattened if it doesn't, but I will be helped if it does.

I see that lifeline site has its own TAX_RATE member. I can delete this, since the same value is on site.

Making Lifeline Site's charge fit the template

I now need to deal with charge() and compare it with Site's template method.

```
class Site
  public Dollars charge() {
      return baseCharge().plus(taxes()).plus(fuelCharge()).plus(fuelChargeTaxes());
  }

class LifelineSite
  private Dollars charge  (int usage) {

      double base = Math.min(usage,100) * 0.03;
      if (usage > 100) {
          base += (Math.min (usage,200) - 100) * 0.05;
      };
      if (usage > 200) {
          base += (usage - 200) * 0.07;
      };

      Dollars result = new Dollars (base);

      Dollars tax = new Dollars (result.minus(new Dollars(8)).
          max(new Dollars (0)).times(TAX_RATE));
      result = result.plus(tax);

      Dollars fuelCharge = new Dollars (usage * 0.0175);
      result = result.plus (fuelCharge);
      return result.plus (new Dollars (fuelCharge.times(TAX_RATE)));
  }
```

Site assumes four calculations added together: base, tax, fuel charge, and the tax on the fuel charge. The fuel charge stuff is an obvious first target. I started with replacing the highlighted code with method calls, but was caught out because I hadn't redefined `fuelChargeTaxes`. So I can start with just the `fuelCharge`.

```
  private Dollars charge  (int usage) {

      double base = Math.min(usage,100) * 0.03;
      if (usage > 100) {
          base += (Math.min (usage,200) - 100) * 0.05;
      };
      if (usage > 200) {
          base += (usage - 200) * 0.07;
      };

      Dollars result = new Dollars (base);
```

```
    Dollars tax = new Dollars (result.minus(new Dollars(8)).
        max(new Dollars (0)).times(TAX_RATE));
    result = result.plus(tax);

    result = result.plus ( fuelCharge());
    return result.plus (new Dollars (fuelCharge().times(TAX_RATE)));
}
```

Looking at the implementations for `fuelChargeTaxes`, I can see that the `LifelineSite`'s method is the same as that for residential site. So I pull residential site's method up to site with *Pull Up Method (271)*. I do that by simply cutting it from residential site and pasting it in site. Then I compile and test. Once that is done I can remove the placeholder from disability site and call the method fro charge.

```
private Dollars charge  (int usage) {

    double base = Math.min(usage,100) * 0.03;
    if (usage > 100) {
        base += (Math.min (usage,200) - 100) * 0.05;
    };
    if (usage > 200) {
        base += (usage - 200) * 0.07;
    };

    Dollars result = new Dollars (base);

    Dollars tax = new Dollars (result.minus(new Dollars(8)).
        max(new Dollars (0)).times(TAX_RATE));
    result = result.plus(tax);

    result = result.plus (fuelCharge());
    return result.plus (fuelChargeTaxes());
}
```

Working upwards the next thing I see is the calculation for the taxes. It looks like I can extract that into taxes.

```
private Dollars charge  (int usage) {

    double base = Math.min(usage,100) * 0.03;
    if (usage > 100) {
        base += (Math.min (usage,200) - 100) * 0.05;
    };
    if (usage > 200) {
        base += (usage - 200) * 0.07;
    };
```

```
        Dollars result = new Dollars (base);

        result = result.plus(taxes(result));

        result = result.plus (fuelCharge());
        return result.plus (fuelChargeTaxes());
    }

    protected Dollars taxes (Dollars base) {
        return new Dollars (base.minus(new Dollars(8)).
                  max(new Dollars (0)).times(TAX_RATE));
    }
```

The form does not exactly match the template, since I have to pass in the result to taxes as an argument. That's only a temporary problem and I will deal with it soon.

The top bit looks like lifeline site's base charge method.

```
    private Dollars charge  (int usage) {

        Dollars result = baseCharge();

        result = result.plus(taxes(result));

        result = result.plus (fuelCharge());
        return result.plus (fuelChargeTaxes());
    }

    protected Dollars baseCharge() {
        double result = Math.min(lastUsage(),100) * 0.03;
        if (lastUsage() > 100) {
            result += (Math.min (lastUsage(),200) - 100) * 0.05;
        };
        if (lastUsage() > 200) {
            result += (lastUsage() - 200) * 0.07;
        };

        return new Dollars (result);
    }
```

All I need to do is fix the taxes method by *Replace Parameter with Method (245)*, and I can eliminate the charge method on lifeline site.

```
  class Site
    public Dollars charge() {
        return baseCharge().plus(taxes()).plus(fuelCharge()).plus(fuelChargeTaxes());
    }

  class LifelineSite
    protected Dollars taxes () {
```

```
        return new Dollars ( baseCharge() .minus(new Dollars(8)).
                max(new Dollars (0)).times(TAX_RATE));
    }
```

That all went very smoothly. The basic method was the same as the previous cases, probably they all shared a common heritage of cut and paste. Figure 15.17 shows lifeline site as part of the hierarchy.

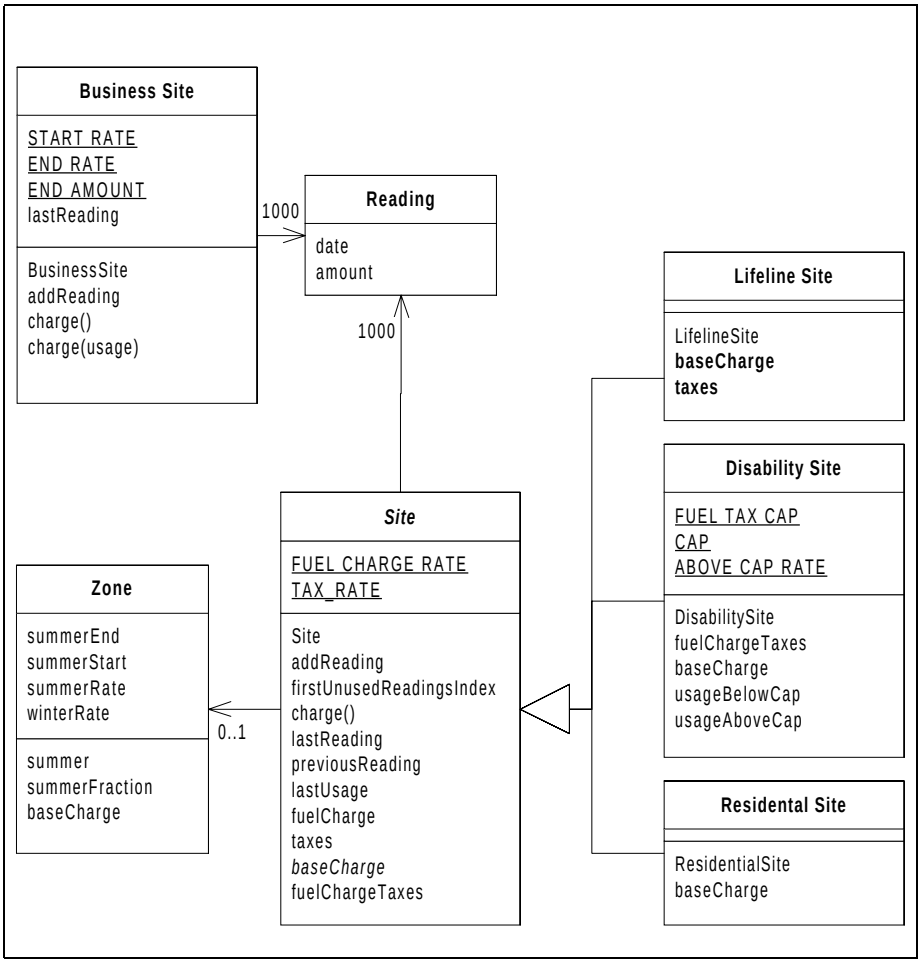

Figure 15.17: After making lifeline site a subclass of site

Refactoring Lifeline Site's baseCharge

I think there is some simplification I can do with the lifeline site's baseCharge method, however. The code looks somewhat repetitive. I think I should be able to factor it with extracting a parameterized method. I start with the first bit of code.

```
class LifelineSite
  protected Dollars baseCharge() {
      double result = usageUnder(100) * 0.03;
      if (lastUsage() > 100) {
          result += (Math.min (lastUsage(),200) - 100) * 0.05;
      };
      if (lastUsage() > 200) {
          result += (lastUsage() - 200) * 0.07;
      };

      return new Dollars (result);
  }

  protected int usageUnder(int limit) {
      return Math.min(lastUsage(),limit);
  }
```

UsageUnder works for that bit, but I need to introduce the conditional and the previous value for it to work for the 200 limit.

```
  protected Dollars baseCharge() {
      double result = usageInRange(0, 100)  * 0.03;
      result += usageInRange (100,200)  * 0.05;
      if (lastUsage() > 200) {
          result += (lastUsage() - 200) * 0.07;
      };

      return new Dollars (result);
  }
  protected int usageInRange(int start, int end) {
      if (lastUsage() > start) return Math.min(lastUsage(),end) - start;
      else return 0;
  }
```

I can then apply it to the topmost part of the range

```
  protected Dollars baseCharge() {
      double result = usageInRange(0, 100) * 0.03;
      result += usageInRange (100,200) * 0.05;
      result += usageInRange (200, Integer.MAX_VALUE)  * 0.07;

      return new Dollars (result);
  }
```

This all works, but there are few too many rules to using it. The programmer has to ensure that the values are set up so the lower number match the upper numbers, and that the top number is set to Integer.MAX_VALUE. I would prefer something where you could set things up with an array of bounds and values. I feel an object coming on. I would like it to work like this

```
//pseudo Java !
RateTable table = {          0.03, 100
                             0.05, 200
                             0.07};
return table.value(lastUsage());
```

I often like to think about how I would like to use an object before I try creating one. From this I can get a sense of what the methods look like. But I now have enough to try it out.

```
class RateTable {
    public RateTable(double[] table) {
      _table = table;
    }
    private double[] _table;
}
```

The key lies in its behavior. I need to move the usageInRange method over to it.

```
private int usageInRange(int amount, int start, int end) {
    if (amount > start) return Math.min(amount,end) - start;
    else return 0;
}
```

Then I need to a create value method to work over the array. One possibility is something along the following lines.

```
// not working code!
public int value(int amount) {
    double result = 0;
    for (int i=0; i < _table.length; i += 2) {
        result += usageInRange(amount, _table[i], _table[i-2]) * _table [i+1]);
    }
    return new Dollars (result);
}
```

The trouble with this is that there is far too much special interpretation of the array indices, not to mention of how to fix it to support the maximum value, or how to deal with an array that would contain both reals and ints.

A better idea is to use two arrays. Just because the constructor only uses one doesn't mean the class can't change it into two.

```
public RateTable(double[] arg) {
    int arrayLengths = arg.length / 2 + 1;
    _rates = new double[arrayLengths];
    _limits = new int[arrayLengths];
    int argIndex = 0;
    for (int i = 0; i < (arrayLengths - 1); i++) {
        _rates[i] = arg[argIndex++];
        _limits[i] = (int) arg[argIndex++];
    };
    _rates[arrayLengths - 1] = arg [arg.length -1];
    _limits[arrayLengths - 1] = Integer.MAX_VALUE;
 }
```

That's a messy method, but it seems to set the rate table up in the right way. It makes value easier to write.

```
public Dollars value(int amount) {
      double result = 0;
      result = usageInRange(amount, 0, _limits[0]) * _rates[0];
      for (int i=1; i < _rates.length; i++)
          result += usageInRange(amount, _limits[i-1], _limits[i]) * _rates[i];
      return new Dollars (result);
   }
```

I now have a useful rate table class (Figure 15.18

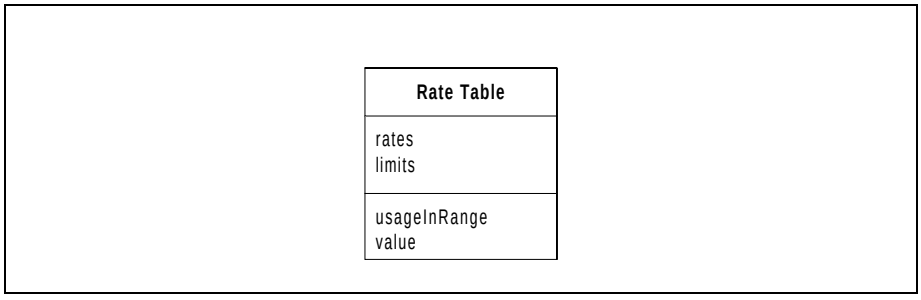

Figure 15.18: Rate table class

I can then alter `baseCharge`.

```
protected Dollars baseCharge() {
    double [] table = {     0.03, 100,
                            0.05, 200,
                            0.07};
    return new Dollars (new RateTable(table).value(lastUsage()));
}
```

Well it took a bit of fiddling to get the constructor to work but it does now work. Was it worth the change? I don't think so. The new class certainly makes it much easier to write baseCharge, but the new class is rather complicated to deal with. I'm sure I can refactor it into a better shape, but I'm not sure its worth the effort to simplify one method. For the moment I will change baseCharge back to what it was at the start of this section.

```
protected Dollars baseCharge() {
    double result = Math.min(lastUsage(),100) * 0.03;
    if (lastUsage() > 100) {
        result += (Math.min (lastUsage(),200) - 100) * 0.05;
    };
    if (lastUsage() > 200) {
        result += (lastUsage() - 200) * 0.07;
    };

    return new Dollars (result);
}
```

A false move like this does happen from time to time in refactoring. You see something that looks like a simplification, and find out that it seems to make things more complex. In these cases you just have to chalk it up to experience and back out of the change. The aim is to make the code simpler, not more complicated.

Changing _readings to a Vector

At the moment the readings field is a 1000 size array. Arrays are awk ward in this situation. You just know that at some point someone is going to add the 1001th element. You also know that if you make the array too big in the first place (like Integer.MAX_VALUE) then you will waste a lot of memory. What we need is an array that can grow as we need it. Java has aVector class for that very purpose.

The readings field can be set up as a vector

```
private Vector _readings = new Vector();
```

I need to alter those methods that access it. To find those I do a find using the editor.

```
public void addReading(Reading newReading) {
    _readings.addElement(newReading);
}
private int firstUnusedReadingsIndex () {
```

```
        return _readings.size();
    }
    protected int lastUsage() {
        return lastReading().amount() - previousReading().amount();
    }
    public Reading lastReading() {
        return (Reading) _readings.lastElement();
    }
    public Reading previousReading() {
        return (Reading) _readings.elementAt(_readings.size() - 2);
    }
```

Those do the trick and also make the code easier to read. `FirstUnusedReadingsIndex` is not necessary any more so I can remove it. The one problem with vectors is that you have to downcast anything you take out of it. So I ensure that `lastReading` and `previousReading` encapsulate the downcasting.

This ability to change the data structure of the class, without altering the interface, is one of the joys of an OO language. In performance tuning it is quite possible that we would find a hotspot in the access or update of the readings vector. Again we should be able to change its implementation easily to provide performance improvements at that time.

Maintaining the invariant for the readings

The way the Site class works, there is an assumption in the Site class about the readings, that is that the readings are in order of dates, earliest first. There is, however, nothing to stop someone from adding a reading that would violate this assumption. The class as an internal invariant which says that the readings are in date order. In a language such as Eiffel, we could code this invariant directly into the class. This is not so easy in Java. However we can improve matters by considering what operations could change the state of the invariant, causing it to become false. The only operation to do this is the `addReading` operation. We need to add a clause to `addReading` to check for incorrect dates.

```
public void addReading(Reading newReading) {
    if (newReading.date().before(lastReading().date()))
        // do something ;
    _readings.addElement(newReading);
}
```

What action should we take? The site could choose to insert the read ing in the appropriate place, instead of adding it at the end, but that would invalidate previous uses of charge. It may be that later iterations of this program will give it the ability to handle these readings intelligently, but all it can do for the moment is throw an exception.

Now I've decided to throw an exception, I have to decide whether I throw a checked or unchecked exception. The Java texts emphasize that you should use checked exceptions, indeed they imply that you should rarely throw unchecked exceptions. I'm not so sure about this rule. Checked exceptions are good, in that you force the user to decide what to do when they go wrong, but they can lead to very cluttered code. Every calling program needs to decide how to handle the excep tion, if they cannot decide they have to propagate the exception, and the decision making.

I prefer to think about it in terms of Design By Contact. Is the fact that the date of the new reading be later than existing readings part of the pre-condition of the `addReading` method? If so then it is the callers responsibility to check that before the call `addReading`, and `addReading` should not throw a checked exception. `AddReading` should only throw a checked exception if you decide that it is `addReading's` responsibility to check for that condition.

It's hard to make a definitive answer from here, since it depends on the context of `addReading` from the overall program, and we are only look ing at this section of the program. However since all the knowledge o whether it is a problem or not lies within the site, I will make it a checked exception.

```
public void addReading(Reading newReading) throws IncorrectReadingException {
  if (newReading.date().before(lastReading().date()))
      throw new IncorrectReadingException ("Reading is before previous reading");
  _readings.addElement(newReading);
}
```

The code complies, but runs into a problem when it executes. The first time this method is used, `lastReading` tries to read from an empty vector, and throws a `NoSuchElementException`. As I look at it I suppose I should alter `lastReading` to return a null if the `_readings` vector is empty, but I would still have to alter `addReading` to deal with the null. I could

avoid that by creating a null object for reading, but that is a lot of effort for one case. So the simplest thing to do is

```
public void addReading(Reading newReading) throws IncorrectReadingException {
      if (!_readings.isEmpty() && newReading.date().before(lastReading().date()))
          throw new IncorrectReadingException ("Reading is before previous reading");
      _readings.addElement(newReading);
    }
```

If I need to adjust lastReading for an empty vector, I can do that later. The condition is rather long winded however, so I extract it.

```
public void addReading(Reading newReading) throws IncorrectReadingException {
    if (isNotLatestReading(newReading) )
        throw new IncorrectReadingException ("Reading is before previous reading");
    _readings.addElement(newReading);
}

private boolean isNotLatestReading(Reading arg) {
    return !_readings.isEmpty() && arg.date().before(lastReading().date());
}
```

And, now I've added the check, I also add a test to check that it is working

```
class LifelineSiteTester
  void testIncorrectReading() throws Exception {
      _subject.addReading(new Reading (25, new MfDate ("8 Sep 1997")));
      try {
          _subject.addReading(new Reading (125, new MfDate ("1 Sep 1997")));
          assert(false);
      } catch (IncorrectReadingException e) {}
  }
```

Refactoring So Far

1) Starting a Hierarchy of Sites
2) Simplifying the Charge Methods
3) Decomposing Site's Long Methods
4) Adding Lifeline Site to the Hierarchy

Lifeline site fitted in without too much trouble, and I also took the trouble to clean up the way site deals with readings. The rate table idea

had possibilities, and might be useful in the future, but is not worth its keep for now.

Adding Business Site to the Hierarchy

Now its time to work on the final class, business site. My first move is to make it a subclass of Site and to compile. It tells me it needs an implementation for baseCharge and a no-arg constructor. The former seems reasonable but the latter is irritating. I only need it because busi ness site, like lifeline site, does not have a zone. Instead I can add a no arg constructor to site, and remove the no-arg constructor from lifeline site.

```
class Site...
  Site() {}
```

This does not affect residential site or disability site for they have their own constructors. They cannot use the no-arg constructor unless I add one.

Business site also has a variation on addReading and how the readings are obtained.

```
class BusinessSite...
  public void addReading(Reading newReading) {
      _readings[++lastReading] = newReading;
  }
  public Dollars charge() {
      int usage = _readings[lastReading].amount() - _readings[lastReading -1].amount();
      return charge(usage);
  }
```

So my first move, as with lifeline site is to remove addReading, change charge, and run the tests to see if all still works.

```
  public Dollars charge() {
      int usage = lastReading().amount() - previousReading().amount();
      return charge(usage);
  }
```

Indeed it did.

Making Business Site's charge method fit the tem-

plate

Now it's time to look at the one argument charge method, and to see if we can break it down into the usual four items to sum. Currently charge looks like this

```
private  Dollars charge(int usage) {
      Dollars result;

    if (usage == 0) return new Dollars(0);

   double t1 = START_RATE - ((END_RATE * END_AMOUNT) - START_RATE) / (END_AMOUNT - 1);
   double t2 = ((END_RATE * END_AMOUNT) - START_RATE) *
           Math.min(END_AMOUNT, usage) / (END_AMOUNT - 1);
   double t3 = Math.max(usage - END_AMOUNT, 0) * END_RATE;

   result = new Dollars (t1 + t2 + t3);

   // fuel charge
   result = result.plus(new Dollars (usage * 0.0175));

   // add in the taxes
   Dollars base = new Dollars (result.min(new Dollars (50)).times(0.07));
       if (result.isGreaterThan(new Dollars (50))) {
           base = new Dollars (base.plus(result.min(new Dollars(75)).minus
                 (new Dollars(50)).times(0.06)));
       };
       if (result.isGreaterThan(new Dollars (75))) {
           base = new Dollars (base.plus(result.minus(new Dollars(75)).times(0.05)));
       };
   result = result.plus(base);
   return result;

    }
```

The situation is slightly different. This time we have a base charge, a fuel charge, but a combined taxes method that does an algorithm on the combined amount. This method will not fit our current template method. But it may be possible to alter the template method to make it all fit. Our first task is to decompose this method to make it more palatable. We can begin by using the superclass's fuel charge.

```
private  Dollars charge(int usage) {
      Dollars result;

    if (usage == 0) return new Dollars(0);

   double t1 = START_RATE - ((END_RATE * END_AMOUNT) - START_RATE) / (END_AMOUNT - 1);
```

```
  double t2 = ((END_RATE * END_AMOUNT) - START_RATE) * Math.min(END_AMOUNT, usage) / (END_AMOUNT
- 1);
  double t3 = Math.max(usage - END_AMOUNT, 0) * END_RATE;

  result = new Dollars (t1 + t2 + t3);

  //add the fuel charge
  result = result.plus( fuelCharge() );

  // add the taxes
  Dollars base = new Dollars (result.min(new Dollars (50)).times(0.07));
      if (result.isGreaterThan(new Dollars (50))) {
          base     =     new     Dollars     (base.plus(result.min(new     Dollars(75)).minus(new
Dollars(50)).times(0.06)));
      };
      if (result.isGreaterThan(new Dollars (75))) {
          base = new Dollars (base.plus(result.minus(new Dollars(75)).times(0.05)));
      };
  result = result.plus(base);
  return result;

    }
```

We can then extract the baseCharge.

```
  private  Dollars charge(int usage) {
      Dollars result =  baseCharge() ;
      result = result.plus(fuelCharge());

      // add the taxes
      Dollars base = new Dollars (result.min(new Dollars (50)).times(0.07));
          if (result.isGreaterThan(new Dollars (50))) {
             base    =    new    Dollars    (base.plus(result.min(new    Dollars(75)).minus(new
Dollars(50)).times(0.06)));
          };
          if (result.isGreaterThan(new Dollars (75))) {
             base = new Dollars (base.plus(result.minus(new Dollars(75)).times(0.05)));
          };
      result = result.plus(base);
      return result;

  }

  public Dollars baseCharge() {
      if (lastUsage() == 0) return new Dollars(0);
      double t1 = START_RATE - ((END_RATE * END_AMOUNT) - START_RATE) / (END_AMOUNT - 1);
      double t2 = ((END_RATE * END_AMOUNT) - START_RATE) *
             Math.min(END_AMOUNT, lastUsage()) / (END_AMOUNT - 1);
      double t3 = Math.max(lastUsage() - END_AMOUNT, 0) * END_RATE;
      return new Dollars (t1 + t2 + t3);
    }
```

And then the taxes

```
private  Dollars charge(int usage) {
    Dollars result = baseCharge();
    result = result.plus(fuelCharge());
    result = result.plus( taxes());
    return result;
}
protected Dollars taxes() {
    Dollars taxable = baseCharge().plus(fuelCharge());
    Dollars result = new Dollars (taxable.min(new Dollars (50)).times(0.07));
    if (taxable.isGreaterThan(new Dollars (50))) {
        result = new Dollars (result.plus(taxable.min(new Dollars(75)).minus(
             new Dollars(50)).times(0.06)));

    };
    if (taxable.isGreaterThan(new Dollars (75))) {
        result = new Dollars (result.plus(taxable.minus(new Dollars(75)).times(0.05)));
    };
    return result;
}
```

We can now alter the template method to work with this site. First I change the name of the current taxes method in Site to baseTaxes. When I do this I need to first look to see if any subclass implements taxes. I can do that with a global search. Lifeline site does this so I rename taxes in both site and lifeline site. Next I need to check which methods call taxes. I can do this by either a text search or by compiling. The compiler tells me that only charge calls taxes. I now add a taxes method on site that works for the three other sites and test.

```
class Site...
  public Dollars charge() {
      return baseCharge().plus(fuelCharge()).plus(taxes());
  }

  protected Dollars taxes() {
      return baseTaxes().plus(fuelChargeTaxes());
  }
```

Now site's template method is in the right form, I can remove the charge methods on business site. Business site is now a fully functioning member of the site family, as Figure 15.19 illustrates.

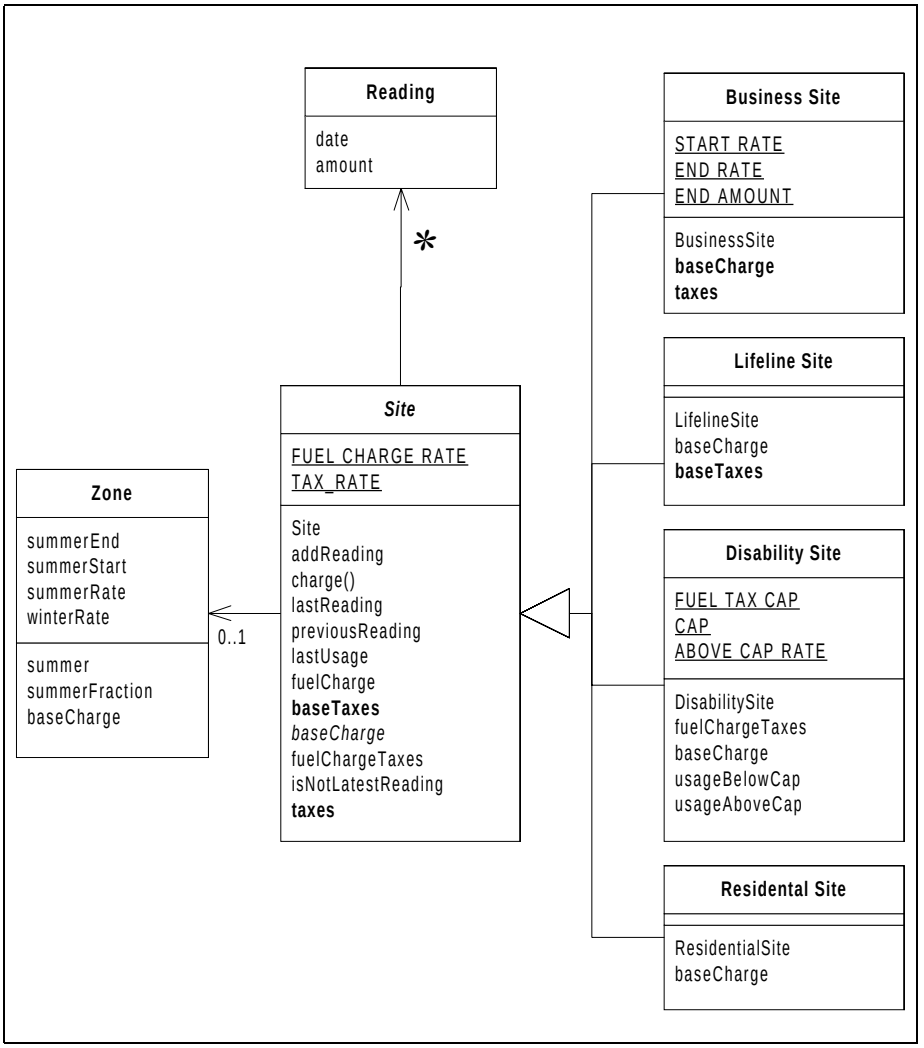

Figure 15.19: After business site joined the happy family

A Second Attempt to Use a Rate Table

Business site's method for taxes has a very familiar form.

```
protected Dollars taxes() {
    Dollars taxable = baseCharge().plus(fuelCharge());
    Dollars result = new Dollars (taxable.min(new Dollars (50)).times(0.07));
```

```
        if (taxable.isGreaterThan(new Dollars (50))) {
            result = new Dollars (result.plus(taxable.min(new Dollars(75)).minus
                  (new Dollars(50)).times(0.06)));

        };
        if (taxable.isGreaterThan(new Dollars (75))) {
            result = new Dollars (result.plus(taxable.minus(new Dollars(75)).times(0.05)));
        };
        return result;
    }
```

The form is exactly the same as that for Lifeline's site's base charge. This makes me want to revisit the rate table class (see p age437), since I will now have two places to use it. First I dig it out and plug it into BusinessSite.taxes and test.

```
class BusinessSite
  protected Dollars taxes() {
      Dollars taxable = baseCharge().plus(fuelCharge());
      double [] table = {               0.07, 50,
                                        0.06, 75,
                                        0.05};
      return new Dollars (new RateTable(table).value(taxable.amount()));
  }
```

I can do some extraction to make that clearer.

```
class BusinessSite...
  protected Dollars taxes() {
      return new Dollars ( taxTable() .value(taxable().amount()));
  }

  protected Dollars taxable () {
      return baseCharge().plus(fuelCharge());
  }

  protected RateTable taxTable() {
      double [] tableData = {           0.07, 50,
                                        0.06, 75,
                                        0.05};
      return new RateTable (tableData);
  }
```

I will do the same for lifeline site

```
class LifelineSite...
  protected Dollars baseCharge() {
      return new Dollars (baseChargeTable().value(lastUsage()));
  }

  protected RateTable baseChargeTable() {
```

```
        double [] tableData = { 0.03, 100,
                                0.05, 200,
                                0.07};
        return new RateTable (tableData);
    }
```

Now I'm going to actually use RateTable I want to refactor its code to make it easier to understand. The main bit that concerns me is the con structor.

```
class RateTable...
    public RateTable(double[] arg) {
        int arrayLengths = arg.length / 2 + 1;
        _rates = new double[arrayLengths];
        _limits = new int[arrayLengths];
        int argIndex = 0;
        for (int i = 0; i < (arrayLengths - 1); i++) {
            _rates[i] = arg[argIndex++];
            _limits[i] = (int) arg[argIndex++];
        };
        _rates[arrayLengths - 1] = arg [arg.length -1];
        _limits[arrayLengths - 1] = Integer.MAX_VALUE;
    }
```

Using vectors seems like it would simplify things a lot, but Vectors must contain objects, not reals or ints, so we would have a lot of down-

casting and conversion in the value method. For the moment nothing really occurs to me that I can do to it, so I will let it be (Figure 15.20).

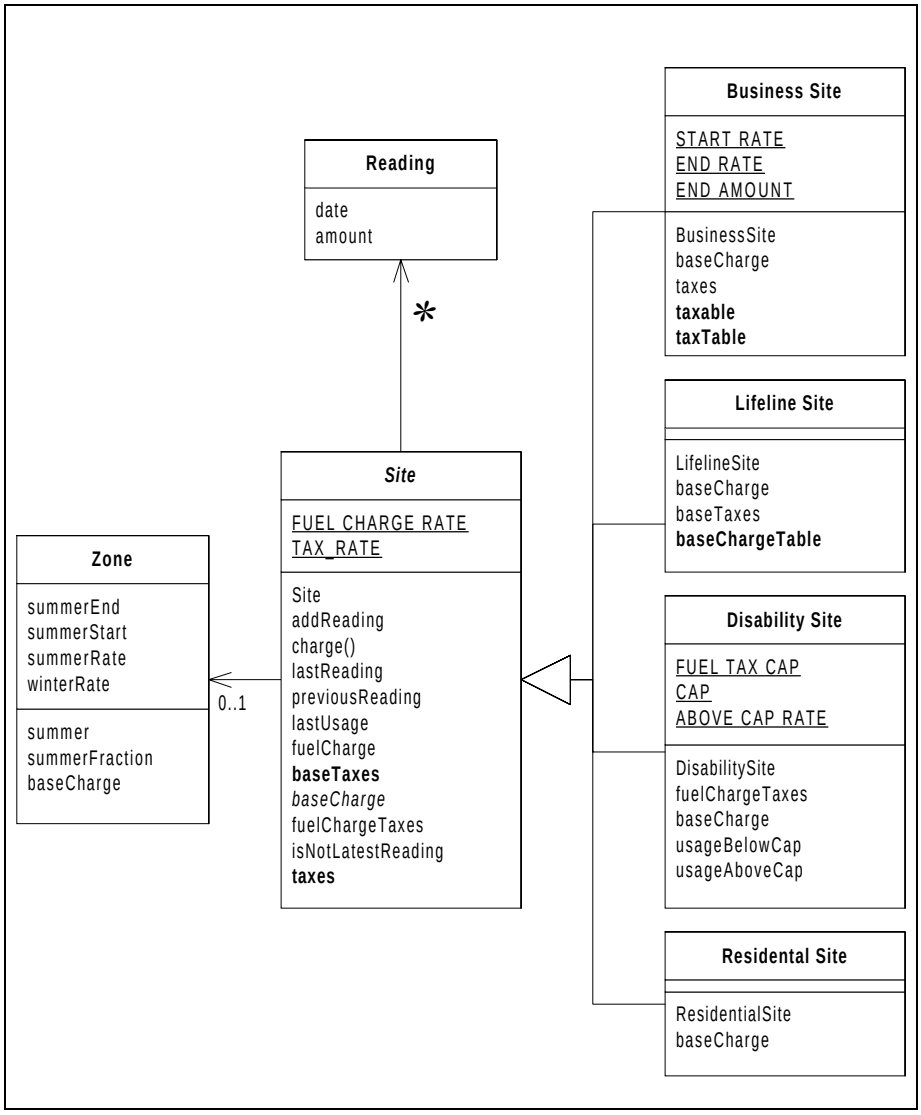

Figure 15.20: After introducing the rate table

Decomposing Business Site's baseCharge

Another awkward looking method is business site's baseCharge.

```
protected Dollars baseCharge() {
    if (lastUsage() == 0) return new Dollars(0);
    double t1 = START_RATE - ((END_RATE * END_AMOUNT) - START_RATE) / (END_AMOUNT - 1);
    double t2 = ((END_RATE * END_AMOUNT) - START_RATE) *
        Math.min(END_AMOUNT, lastUsage()) / (END_AMOUNT - 1);
    double t3 = Math.max(lastUsage() - END_AMOUNT, 0) * END_RATE;
    return new Dollars (t1 + t2 + t3);
}
```

I can refactor this by first looking at the way the lastUsage is used. One part, up to the END_AMOUNT is multiplied by a factor to make t2. That part over END_AMOUNT is multiplied by the END_RATE. I can thus divide the lastUsage into its two parts with methods.

```
protected Dollars baseCharge() {
    if (lastUsage() == 0) return new Dollars(0);
    double t1 = START_RATE - ((END_RATE * END_AMOUNT) - START_RATE) / (END_AMOUNT - 1);
    double t2 = ((END_RATE * END_AMOUNT) - START_RATE) *
           usageBelowLimit()  / (END_AMOUNT - 1);
    double t3 =  usageAboveLimit()  * END_RATE;
    return new Dollars (t1 + t2 + t3);
}

protected int usageAboveLimit () {
    return Math.max(lastUsage() - END_AMOUNT, 0);
}

protected int usageBelowLimit() {
    return Math.min(END_AMOUNT, lastUsage());
}
```

The usageBelowLimit is being multiplied by a static factor that is calculated from the current static contsants. I can factor this out.

```
protected Dollars baseCharge() {
    if (lastUsage() == 0) return new Dollars(0);
    double t1 = START_RATE -  belowLimitRate() ;
    double t2 = usageBelowLimit() *  belowLimitRate() ;
    double t3 = usageAboveLimit() * END_RATE;
    return new Dollars (t1 + t2 + t3);
}

protected static double belowLimitRate() {
    return ((END_RATE * END_AMOUNT) - START_RATE) / (END_AMOUNT - 1);
}
```

Finally I can rename the temps to something that better reflects my understanding of their use

```
protected Dollars baseCharge() {
```

```
    if (lastUsage() == 0) return new Dollars(0);
    double constant = START_RATE - belowLimitRate();
    double chargeBelowLimit  = usageBelowLimit() * belowLimitRate();
    double chargeAboveLimit  = usageAboveLimit() * END_RATE;
    return new Dollars ( constant + chargeBelowLimit  + chargeAboveLimit );
}
```

Refactoring So Far

1) Starting a Hierarchy of Sites
2) Simplifying the Charge Methods
3) Decomposing Site's Long Methods
4) Adding Lifeline Site to the Hierarchy
5) Adding Business Site to the Hierarchy

With that, the classes end up looking like Figure 15.21. The code is now much more amenable to changes in the charge calculations for the existing sites, and the addition of new sites. As we add new sites, with new rules, there may be some further refactoring needed, just as happened with the business site. But many new sites and charge changes should only involve one or two of the now very focused methods. Refactoring never stops, it is a continuous process of improvement to a program.

This is a long example, but it gives you a taste of how refactoring works in practice. At the beginning you only have a vague idea of what to do. You start by refactoring little things you can see just to understand how the system works. As your understanding increases, and as the code clarifies under your manipulations, you see the bigger picture. Then you can direct the refactoring a little more. But you must

always be responsive to the state of the code, looking for necessary complexity and bad smells.

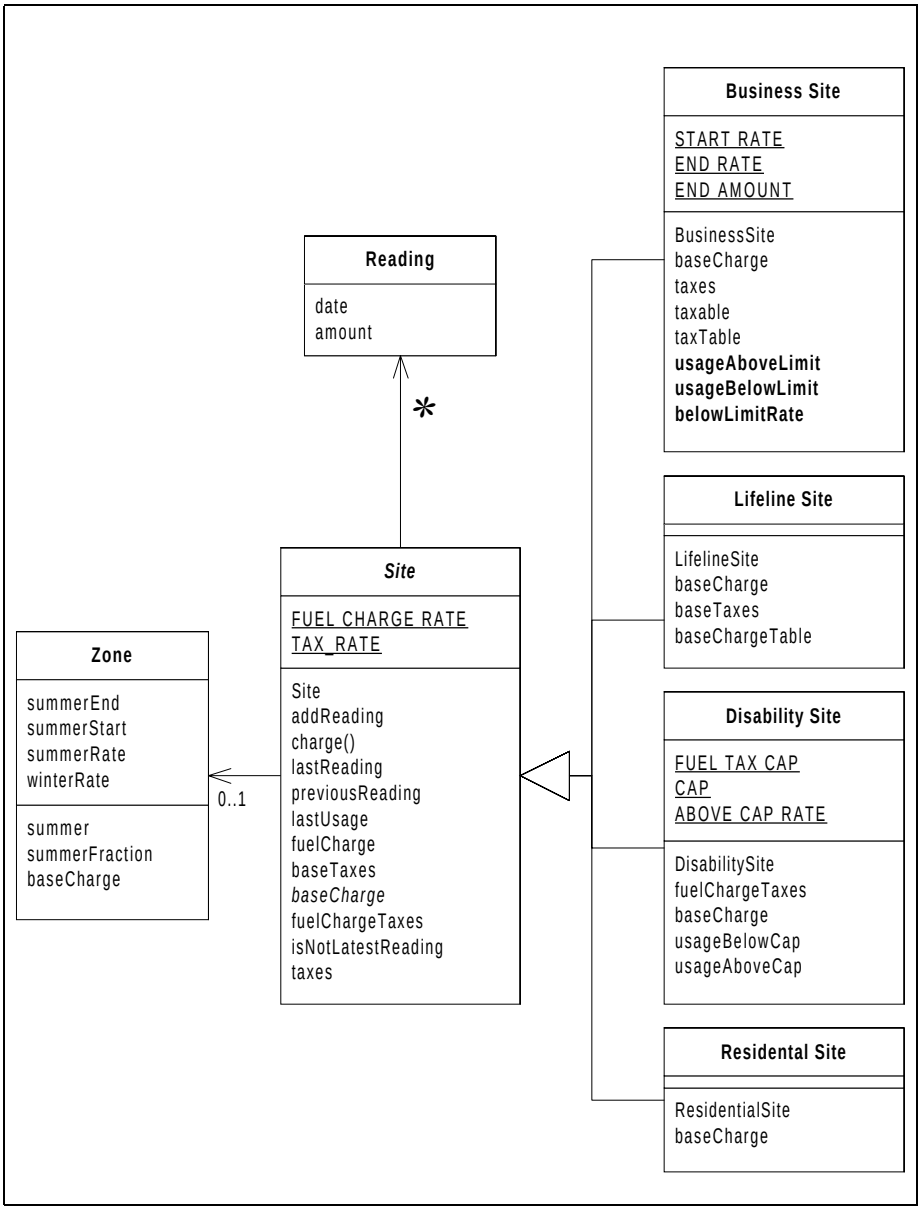

Figure 15.21: Final state of the classes