Encapsulating Algorithms with the Template Method Design Pattern

by Barry A. Burd and Michael P. Redlich

Introduction

This article, the sixth in a series about design patterns, introduces the Template Method design pattern, one of the 23 design patterns defined in the legendary 1995 book *Design Patterns – Elements of Reusable Object-Oriented Software*. The authors of this book, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, are known affectionately as the *Gang of Four* (GoF).

Design Patterns

The GoF book defines 23 design patterns. The patterns fall into three categories:

- A creational pattern abstracts the instantiation process.
- A structural pattern groups objects into larger structures.
- A behavioral pattern defines better communication among objects.

The Template Method design pattern fits into the behavioral category.

The Template Method Pattern

According to the GoF book, the Template Method design pattern "defines the skeleton of an algorithm in a method, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure."

Motivation

Warning: The example that follows uses some mathematical terminology. If math isn't your strong suit, don't worry. Just gloss over any details that make you uneasy. You can skim all you want and still get the gist of the idea.

Let's say you want to calculate the area under a curve, say $f(x) = \sin(x)$, from 0 to π within an application. Figure 1 is a graphical representation of this function.

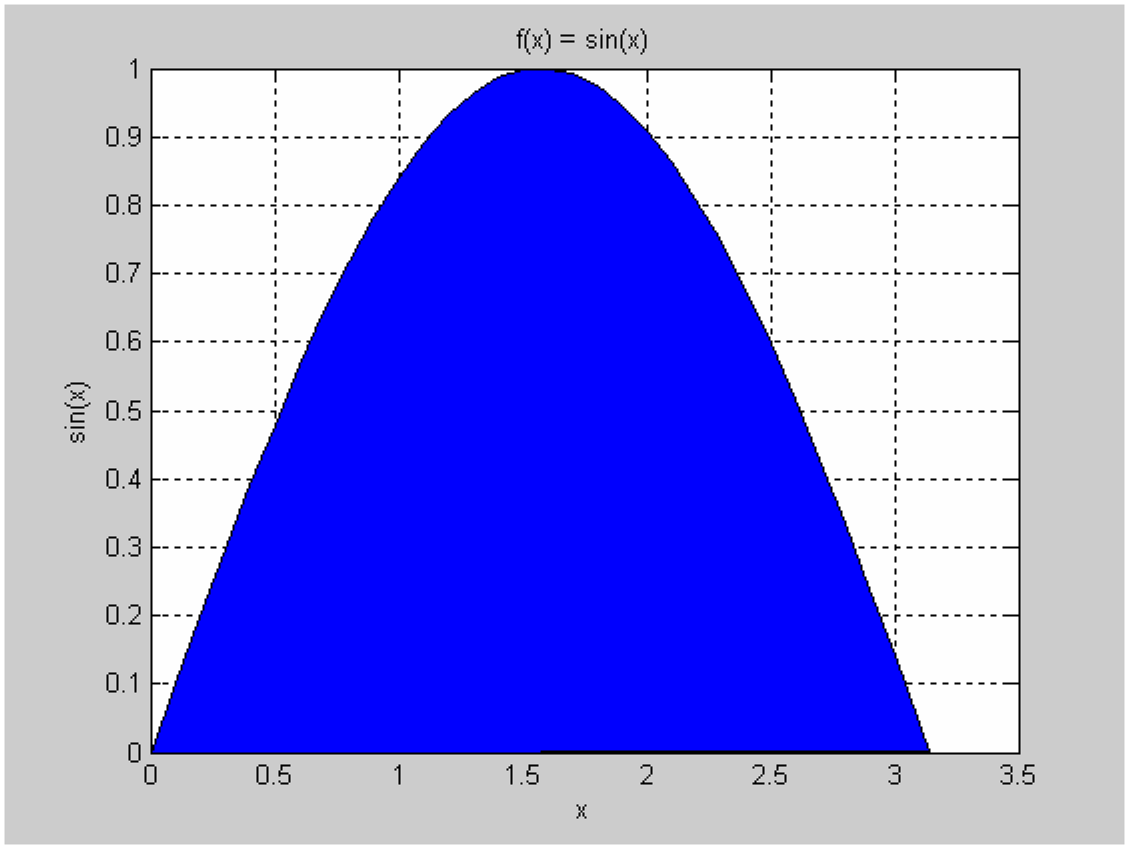

Figure 1: The plot of the curve $f(x) = \sin(x)$

To calculate the area under a curve, you dust off your old Calculus book. After pouring through the chapter on definite integrals, you come to the following conclusions:

* The area under the curve in Figure 1 is 2. Whew! That wasn't difficult, but,...
* Finding the areas under some other curves can be very tricky.

To help you find the areas under some curves, you use a *numerical method*. With a numerical method, you divide an area into smaller, more manageable chunks. Then you add up the areas of all the chunks. The chunks may not fit exactly inside the curve, so you don't always get an exact answer. But in many situations, an approximate answer is good enough.

The simplest numerical methods are the ***Trapezoid Rule*** and ***Simpson's Rule***. Don't worry - you don't have to understand the math in order to read the rest of this article. If you're curious, the difference between these two rules is the shape of the chunks. The Trapezoidal Rule uses rectangular chunks (as in Figure 2), and Simpson's Rule uses chunks slightly curved chunks (as in Figure 3).

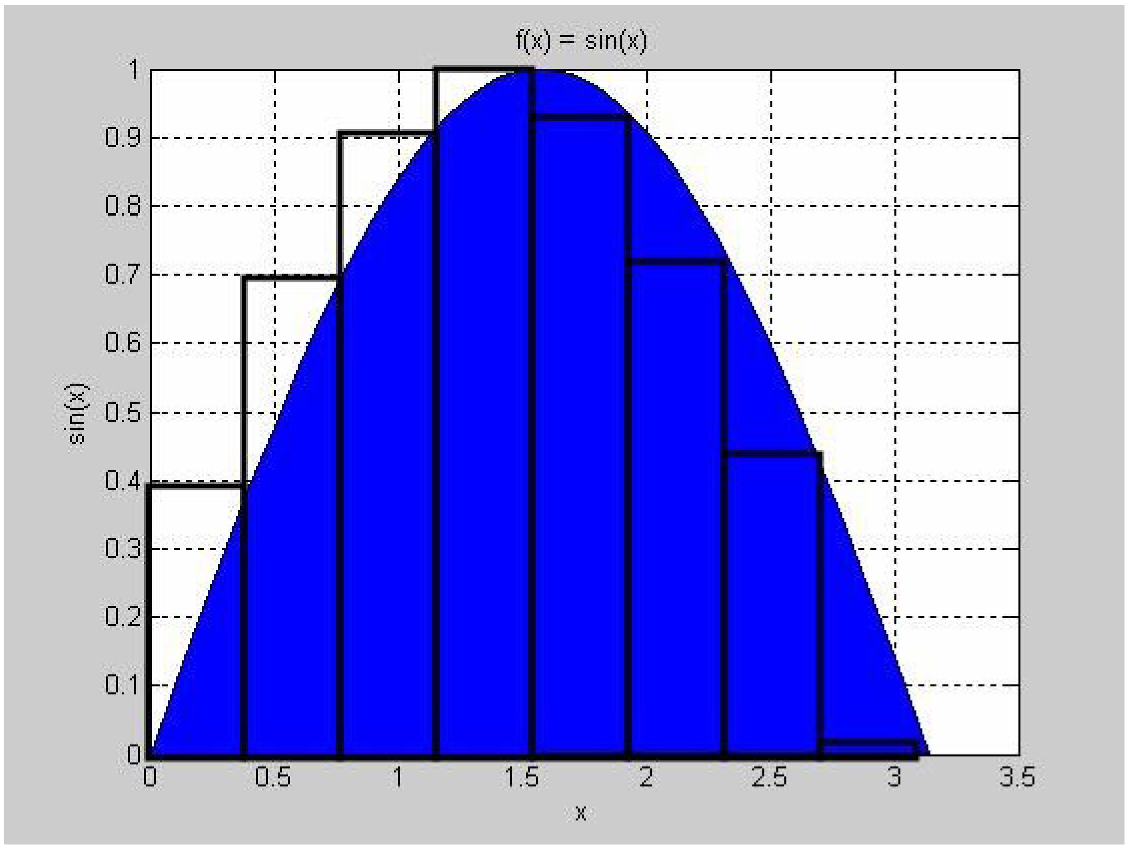

Figure 2: Approximating $\sin(x)$ with the Trapezoidal Rule

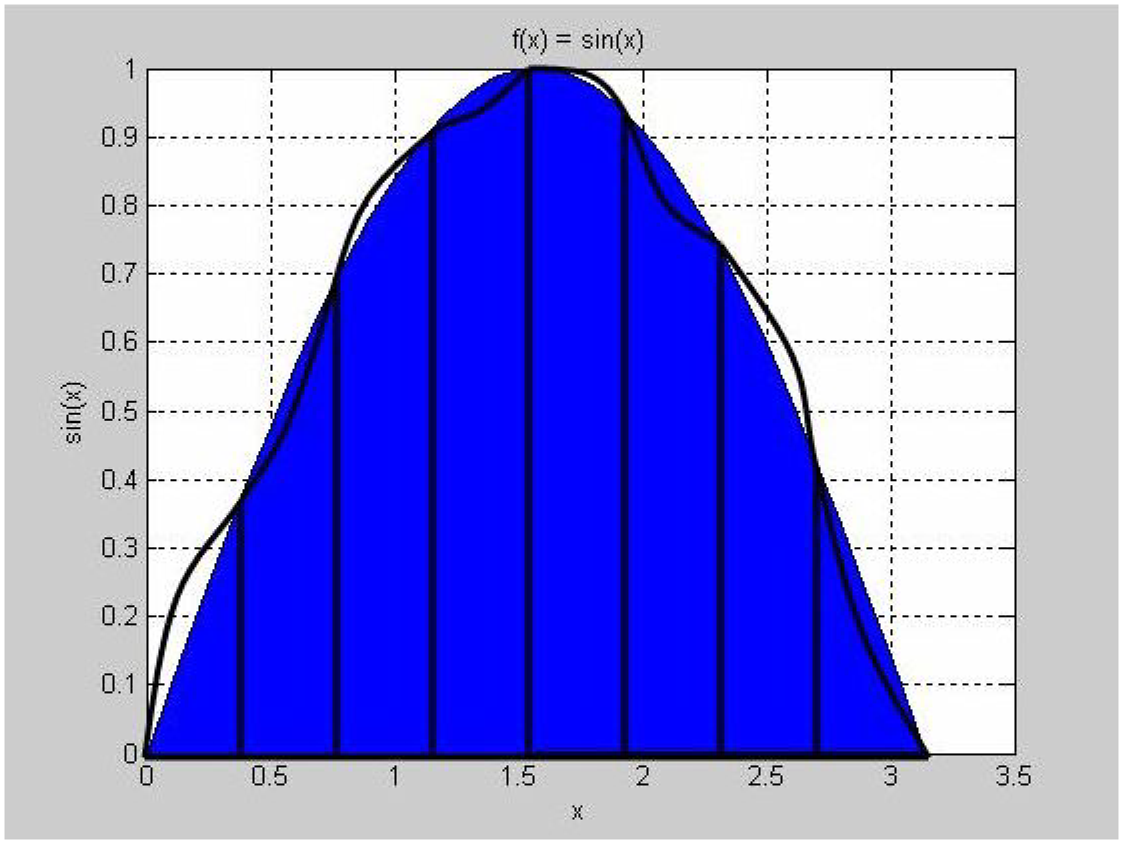

Figure 3: Approximating $\sin(x)$ with Simpson's Rule

(Disclaimer: The authors offer their sincerest apologies to the estate of Thomas Simpson for mangling his rule in Figure 3. Apparently the Trapezoid's aren't happy with us either.)

The choice between Simpson's Rule and the Trapezoidal Rule depends on several factors, including the shape of the original curve, the amount of error you can tolerate, and the amount of computing time you can devote.

You can understand most of this article's Java code without knowing anything about the formulas for Simpson's Rule and the Trapezoidal Rule. But if you're interested, both Simpson's Rule and the Trapezoidal Rule make use of four basic variables. The variables are illustrated in Figure 4.

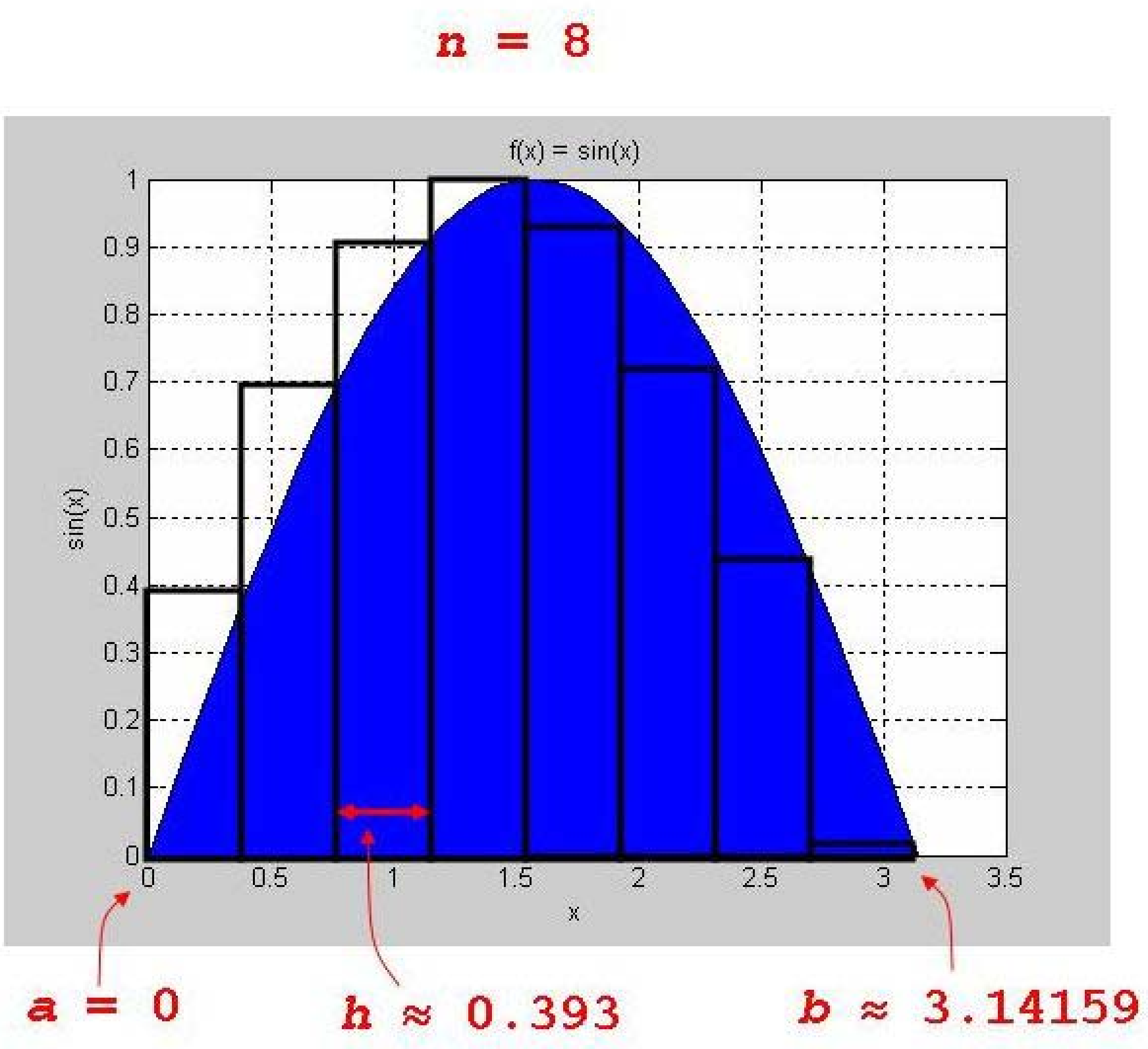

Figure 4: The variables used in applying the Trapezoidal Rule or Simpson's Rule

The formulas for the Trapezoidal Rule and Simpson's Rule are different but the procedures for applying these formulas are the same. Here's an outline that applies to both rules:

- Call a method, **`getN()`**, to obtain the value of **`n`**, the number of intervals.
- Call a method, **`getEndpoints()`**, to obtain the values of **`a`** and **`b`**, the endpoints for the definite integral.
- Call a method, **`calcH()`**, to calculate the value of **`h`**, the width of each interval.
- Call a method, **`calcArea()`**, to calculate the area under the curve according to the formula.
    - The **`calcArea()`** method uses the methods, **`calcXi()`** and **`calcFunction()`**, to assist in calculating the area under the curve.

The code in Listing 1 applies the Trapezoid Rule formula, and the code in Listing 2 applies the Simpson's Rule formula. If you're not interested in mathematical formulas, you can treat Listings 1 and 2 as black boxes. (Go ahead and ignore Listings 1 and 2. No one will be offended.)

```
public class TrapezoidRule
      {
      private double a; // the lower limit of the integration
      private double b; // the upper limit of the integration
      private int n; // the number of points for the integration

      public TrapezoidRule()
            {
            setA(0);
            setB(1);
            setN(10);
            }

      public TrapezoidRule(double a,double b,int n)
            {
            setA(a);
            setB(b);
            setN(n);
            }

      public double integrate()
            {
            int n = getN();
            double[] endpoints = getEndpoints();
            double h = calcH(endpoints[0],endpoints[1],n);
            double area = calcArea(n,h);
            return area;
            }

      private double getA()
            {
            return a;
            }

      private void setA(double a)
            {
            this.a = a;
            }

      private double getB()
            {
            return b;
            }

      private void setB(double b)
            {
            this.b = b;
            }

      private int getN()
            {
            return n;
            }

      private void setN(int n)
            {
            this.n = n;
            }
      private double[] getEndpoints()
            {
            double[] endpoints =
                  {
                  getA(),
                  getB(),
                  };
            return endpoints;
            }

      private double calcH(double a,double b,int n)
            {
            return (a + b) / (double)n;
            }

      private double calcArea(int n,double h)
            {
            double xi = 0.0;
            double area = 0.0;
            double factor = h / 2D;
            for(int i = 0;i <= n;++i)
                  {
                  xi = calcXi(i,h);
                  if(i == 0 || i == n)
                        area += calcFunction(xi);
                  else
                        area += 2D * calcFunction(xi);
                  }
            return factor * area;
            }

      private double calcXi(int i,double h)
            {
            return getA() + i * h;
            }

      private double calcFunction(double x)
            {
            return Math.sin(x);
            }
      }
```

Listing 1: The Trapezoid Rule class

```
public class SimpsonsRule
      {
      private double a; // the lower limit of the integration
      private double b; // the upper limit of the integration
      private int n; // the number of points for the integration

      public SimpsonsRule()
            {
            setA(0);
            setB(1);
            setN(10);
            }

      public SimpsonsRule(double a,double b,int n)
            {
            setA(a);
            setB(b);
            setN(n);
            }
```

```
    public double integrate()
          {
          int n = getN();
          if(!isEven(n))
                setN(++n);
          double[] endpoints = getEndpoints();
          double h = calcH(endpoints[0],endpoints[1],n);
          double area = calcArea(n,h);
          return area;
          }

    private double getA()
          {
          return a;
          }

    private void setA(double a)
          {
          this.a = a;
          }

    private double getB()
          {
          return b;
          }

    private void setB(double b)
          {
          this.b = b;
          }

    private int getN()
          {
          return n;
          }

    private void setN(int n)
          {
          this.n = n;
          }

    private double[] getEndpoints()
          {
          double[] endpoints =
                {
                getA(),
                getB(),
                };
          return endpoints;
          }

    private double calcH(double a,double b,int n)
          {
          return (a + b) / (double)n;
          }

    private double calcArea(int n,double h)
          {
          double xi = 0.0;
          double area = 0.0;
          double factor = h / 3D;
          for(int i = 0;i <= n;++i)
```

```
                  {
                  xi = calcXi(i,h);
                  if(i == 0 || i == n)
                        area += calcFunction(xi);
                  else if(isEven(i))
                        area += 2D * calcFunction(xi);
                  else
                        area += 4D * calcFunction(xi);
                  }
            return factor * area;
            }

      private double calcXi(int i,double h)
            {
            return getA() + i * h;
            }

      private double calcFunction(double x)
            {
            return Math.sin(x);
            }

      private boolean isEven(int n)
            {
            if(n % 2 == 0)
                  return true;
            else
                  return false;
            }
      }
```

Listing 2: The Simpson's Rule class

Listing 1 contains its own definitions of the basic methods (**`getN()`**, **`getEndpoints()`**, and so on). Listing 1 also contains a method named **`integrate()`** which combines calls to the basic methods and produces a numerical answer (an approximate area under a curve).

Similarly, Listing 2 contains definitions of the basic methods, and contains a big **`integrate()`** method. Hmm! Maybe we can take advantage of the similarity between Listings 1 and 2.

The client code (Listing 3) pulls everything together.

```
public class IntegrationApp
      {
      public static void main(String[] args)
            {
            if(args.length < 3)
                  {
                  System.out.println("Usage: Integration a b n");
                  System.exit(1);
                  }
            double a = Double.parseDouble(args[0]);
            double b = Double.parseDouble(args[1]);
            int n = Integer.parseInt(args[2]);
            String function = "f(x) = sin(x)";
            System.out.println("-- Numerical Integration --");
            System.out.println("Function: " + function);
            System.out.println("a = " + a);
            System.out.println("b = " + b);
            System.out.println("n = " + n);
            TrapezoidRule trapezoid = new TrapezoidRule(a,b,n);
            System.out.println("Trapezoid Rule: area = " + trapezoid.integrate());
            SimpsonsRule simpsons = new SimpsonsRule(a,b,n);
            System.out.println("Simpson's Rule: area = " + simpsons.integrate());
            }
      }
```

Listing 3: The IntegrationApp class – the client application

Sample output of the client code appears in Figure 5.

```
-- Numerical Integration --
Function: f(x) = sin(x)
a = 0.0
b = 3.14159265
n = 25
Trapezoid Rule: area = 1.9973674125516483
Simpson's Rule: area = 2.0000023725694533
```

Figure 5: The output of the numerical integration application

At this point, you may be asking yourself the following questions:

- Why is the Simpson's Rule result different from the Trapezoid Rule result?
- Why are the methods duplicated in each of the classes?
- Arrrg! Why are these guys using a math example?

Notice the duplication of methods in both the `TrapezoidRule` and `SimpsonsRule` classes. Almost all of the methods in the two classes are exactly the same. The only exception is in the `calcArea()` method that accounts for the differences between the Trapezoid Rule and Simpson's Rule algorithms. "*Don't Repeat Yourself*" is a good rule of thumb in programming, but Listings 1 and 2 contain lots of repetition. Listings 1 and 2 can certainly use some improvement.

(To answer the "Arrrg" question, we used a math example because we just happen to like math. If you've followed along with the authors in this design pattern series, you know that we always try to use practical examples. This Template Method pattern is the perfect opportunity for us to use a real math example to demonstrate a design pattern. So there!)

UML Diagram

Figure 6 shows the official UML diagram for the Template Method pattern.

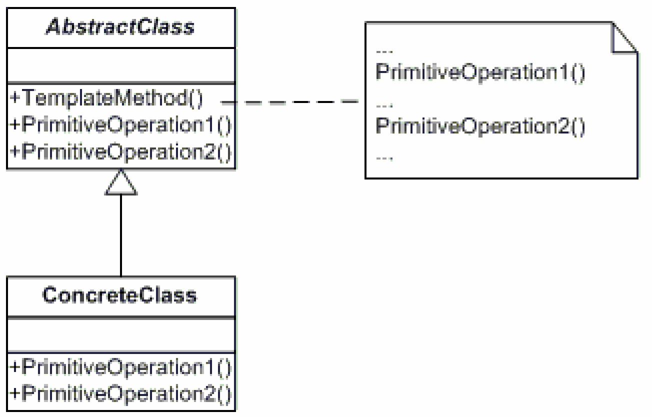

Figure 6: The UML diagram for the Template Method design pattern

The diagram contains two classes:

- The **`AbstractClass`** holds the **`templateMethod()`** method which defines the algorithm for a particular task. The **`templateMethod()`** may contain abstract and concrete **`primitiveOperation()`** methods that declare abstract methods for subclasses to implement and concrete methods for subclasses to share, respectively.
- Each **`ConcreteClass`** implements its own version of the abstract **`primitiveOperation()`** methods that were declared in **`AbstractClass`**.

Using the Template Method Design Pattern

Using Figure 6 as a guide, you can refactor Listings 1 and 2 using the UML diagram in Figure 7.

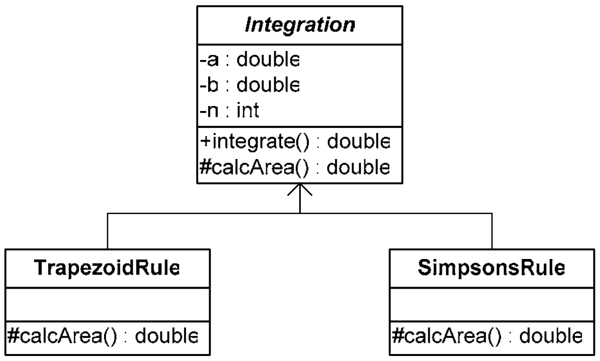

Figure 7: The UML diagram for the numerical integration application.

A new abstract base class named **`Integration`** encapsulates the methods that the Trapezoid Rule and Simpson's Rule algorithms share. The **`Integration`** class also declares any abstract methods that the two rules implement differently. (See Listing 4.)

```
public abstract class Integration
      {
      private double a; // the lower limit of the integration
      private double b; // the upper limit of the integration
      private int n; // the number of points for the integration

      public Integration()
            {
            setA(0);
            setB(1);
            setN(10);
            }

      public Integration(double a,double b,int n)
            {
            setA(a);
            setB(b);
            setN(n);
            }

      public double integrate()
            {
            int n = getN();
            if(!isEven(n) && this instanceof SimpsonsRule)
                  setN(++n);
            double[] endpoints = getEndpoints();
            double h = calcH(endpoints[0],endpoints[1],n);
            double area = calcArea(n,h);
            return area;
            }

      protected double getA()
            {
            return a;
            }

      protected void setA(double a)
            {
            this.a = a;
            }

      protected double getB()
            {
            return b;
            }

      protected void setB(double b)
            {
            this.b = b;
            }

      protected int getN()
            {
            return n;
            }

      protected void setN(int n)
            {
            this.n = n;
            }

      protected double[] getEndpoints()
            {
            double[] endpoints =
                  {
                  getA(),
                  getB(),
                  };
            return endpoints;
            }

      protected double calcH(double a,double b,int n)
            {
            return (a + b) / (double)n;
            }

      protected abstract double calcArea(int n,double h);

      protected double calcXi(int i,double h)
            {
            return getA() + i * h;
            }

      protected double calcFunction(double x)
            {
            return Math.sin(x);
            }

      protected boolean isEven(int n)
            {
            return true;
            }
      }
```

Listing 4: The Integration class (the AbstractClass in Figure 6)

The only abstract method in Listing 4 is **`calcArea()`**. The refactored **`TrapezoidRule`** and **`SimpsonsRule`** classes (Listings 5 and 6) are subclasses from the **`Integration`** base class. These refactored **`TrapezoidRule`** and **`SimpsonsRule`** classes implement their own versions of **`calcArea()`**. They share all of the methods defined in **`Integration`**. This example declares only one abstract method, but in general the Template Method design pattern can involve any number of abstract Java methods.

```
public class TrapezoidRule extends Integration
      {
      public TrapezoidRule()
            {
            super();
            }

      public TrapezoidRule(double a,double b,int n)
            {
            super(a,b,n);
            }

      protected double calcArea(int n,double h)
            {
            double xi = 0.0;
            double area = 0.0;
            double factor = h / 2D;
            for(int i = 0;i <= n;++i)
                  {
                  xi = calcXi(i,h);
                  if(i == 0 || i == n)
                        area += calcFunction(xi);
                  else
                        area += 2D * calcFunction(xi);
                  }
            return factor * area;
            }
      }
```

Listing 5. The TrapezoidRule class (a ConcreteClass from Figure 6)

```
public class SimpsonsRule extends Integration
      {
      public SimpsonsRule()
            {
            super();
            }

      public SimpsonsRule(double a,double b,int n)
            {
            super(a,b,n);
            }

      protected double calcArea(int n,double h)
            {
```

```
            double xi = 0.0;
            double area = 0.0;
            double factor = h / 3D;
            for(int i = 0;i <= n;++i)
                  {
                  xi = calcXi(i,h);
                  if(i == 0 || i == n)
                        area += calcFunction(xi);
                  else if(isEven(i))
                        area += 2D * calcFunction(xi);
                  else
                        area += 4D * calcFunction(xi);
                  }
            return factor * area;
            }

      protected boolean isEven(int n)
            {
            if(n % 2 == 0)
                  return true;
            else
                  return false;
            }
      }
```

Listing 6. The SimpsonsRule class (a ConcreteClass from Figure 6)

The original client code in Listing 3 works with the new Template Method pattern code in Listings 4, 5, and 6. This is an excellent example of the separation of an implementation from its interface.

Details, Details, Details

What happens under the hood in Listings 3 through 6? After getting the necessary variables from the command line (variables **`a`**, **`b`**, and **`n`**) and printing out some basic information to the console, the code in Listing 3 creates an instance of the **`TrapezoidRule`** class.

```
TrapezoidRule trapezoid = new TrapezoidRule(a,b,n);
```

The base class, **`Integration`**, stores the variables, **`a`**, **`b`**, and **`n`**. (Actually the **`Integration`** class passes these variables to the **`TrapezoidRule`** constructor, which immediately passes the variables back to the of **`Integration`** parent constructor. It's very slick.) After all the preliminaries, Listing 3 starts the big calculations by calling the **`integrate()`** method:

```
System.out.println("Trapezoid Rule: area = " + trapezoid.integrate());
```

Everything pivots on the **`integrate()`** method in Listing 4 (which corresponds to the template method in Figure 6). This all important **`integrate()`** method is defined in the **`Integration`** class as follows:

```
public double integrate()
      {
      int n = getN();
      if(!isEven(n) && this instanceof SimpsonsRule)
            setN(++n);
      double[] endpoints = getEndpoints();
      double h = calcH(endpoints[0],endpoints[1],n);
      double area = calcArea(n,h);
      return area;
      }
```

The calls to **`getN()`**, **`getEndpoints()`**, and **`calcH()`** are encapsulated in the base class, **`Integration`**. But the call to **`calcArea()`** is where everything changes. With an instance of the **`TrapezoidRule`** class, you get the **`calcArea()`** method defined in the **`TrapezoidRule`** class. And with an instance of the **`SimpsonsRule`** class, the program calls the **`SimpsonsRule`** class's **`calcArea()`** method.

... and One More Detail

The **`integrate()`** method contains a conditional statement:

```
if(!isEven(n) && this instanceof SimpsonsRule)
      setN(++n);
```

For Simpson's Rule to work correctly, the value of **`n`** ***must*** be even. So the **`Integration`** class has an **`isEven()`** method:

```
protected boolean isEven(int n)
      {
      return true;
      }
```

But this **`isEven()`** method simply returns the value of **`true`**! What's going on here?

The **`SimpsonsRule`** class overrides the **`isEven()`** method:

```
protected boolean isEven(int n)
      {
      if(n % 2 == 0)
            return true;
      else
            return false;
      }
```

Now this is useful. The **`isEven()`** method is a ***hook*** for subclasses to use as appropriate. The base class, **`Integrate`**, defines a vanilla version of the **`isEven()`** method. Then subclasses override the **`isEven()`** method with a more concrete version.

The only trick in the numerical integration application is to keep an instance of **`TrapezoidRule`** from calling the vanilla version of **`isEven()`**. That why the **`integrate()`** method's condition uses the **`instanceof`** operator.

Why Use the Template Method?

The Template Method eliminates the duplication of methods in separate, but related classes. This obeys the principle of code re-use, and leaves you with only one place to make changes in the code.

But the Template Method pattern has another important advantage. With this article's Template Method example, control of the numerical integration application moves from the individual classes (**`TrapezoidRule`** and **`SimpsonsRule`**) to a central class, **`Integration`**. The base class relies on the derived classes only when the base class needs the methods specific to those derived classes, i.e., for the implementation of the abstract methods.

Other Uses of the Template Method

The Template Method encapsulates all kinds of algorithms (not only math algorithms, like the algorithm in this article's example). Any application that uses a step-by-step procedure to accomplish a task is a candidate for the Template Method design pattern.

Resources

Design Patterns – Elements of Reusable Object-Oriented Software
http://www.awprofessional.com/bookstore/product.asp?isbn=0201633612&rl=1
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
ISBN 0-201-63361-2

Head First Design Patterns
http://www.oreilly.com/catalog/hfdesignpat/
Eric & Elisabeth Freeman
ISBN 0-596-00712-4

Object-Oriented Software Construction
http://vig.prenhall.com/catalog/academic/product/0,1144,0136291554.html,00.html
Bertrand Meyer
ISBN 0-13-629155-4

Data & Object Factory
http://www.dofactory.com/Patterns/Patterns.aspx

About the Authors

Barry Burd is a professor in the Department of Mathematics and Computer Science at Drew University in Madison, New Jersey. When he's not lecturing at Drew University, Dr. Burd leads training courses for professional programmers in business and industry. He has lectured at conferences in America, Europe, Australia and Asia. He is the author of several articles and books, including *Java For Dummies* and *Ruby on Rails For Dummies*, both published by Wiley.

Michael P. Redlich is a Senior Research Technician (formerly a Systems Analyst) at ExxonMobil Research & Engineering, Co. in Clinton, New Jersey with extensive experience in developing custom web and scientific laboratory applications. Mike also has experience as a Technical Support Engineer for Ai-Logix, Inc. where he developed computer telephony applications. As a member of the Amateur Computer Group of New Jersey (ACGNJ), he dedicates much of his free time facilitating the monthly ACGNJ Java Users Group and serving on the ACGNJ Board of Directors. Mike is the current ACGNJ President and has previously served as Secretary. He has a Bachelor of Science in Computer Science from Rutgers University. Mike's computing experience includes computer security, relational database design and development, object-oriented design and analysis, C/C++, Java, Visual Basic, FORTRAN, Pascal, MATLAB, HTML, XML, ASP, VBScript, and JavaScript in both the PC and UNIX environments.