Class Design

Christopher Simpkins
chris.simpkins@gatech.edu

Data Abstraction with Classes

Consider a concrete `Point` data type:

```
public class Point {
  public double x, y;
}
```

and an abstract `Point` data type:

```
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
  void setPolar(double r, double theta);
}
```

- The concrete `Point` exposes its implementation, the abstract `Point` hides it.
- Abstract `Point` expresses that it take two elements to define a point, concrete `Point` allows x and y to be set independently.

Data Abstraction with Classes

```
public interface Point {
  double getX();
  double getY();
  void setCartesian(double x, double y);
  double getR();
  double getTheta();
  void setPolar(double r, double theta);
}
```

- The abstract `Point` class is truly an absraction - its interface expresses the essence of pointness and hides its implementation.

Data abstraction isn't just making instance variables private and providing getters and setters.

Classes as Data Structures

```
public class Rectangle {
  public Point topLeft;
  public double height;
  public double width;
}
public class Circle {
  public Point center;
  public double radius;
}
public class Geometry {
  public double area(Object shape) throws NoSuchShapeException {
    if (shape instanceof Rectangle) {
      Rectangle r = (Rectangle)shape;
      return r.height * r.width;
    }
    else if (shape instanceof Circle) {
      Circle c = (Circle)shape;
      return Math.PI * c.radius * c.radius;
    }
    throw new NoSuchShapeException();
  }
}
```

Object-Oriented Classes

```
public interface Shape {
  public double area();
}
public class Rectangle implements Shape {
  private Point topLeft;
  private double height;
  private double width;
  public double area() { return height * width; }
}
public class Square implements Shape {
  private Point topLeft;
  private double side;
  public double area() { return side*side; }
}
public class Circle implements Shape {
  private Point center;
  private double radius;
  public double area() { return Math.PI * radius * radius; }
}
```

Data/Object Anti-symmetry

Look back at the two implementations of the shape family.

- In the procedural classes-as-data-structures implementation:
  - Adding a shape requires adding a new shape class and then touching every function in `Geometry`.
  - Adding a function only requires adding it to `Geometry` and coding it to work with each shape.
- In the object-oriented implementation:
  - Adding a class requires only creating a class that implements each of the functions in `Shape`.
  - Adding a function requires adding its declaration to `Shape`, and then adding a defintion to every class that implements `Shape`

Data/Object Anti-Symmetry

The observations above lead to two complementary general rules:

> *Procedural code (code using data structures) makes it easy to add new functions without changing the existing data structures. OO code, on the other hand, makes it easy to add new classes without changing existing functions.*

and

> *Procedural code makes it hard to add new data structures because all the functions must change. OO code makes it hard to add new functions because all the classes must change.*

Clean design requires knowing when to apply each style (will you be more likely to add new functions or new classes?). Don't drive every nail with the same hammer.

The Law of Demeter

A module should not know about the internal structure of an object it uses. Consider:

```
final String outputDir =
    ctxt.getOptions().getScratchDir().getAbsolutePath();
```

Code like this is a *train wreck* because it looks like a train of method calls on objects returned from a succession of methods.

Is this better?

```
Options opts = ctxt.getOptions();
File scratchDir = opts.getScratchDir();
final String outputDir = scratchDir.getAbsolutePath();
```

Maybe, but probably not.

- Internal structure is still exposed and relied upon.
- A protocol-ish interface is a design smell - the client of the `ctxt` object is trying to do something - give that something a name and represent it as a method

Hiding Internal Structure

What is the something that the client is doing with an absolute path?

```
String outFile = outputDir + "/" + className.replace('.', '/') +
    ".class";
FileOutputStream fout = new FileOutputStream(outFile);
BufferedOutputStream bos = new BufferedOutputStream(fout);
```

First, this code smells: multiple levels of abstraction are mixed together. But ultimately the client code is using the absolute path of the scratch directory to create a file in that directory.

Better OO design to let the `ctxt` object do this for us:

```
BufferedOutputStream bos = ctxt.createScratchFileStream(classFileName);
```

- Now the internal structure of the `ctxt` object is no longer exposed and is free to change without affecting client code.
- Client code is much cleaner: several messy lines replaced with one method call whose intent is crystal clear.

Data Transfer Objects and Active Data Objects

Data Transfer Objects (DTOs) are simple data structures useful for passing data between clients and servers, into and out of databases.

```
public class Person {
  private String name, email;
  public Person(String name, String email) {
    this.name = name; this.email = email;
  }
  public String getName() { return name; }
  public String getEmail() { return email; }
}
```

- Other than meeting the JavaBean spec, no need for private instance variables and getters. (This is one of Java's warts.)
- Sometimes a DTO will include methods like `save` and `find` that operate on the database in which the DTOs are stored. These are called active data objects (ADOs).
- Don't put business logic in an ADO. Create a separate class to hold business logic and let the ADO have a single responsbility: transferring data to and from a database.

Class Organization

A class should folow the standard Java organization:

- public static constants
- private static variables
- private instance variables
- public functions
- private helper functions right after the functions they serve (stepdown rule/newspaper metaphor)

Should nearly never have public instance variables, but they'd go right after the private instance variables.

Only valid reason to break encapsulation is to facilitate unit testing. Do this by giving protected or package access – the unit test should be in the same package as the class it tests.

Small Classes and the Single Responsiblity Principle

Classes should be small. A class with many methods is obviously large, but this one is small, right?

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

- Size for classes is measured in responsbilities, not lines of code.
- Vague name (SuperDashBoard) and implementing multiple types (JFrame, MetaDataUser) are two sings of excessive responsiblities.
- This class has two responsiblities: GUI (last focused component), and version numbers.

The Single Responsiblity Principle

Single Responsibility Principle (SRP): a class or module should have one, and only one, reason to change. SRP gives us:

- a definition of responsibility, and
- a guidelines for class size.

Classes should have one responsibility – one reason to change.

So our `SuperDashboard` from the previous slide is too big. Create another class to handle one of it's responsibilities.

Cohesion

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

  public int size() { return topOfStack; }

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

`Stack` is cohesive because:

- it has a small number of instance variables, and
- most of its methods use all of its instance variables.

A clean system will have many classes with nullary or unary methods.

Organize for Change

Consider this class which creates SQL statements.

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

If we add a new type of statement, we must change this class, which risks breaking other functionality of the class. This `Sql` class is open for modification when the system is extended – this is a design smell.

Open Closed Principle

Better OO design:

```
public abstract class Sql {
  public Sql(String table, Column[] columns)
  public abstract String generate();
}
public class CreateSql extends Sql {
  public CreateSql(String table, Column[] columns)
  @Override public String generate()
}
public class SelectSql extends Sql {
  public SelectSql(String table, Column[] columns)
  @Override public String generate()
}
...
```

These classes are open for extension and closed for modification.

- Extend with new functionality by adding a new subclass of `Sql`.
- Existing classes need not be touched.