Principles of Software Construction: Objects, Design, and Concurrency

Part 2: Class-level design

'tis a gift to be simple, *or* Cleanliness is next to godliness

Josh Bloch **Charlie Garrod** Darya Melicher

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Reading due today: none!
- Homework 4a due Thursday at 11:59 p.m.
  - Mandatory design review meeting before the homework deadline
- PA voter registration deadline: Tuesday, October 9th

Key concepts from last Tuesday

Assign object responsibility using interaction diagrams

- For a given system-level operation, create an object interaction diagram at the *implementation-level* of abstraction
- Implementation-level concepts:
  - Implementation-like method names
  - Programming types
  - Helper methods or classes
  - Artifacts of design patterns

Heuristics for responsibility assignment

- Controller heuristic
- Information expert heuristic
- Creator heuristic

Object-level artifacts of this design process

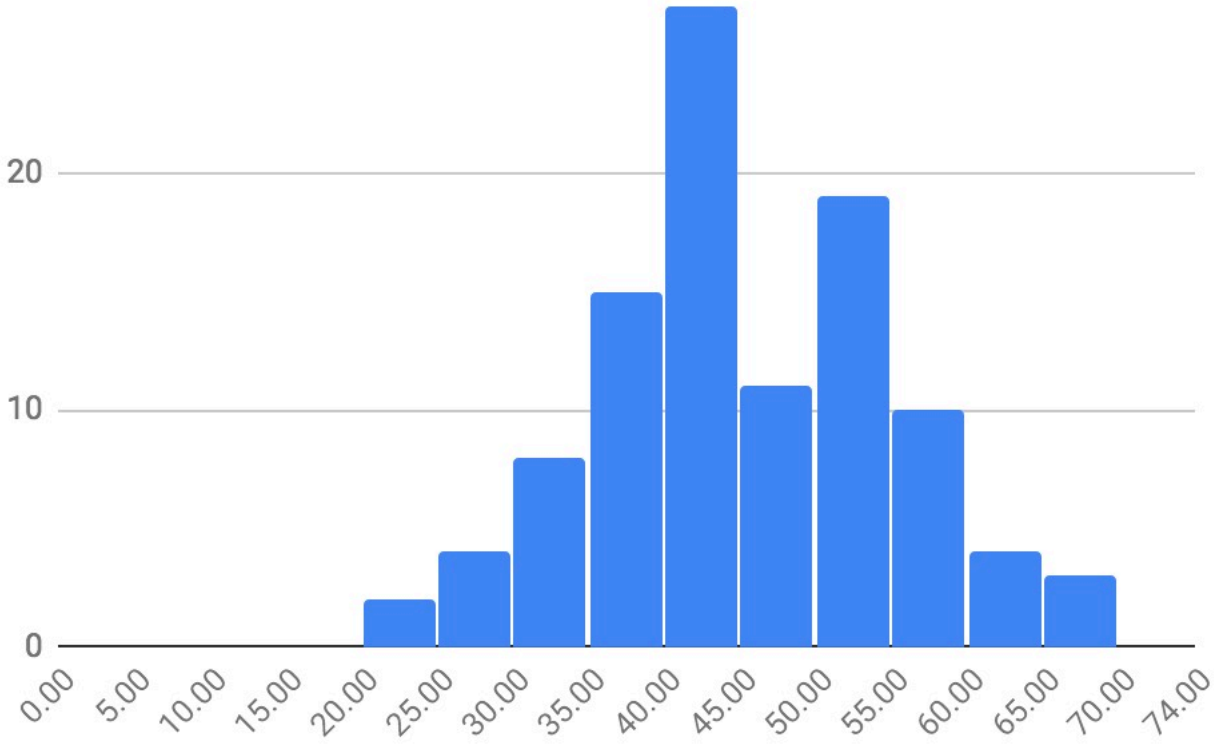
- Object interaction diagrams add methods to objects
  - Can infer additional data responsibilities
  - Can infer additional data types and architectural patterns
- Object model aggregates important design decisions
  - Is an implementation guide

Today

- Midterm exam post-mortem
- Homeworks 2 and 3 post-mortem

Midterm exam results

- Average: 45 out of 74
- Standard deviation: 10



Not so big data

In this problem, you will demonstrate your understanding of two design patterns by showing the design of a data processing application that allows a flexible choice of database, while avoiding code duplication. Using your design, one should be able to add support for a new database with minimal changes to the existing code.

Regardless of the database being used, the application must support a single method called `loadData()` which connects to a database, executes a database query, and disconnects from a database. All database systems provide an API that supports these operations, but they differ in how they implement them. Your solution should allow a flexible implementation of these operations for each database system, i.e., a flexible `connect()` method, an `executeQuery()` method, and a `disconnect()` method for each database system.

A mini-puzzler…

- How long will the following program take to run?

```
public static void main(String[] args) {
    long startTime = System.nanoTime();
    for (int i = 0; i <= Integer.MAX_VALUE; i++) {

    }
    long endTime = System.nanoTime();
    long millis = (endTime - startTime) / 1_000_000;
    System.out.println("The program took "
          + millis + " milliseconds.");
}
```

A mini-puzzler…

- How long will the following program take to run?

```
public static void main(String[] args) {
    long startTime = System.nanoTime();
    for (int i = 0; i <= Integer.MAX_VALUE; i++) {

    }
    long endTime = System.nanoTime();
    long millis = (endTime - startTime) / 1_000_000;
    System.out.println("The program took "
          + millis + " milliseconds.");
}
```

a) Faster than an eye-blink.
b) Get some coffee.
c) Go to lunch.
d) Something else…

Watch it go!

An `int` is always <= to `Integer.MAX_VALUE`

- How long will the following program take to run?

```
public static void main(String[] args) {
    long startTime = System.nanoTime();
    for (int i = 0; i <= Integer.MAX_VALUE; i++) {

    }
    long endTime = System.nanoTime();
    long millis = (endTime - startTime) / 1_000_000;
    System.out.println("The program took "
          + millis + " milliseconds.");
}
```

a) Faster than an eye-blink.
b) Get some coffee.
c) Go to lunch.
d) Something else…

Home on the range preliminaries: int overflow

```
/**
* Returns an immutable list consisting of consecutive Integers in a
* specified range from start (inclusive) to stop (exclusive). The
* returned list logically contains (stop - start) elements (as
* reported by its size method) but its memory consumption is constant
* regardless of its logical size.
*
* @param start the (inclusive) initial value of the range
* @param stop the (exclusive) upper bound of the range
* @throws IllegalArgumentException if stop < start or if
* (stop - start) would be greater than Integer.MAX_VALUE
*/
public static List<Integer> range(int start, int stop) { ... }
```

Home on the range preliminaries: int overflow

```
/**
* Returns an immutable list consisting of consecutive Integers in a
* specified range from start (inclusive) to stop (exclusive). The
* returned list logically contains (stop - start) elements (as
* reported by its size method) but its memory consumption is constant
* regardless of its logical size.
*
* @param start the (inclusive) initial value of the range
* @param stop the (exclusive) upper bound of the range
* @throws IllegalArgumentException if stop < start or if
* (stop - start) would be greater than Integer.MAX_VALUE
*/
public static List<Integer> range(int start, int stop) {
    if (stop < start || (stop – start) > Integer.MAX_VALUE)
    ...
}
```

Home on the range preliminaries: int overflow

```
/**
* Returns an immutable list consisting of consecutive Integers in a
* specified range from start (inclusive) to stop (exclusive). The
* returned list logically contains (stop - start) elements (as
* reported by its size method) but its memory consumption is constant
* regardless of its logical size.
*
* @param start the (inclusive) initial value of the range
* @param stop the (exclusive) upper bound of the range
* @throws IllegalArgumentException if stop < start or if
* (stop - start) would be greater than Integer.MAX_VALUE
*/
public static List<Integer> range(int start, int stop) {
    if (stop < start || ((long) stop – start) > Integer.MAX_VALUE)
    ...
}
```

Home on the range preliminaries: int overflow

```
/**
* Returns an immutable list consisting of consecutive Integers in a
* specified range from start (inclusive) to stop (exclusive). The
* returned list logically contains (stop - start) elements (as
* reported by its size method) but its memory consumption is constant
* regardless of its logical size.
*
* @param start the (inclusive) initial value of the range
* @param stop the (exclusive) upper bound of the range
* @throws IllegalArgumentException if stop < start or if
* (stop - start) would be greater than Integer.MAX_VALUE
*/
public static List<Integer> range(int start, int stop) {
    if (stop < start || (stop – start) < 0)
    ...
}
```

Home on the range preliminaries: non-functional spec

```
/**
* Returns an immutable list consisting of consecutive Integers in a
* specified range from start (inclusive) to stop (exclusive). The
* returned list logically contains (stop - start) elements (as
* reported by its size method) but its memory consumption is constant
* regardless of its logical size.
*
* @param start the (inclusive) initial value of the range
* @param stop the (exclusive) upper bound of the range
* @throws IllegalArgumentException if stop < start or if
* (stop - start) would be greater than Integer.MAX_VALUE
*/
public static List<Integer> range(int start, int stop) { ... }
```

Home on the range preliminaries: non-functional spec

```
/**
* Returns an immutable list consisting of consecutive Integers in a
* specified range from start (inclusive) to stop (exclusive). The
* returned list logically contains (stop - start) elements (as
* reported by its size method) but its memory consumption is constant
* regardless of its logical size.
*
* @param start the (inclusive) initial value of the range
* @param stop the (exclusive) upper bound of the range
* @throws IllegalArgumentException if stop < start or if
* (stop - start) would be greater than Integer.MAX_VALUE
*/
public static List<Integer> range(int start, int stop) {
    if (…) { throw new IllegalArgumentException(); }
    List<Integer> result = new ArrayList<>();
    for (int i = start; i < stop; i++) {
        result.add(i);
    }
    return result;
}
```

Home on the range preliminaries: non-functional spec

```
/**
* Returns an immutable list consisting of consecutive Integers in a
* specified range from start (inclusive) to stop (exclusive). The
* returned list logically contains (stop - start) elements (as
* reported by its size method) but its memory consumption is constant
* regardless of its logical size.
*
* @param start the (inclusive) initial value of the range
* @param stop the (exclusive) upper bound of the range
* @throws IllegalArgumentException if stop < start or if
* (stop - start) would be greater than Integer.MAX_VALUE
*/
public static List<Integer> range(int start, int stop) {
    if (…) { throw new IllegalArgumentException(); }
    int values[] = new int[Integer.MAX_VALUE];
    for (int i = start; i < stop; i++) {
        values[start-i] = i;
    }
    …
}
```

Metrics of software quality, i.e., *design goals*

Functional correctness	Adherence of implementation to the specifications
Robustness	Ability to handle anomalous events
Flexibility	Ability to accommodate changes in specifications
Reusability	Ability to be reused in another application
Efficiency	Satisfaction of speed and storage requirements
Scalability	Ability to serve as the basis of a larger version of the application
Security	Level of consideration of application security

Source: Braude, Bernstein, Software Engineering. Wiley 2011

A Collections aside…

```
/**
* Returns an immutable list consisting of consecutive Integers in a
* specified range from start (inclusive) to stop (exclusive). The
* returned list logically contains (stop - start) elements (as
* reported by its size method) but its memory consumption is constant
* regardless of its logical size.
*
* @param start the (inclusive) initial value of the range
* @param stop the (exclusive) upper bound of the range
* @throws IllegalArgumentException if stop < start or if
* (stop - start) would be greater than Integer.MAX_VALUE
*/
public static List<Integer> range(int start, int stop) {
    if (…) { throw new IllegalArgumentException(); }
    int values[] = new int[Integer.MAX_VALUE];
    for (int i = start; i < stop; i++) {
        values[start-i] = i;
    }
    return Collections.unmodifiableList(Arrays.asList(values));
}
```

Generic asides…

- A generic list implementation is not necessary

```
public static List<Integer> range(int start, int stop) { … }


class RangeList implements AbstractList<Integer> {
    …
}
```

Generic asides…

- A generic list implementation is not necessary

```
public static List<Integer> range(int start, int stop) { … }


class RangeList implements AbstractList<Integer> {
    …
}

class RangeList<E> implements AbstractList<E> {
    …
    public E get(int index) { ...; return ???; }
}
```

Generic asides…

- A generic list implementation is not necessary

```
public static List<Integer> range(int start, int stop) { … }


class RangeList implements AbstractList<Integer> {
    …
}

class RangeList<E> implements AbstractList<E> {
    …
    public E get(int index) { ...; return ???; }
}

class RangeList<E> implements AbstractList<Integer> {
    …
}
```

On to the sample solutions…

Are there any design patterns in my solutions?

Extending `AbstractList` is the template method pattern

```
abstract E   get(int i);
abstract int size();
boolean      set(int i, E e);          // pseudo-abstract
boolean      add(E e);                 // pseudo-abstract
boolean      remove(E e);              // pseudo-abstract
boolean      addAll(Collection<? extends E> c);
boolean      removeAll(Collection<?> c);
boolean      retainAll(Collection<?> c);
boolean      contains(E e);
boolean      containsAll(Collection<?> c);
void         clear();
boolean      isEmpty();
Iterator<E>  iterator();
Object[]     toArray()
<T> T[]      toArray(T[] a);
…
```

Testing the `range()` method

- You must test both `range(…)` and the returned list

Today

- Midterm exam post-mortem
- Homeworks 2 and 3 post-mortem

Enums *(review)*

- Java has object-oriented enums
- In simple form, they look just like C enums:

```
public enum Planet { MERCURY, VENUS, EARTH, MARS,
                     JUPITER, SATURN, URANUS, NEPTUNE }
```

- But they have many advantages [EJ Item 34]!
  - Compile-time type safety
  - Multiple enum types can share value names
  - Can add or reorder without breaking constants
  - High-quality `Object` methods
  - Screaming fast collections (`EnumSet`, `EnumMap`)
  - Can easily iterate over all constants of an enum

You can add data to enums

```
public enum Planet {
    MERCURY(3.302e+23, 2.439e6), VENUS (4.869e+24, 6.052e6),
    EARTH(5.975e+24, 6.378e6), MARS(6.419e+23, 3.393e6);

    private final double mass;   // In kg.
    private final double radius; // In m.

    private static final double G = 6.67300E-11;

    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }

    public double mass()   { return mass; }
    public double radius() { return radius; }
    public double surfaceGravity() {
        return G * mass / (radius * radius);
    }
}
```

You can add behavior too

```
public enum Planet {
    ... // As on previous slide

     public double surfaceWeight(double mass) {
         return mass * surfaceGravity; // F = ma
     }
}
```

Watch it go!

```
public static void main(String[] args) {
   double earthWeight = Double.parseDouble(args[0]);
   double mass = earthWeight / EARTH.surfaceGravity();

   for (Planet p : Planet.values()) {
      System.out.printf("Your weight on %s is %f%n",
                        p, p.surfaceWeight(mass));
   }
}
```

```
$ java Planet 180
Your weight on MERCURY is 68.023205
Your weight on VENUS is 162.909181
Your weight on EARTH is 180.000000
Your weight on MARS is 68.328719
```

You can even add constant-specific behavior

- Each constant can have its own override of a method
  - Don't do this unless you have to
  - If adding data is sufficient, do that instead

```
public interface Filter {
    Image transform(Image original);
}

public enum InstagramFilter implements Filter {
  EARLYBIRD {public Image transform(Image original) { ... }},
  MAYFAIR   {public Image transform(Image original) { ... }},
  AMARO     {public Image transform(Image original) { ... }},
  RISE      {public Image transform(Image original) { ... }};
}
```

See Effective Java Items 34 - 38 for more information

A simple solution to HW 2 and 3

Lessons (practical)

- Choose low level abstractions that make higher level tasks easy
- When you want to represent a fixed set of values known at compile time, consider enums
- If users need to extend set consider emulated extensible enum
- Bit twiddling should be part of every programmers tool set
  - Don’t overuse it…
  - But do consider it even when performance doesn’t demand it

Lessons (philosophical)

- Good habits matter
  - "The way to write a perfect program is to make yourself a perfect programmer and then just program naturally." – Watts S. Humphrey, 1994
- Don't just hack it up and say you'll fix it later
  - You probably won't
  - but you will get into the habit of just hacking it up
  - Also it's way more fun to work on nice, well-structured code
- Even small design decisions matter
  - If your code is getting ugly, go back to the drawing board
  - "A week of coding can often save a whole hour of thought."
- Strive for clarity
  - It's not enough to be merely correct; aim for clearly correct