Principles of Software Construction

’tis a Gift to be Simple or Cleanliness is Next to Godliness

Midterm 1 and Homework 3 Post-Mortem

Josh Bloch  Charlie Garrod
Administrivia

- Homework 4a due Thursday, 11:59 p.m.
  - Design review meeting is mandatory
- Office hours schedule this week TBD
- Final exam scheduled: Friday, December 16th, 5:30-8:30 pm
Key concepts from last Tuesday...
Understanding system behavior with sequence diagrams

- A *system sequence diagram* is a model that shows, for one scenario of use, the sequence of events that occur on the system’s boundary
- Design goal: Identify and define the interface of the system
  - Two components: A user and the overall system
Formalize system behavior with behavioral contracts

• A system behavioral contract describes the pre-conditions and post-conditions for some operation identified in the system sequence diagrams
  – System-level textual specifications, like software specifications
Using interaction diagrams to assign object responsibility

- 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
Another design principle: Minimize conceptual weight

- Label the concepts for a proposed object
  - Related to representational gap and cohesion
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
Creating an object model

• Extract data, method names, and types from interaction diagrams
  – Include implementation details such as visibilities
Outline

• A formal design process (conclusion)
• Midterm exam post-mortem
• Permutation generator post-mortem
• Cryptarithm post-mortem
Exam grading fortune cookies...
Exam grading fortune cookies...

Mistakes are the portals of discovery.
Exam grading fortune cookies...

Mistakes are the portals of discovery.

It is more difficult to judge oneself than to judge others.
Exam grading fortune cookies...

Mistakes are the portals of discovery.

It is more difficult to judge oneself than to judge others.

The axe soon forgets, but the tree always remembers.
Midterm exam results

- Average: 38 out of 77
- Standard deviation: 15
Outline

• A formal design process (conclusion)
• Midterm exam post-mortem
• Permutation generator post-mortem
• Cryptarithm post-mortem
Anyone know a simpler expression for this?

```java
if (myDog.hasFleas()) {
    return true;
} else { 
    return false;
}
```
Hint: it’s not this

```java
return myDog.hasFleas() ? true : false;
```
Please do it this way from now on

*We reserve the right to deduct points if you don’t*

```python
return myDog.hasFleas();
```
Also, we saw some hash functions like these

```java
return 31 * x + 31 * y;   // Multiplication doesn’t help!
return 31 * x + 32 * y;   // Multiplication hurts!
return Objects.hash(map);  // Objects.hash unnecessary!
```
Here's how these should look

```java
return 31 * x + 31 * y;
return 31 * x + 32 * y;
return Objects.hash(map);
```

```java
return 31 * x + y;
return 31 * x + y;
return map.hashCode();
```
What should a hash code look like, in general?

- **Single-field object**
  - `field.hashCode()`

- **Two-field object**
  - `31*field1.hashCode() + field0.hashCode()`

- **3-field object**
  - `31*(31*field2.hashCode() + field1.hashCode) + field0.hashCode`
  - `= 31^2 * field2.hashCode() + 31 * field1.hashCode() + field0.hashCode`

- **N-field object**
  - Repeatedly multiply total by 31 and add in next field
    - `= \Sigma 31^i \cdot \text{hashCode(field}_i)`
    - Alternatively: `Objects.hash(field0, field1, ... fieldN)`

- **For much more information, see Effective Java Item 9**
We saw a lot of public enum types on the exam

- Enums appropriate **only** if you know *all* values at compile time
  - They are *not* appropriate if you’ll be adding values at run time
- There is the **extensible enum pattern** (EJ Item 34)
  - Enum type(s) and class(es) implement a common interface
  - But this is *not* a good match for this problem!
  - Why not?
We saw a lot of public enum types on the exam

- Enums appropriate **only** if you know *all* values at compile time
  - They are **not** appropriate if you’ll be adding values at run time
- There is the *extensible enum pattern* (EJ Item 34)
  - Enum type(s) and class(es) implement a common interface
  - But this is **not** a good match for this problem!
  - The resulting type is not a value type; equals & hashCode won’t work
  - You cannot override equals and hashCode in an enum
  - Could fix using *instance control*, but the result would be a mess
We saw many solutions that used trees

```java
public abstract class Unit {
    private static class BaseType extends Unit { ... }
    public static final Unit METER = new BaseType(...);
    public static final Unit KILOGRAM = new BaseType(...);
    public static final Unit SECOND = new BaseType(...);

    private enum Op { MULTIPLY, DIVIDE }
    private static class DerivedType extends Unit {
        private final Unit left, right;
        private final Op op;
        ...
    }

    public Unit multiply(Unit other) {
        return new DerivedType(...);
    }
}
```

What’s wrong with this representation?
Trees describe the expression, not the result!

- Work fine for multiplication and division
- Awful for equals, hashCode, and toString
  - Infinitely many trees represent the same unit!
  - We want to erase construction process, not highlight it
- What are the key components of an SI unit?
  - The answer dictates an appropriate internal representation
A good, basic solution – fields and constructor

```java
public class Unit {
    /** Representation: exponents on the base units in this unit */
    private final int mExp;
    private final int kgExp;
    private final int sExp;

    private Unit(int mExp, int kgExp, int sExp) {
        this.mExp = mExp;
        this.kgExp = kgExp;
        this.sExp = sExp;
    }

    // Base Units
    public static final Unit METER = new Unit(1, 0, 0);
    public static final Unit KILOGRAM = new Unit(0, 1, 0);
    public static final Unit SECOND = new Unit(0, 0, 1);
    public static final Unit UNITLESS = new Unit(0, 0, 0);
}
```
A good, basic solution – Object methods

```java
@override public boolean equals(Object o) {
    if (!(o instanceof Unit)) return false;
    Unit u = (Unit) o;
    return u.mExp == mExp && u.kgExp == kgExp && u.sExp == sExp;
}

@override public int hashCode() {
    return 31 * 31 * mExp + 31 * kgExp + sExp;
}

@override public String toString() {
    return (str("m", mExp) + str("kg", kgExp) + str("s", sExp)).trim();
}

private static String str(String sym, int exp) {
    switch (exp) {
        case 0: return "";
        case 1: return sym + " ";
        default: return String.format("%s^%d ", sym, exp);
    }
}
```
A good, basic solution – arithmetic methods

```java
public Unit times(Unit other) {
    return new Unit(
        mExp + other.mExp, kgExp + other.kgExp, sExp + other.sExp);
}

public Unit dividedBy(Unit other) {
    return new Unit(
        mExp - other.mExp, kgExp - other.kgExp, sExp - other.sExp);
}

public Unit squared() {  // Convenience method; not required
    return this.times(this);
}
```
A good, basic solution – Part b: derived units

```java
public static final Unit HERTZ = UNITLESS.dividedBy(SECOND);
public static final Unit NEWTON = KILOGRAM.times(METER)
                                .dividedBy(SECOND.squared());
public static final Unit PASCAL = NEWTON.dividedBy(METER.squared());
public static final Unit JOULE = NEWTON.times(METER);
public static final Unit WATT = JOULE.dividedBy(SECOND);

private static final Map<Unit, String> symbols = new HashMap<>(
        static {
            symbols.put(HERTZ,  "Hz");
            symbols.put(NEWTON, "N");
            symbols.put(PASCAL, "Pa");
            symbols.put(JOULE,  "J");
            symbols.put(WATT,   "W");
        }
)

@Override public String toString() {
    String result = symbols.get(this);
    if (result != null) return result;
```
A more flexible solution – fields and constructor

```java
public class Unit {

    private enum Base { m, kg, s } // Names must be actual symbols!
    private static final Base[] BASES = Base.values(); // Optimization

    private final Map<Base, Integer> exponents;

    private Unit(Base base) {
        exponents = new EnumMap<Base, Integer>(Base.class);
        for (Base b : BASES) {
            exponents.put(b, 0);
        }
        if (base != null) {
            exponents.put(base, 1);
        }
    }

    public static final Unit METER = new Unit(Base.m);
    public static final Unit KILOGRAM = new Unit(Base.kg);
    public static final Unit SECOND = new Unit(Base.s);

    public static final Unit UNITLESS = new Unit((Base) null);
}
```
A more flexible solution – Object methods

```java
@override public boolean equals(Object o) {
    return o instanceof Unit && ((Unit) o).exponents.equals(exponents);
}

@override public int hashCode() {
    return exponents.hashCode();
}

@override public String toString() {
    result = "";
    for (Base b : BASES) {
        int exp = exponents.get(b);
        if (exp == 1) {
            result += b + " ";
        } else if (exp != 0) {
            result += String.format("%s%^d ", b, exp);
        }
    }
    return result.trim();
}
```
A more flexible solution – arithmetic methods

```java
public Unit times(Unit other) {
    return timesOrDiv(other, 1);
}

public Unit dividedBy(Unit other) {
    return timesOrDiv(other, -1);
}

private Unit timesOrDiv(Unit other, int op) {
    Map<Base, Integer> newExps = new EnumMap<>(Base.class);
    for (Base b : BASES)
        newExps.put(b, exponents.get(b) + op * other.exponents.get(b));
    return new Unit(newExps);
}

private Unit(Map<Base, Integer> exponents) {
    this.exponents = exponents;
}
```
Some solutions were correct but repetitious

• Repetition isn’t just inelegant, it’s toxic
• Avoiding repetition is essential to good programming
• Provides not just elegance, but quality
• Ease of understanding aids in
  – Establishing correctness
  – Maintaining the code
• If code is repeated, each bug must be fixed repeatedly
  – If you forget to fix one occurrence, program is subtly broken
Outline

• A formal design process (conclusion)
• Midterm exam post-mortem
• Permutation generator post-mortem
• Cryptarithmetic post-mortem
Design comparison for permutation generator

• Template Method pattern
  – Easy to code
  – Ugly to use

• Command pattern
  – Easy to code
  – Reasonably pretty to use

• Iterator pattern
  – Tricky to code because algorithm is recursive and Java lacks *yield iterators*
  – Gorgeous to use

• Performance of all three is similar
A complete (!), general-purpose permutation generator using the command pattern
How do you test a permutation generator?

Make a list of items to permute (integers should do nicely)

For each permutation of the list {
    Check that it’s actually a permutation of the list
    Check that we haven’t seen it yet
    Put it in the set of permutations that we have seen
}

Check that the set of permutations we’ve seen has right size (n!)

Do this for all reasonable values of n, and you’re done!
And now, in code – this is the whole thing!

```java
static void exhaustiveTest(int size) {
    List<Integer> list = new ArrayList<>(size);
    for (int i = 0; i < size; i++)
        list.add(i);
    Set<Integer> elements = new HashSet<>(list);
    Set<List<Integer>> alreadySeen = new HashSet<>();
    doForAllPermutations(list, (perm) -> {
        Assert.assertEquals(perm.size(), size);
        Assert.assertEquals(new HashSet(perm), elements);
        Assert.assertFalse("Duplicate", alreadySeen.contains(perm));
        alreadySeen.add(new ArrayList<>(perm));
    });
    Assert.assertEquals(alreadySeen.size(), factorial(size));
}

@Test public void test() {
    for (int size = 0; size <= 10; size++)
        exhaustiveTest(size);
}
```
Pros and cons of exhaustive testing

• Pros and cons of exhaustive testing
  + Gives you absolute assurance that the unit works
  + Exhaustive tests can be short and elegant
  + You don’t have to worry about what to test
  - Rarely feasible; Infeasible for:
    • Nondeterministic code, including most concurrent code
    • Large state spaces

• If you can test exhaustively, do!
• If not, you can often approximate it with random testing
Outline

• A formal design process (conclusion)
• Midterm exam post-mortem
• Permutation generator post-mortem
• Cryptarithm post-mortem
A fast, fully functional cryptarithm solver in 6 slides

To refresh your memory, here’s the grammar

cryptarithm ::= <expr> "=" <expr>
expr ::= <word> [<operator> <word>]*
word ::= <alphabetic-character>+
operator ::= "+" | "-" | "*"
Cryptarithm class (1) - fields
Cryptarithm class (2) - constructor
Parsing a word into an expression

$$(((M \times 10 + O) \times 10 + N) \times 10 + E) \times 10 + Y$$

$$= M \times 10^4 + O \times 10^3 + N \times 10^2 + E \times 10 + Y$$
Cryptarithmetic class (3) - word parser
Cryptarithm class (4) – operator parser
Cryptarithm class (5) – solver
Cryptarithm class (6) - helper functions
Cryptarithm solver command line program
Conclusion

• Good habits really 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

• Representations matter! Choose carefully.
  – If your code is getting ugly, think again
  – “A week of coding can often save a whole hour of thought.”

• Not enough to be merely correct; code must be clearly correct
  – Nearly correct is right out.