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

School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 4a due Thursday, 11:59 p.m.
- Design review meeting is mandatory
  - But we expect it to be really helpful
  - feedback is a wonderful thing

Key concepts from Tuesday…

- A formal design process
- Domain model – concepts
- System sequence diagram – behaviors
- Object interaction diagram – object responsibilities
- Object model – system architecture

Pop Quiz - Anyone know a simpler expression for this?

```
if (whatever.whoKnows()) {
    return true;
} else {
    return false;
}
```

It’s not rocket science

```
return whatever.whoKnows();
```

- **Please do it this way from now on**
  - We reserve the right to deduct points if you don’t

Pop Quiz 2 - What's wrong with this hash function?

```
@Override public int hashCode() {
    return 0;
}
```

DEMO

Constant hash functions

- It is said that some of our fine TA's said they were OK on exams
- **They are not OK anywhere!**
- Here's what *Effective Java* has to say (Item 9)

```
// The worst possible legal hash function - never use!
@Override public int hashCode() { return 42; }
```

- While they obey the letter of the spec, they violate its spirit
  - Unequal objects should generally have unequal hash codes
- **In the future there will be a penalty on exams**


I will not write hashCode() that returns a constant.
I will not write hashCode() that returns a constant.
I will not write hashCode() that returns a constant.
I will not write hashCode() that returns a constant.
I will not write hashCode() that returns a constant.
I will not write hashCode() that returns a constant.
I will not write hashCode() that returns a constant.
I will not write hashCode() that returns a constant.
TXT2PIC.COM

So what should a hash code look like?

- 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
  - = $\Sigma\ 31^i \cdot \text{hashCode}(\text{field}_i)$
- For much more information, see *Effective Java* Item 9

Outline

- “Mixed messages” post-mortem
- “Are you my type” post-mortem
- Permutation generator post-mortem
- Cryptarithm post-mortem

“Mixed messages” AKA true-and-false questions

- One **advantage** of using a design pattern is that that it makes programs **easier** to understand.
- One **disadvantage** of using a design pattern is that it makes programs **harder** to understand.
- Formal specification of behavioral contracts is **better** than informal textual specification.
- Formal specification of behavioral contracts is **worse** than informal textual specification.
- It is a **bad** design practice to provide one method that both mutates and returns an object’s state, rather than providing separate accessor and mutator methods.
- It is **good** design practice to use a unified accessor/mutator

Outline

- “Mixed messages” post-mortem
- “Are you my type” post-mortem
- Permutation generator post-mortem
- Cryptarithm post-mortem

We saw a lot of code like this on the exam

```
public enum Group {
    O("O"), A("A"), B("B"), AB("AB");

    private final String name;

    Group(String name) {
        this.name = name;
    }

    @Override
    public String toString() {
        return name;
    }
}
```

`toString` logic is unnecessary; this is entirely equivalent

```
public enum Group { O, A, B, AB }
```

Java generates high-quality `Object` methods for *every* enum

- Even simple ones
  - `enum Stooge { Larry, Moe, Curly }`
- You get for free: `equals`, `hashCode`, `toString`, `compareTo`
- The only one you're allowed to override is `toString`
- But don't unless you have a good reason!

Many solutions were behaviorally 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

To those of you who turned in repetitious solutions

A workmanlike solution – fields and constructor

```
public final class BloodType {
    public enum Group { O, A, B, AB }
    public enum RhFactor { NEGATIVE, POSITIVE }

    private final Group group;
    private final RhFactor rhFactor;

    public BloodType(Group group, RhFactor rhFactor) {
        if (group == null)
            throw new IllegalArgumentException("Group null”);
        if (rhFactor == null)
            throw new IllegalArgumentException("Rh factor null");
        this.group = group;
        this.rhFactor = rhFactor;
    }
```

A workmanlike solution – `Object` methods

```
@Override public boolean equals(Object o) {
    if (!(o instanceof  BloodType))
        return false;
    BloodType bt = (BloodType) o;
    return bt.group == group && bt.rhFactor == rhFactor;
}

@Override public int hashCode() {
    return 31 * group.hashCode() + rhFactor.hashCode();
}

@Override public String toString() {
    return group.toString() + (rhFactor == NEGATIVE ? "-" : "+");
}
```

A workmanlike solution – compatibility methods

```
public boolean canReceiveFrom(BloodType donor) {
    boolean groupOk = group == Group.AB || donor.group == Group.O
                      || donor.group == group;
    boolean rhOk = rhFactor == RhFactor.POSITIVE
        || donor.rhFactor == RhFactor.NEGATIVE;
    return groupOk && rhOk;
}

//Extra credit!
public boolean canDonateTo(BloodType recipient) {
    return recipient.canReceiveFrom(this);
}
```

Can we do better?

- Yes – there are only eight distinct values
  - but potentially millions of instances ☹
- Solution
  - Replace public constructor with pubic static factory & private constructor
  - Keep table of instances
- Resulting class is said to be *instance-controlled*

Code to achieve instance control

```
public static BloodType instanceOf(Group group, RhFactor rhFactor) {
    if (rhFactor == null) throw new NullPointerException("RhFactor");
    return instanceMap.get(group).get(rhFactor);
}
private static final Map<Group, Map<RhFactor, BloodType>> instanceMap =
    new EnumMap<>(Group.class);
static {
    for (Group group : Group.values()) {
        Map<RhFactor, BloodType> rhMap = new EnumMap<>(RhFactor.class);
        for (RhFactor rhFactor : RhFactor.values())
            rhMap.put(rhFactor, new BloodType(group, rhFactor));
        instanceMap.put(group, rhMap);
    }
}
private BloodType(Group group, RhFactor rhFactor) {
    this.group = group;
    this.rhFactor = rhFactor;
}
```

Instance control assessment

- You no longer need to override `equals` and `hashCode` !
  - All equal instances are identical, so Object implementation suffices
- Net increase of five lines of code
- Significant improvement in space and time performance
- Questionable on an exam, but worthwhile in real life

Can we do still better?

- Perhaps – compatibility function seems clunky
- Blood group and Rh factor have similar structure
  - Presence or absence of certain antigens
- Let's exploit that structure and see what happens

Code to exploit common structure of group and Rh Factor

```
private enum Antigen { A, B, Rh }
private final EnumSet<Antigen> antigens;
public enum Group {
    O(EnumSet.noneOf(Antigen.class)), A(EnumSet.of(Antigen.A)),
    B(EnumSet.of(Antigen.B)), AB(EnumSet.of(Antigen.A, Antigen.B));
    private final EnumSet<Antigen> antigens;
    Group(EnumSet<Antigen> antigens) { this.antigens = antigens; }
}
public enum RhFactor {
    NEGATIVE(EnumSet.noneOf(Antigen.class)),
    POSITIVE(EnumSet.of(Antigen.Rh));
    private final EnumSet<Antigen> antigens;
    RhFactor(EnumSet<Antigen> antigens) { this.antigens = antigens; }
}
public BloodType(Group group, RhFactor rhFactor) {
   antigens = EnumSet.copyOf(group.antigens);
   antigens.addAll(rhFactor.antigens);
}
```

Accessors for this implementation

```
public Group group() {
    return antigens.contains(Antigen.A)
        ? antigens.contains(Antigen.B) ? Group.AB : Group. A
        : antigens.contains(Antigen.B) ? Group.B  :  Group.O;
}

public  RhFactor rhFactor() {
    return antigens.contains(Antigen.Rh) ?
        RhFactor.POSITIVE : RhFactor.NEGATIVE;
}
```

Compatibility functions are gorgeous!

```
public boolean canReceiveFrom(BloodType bt) {
    return antigens.containsAll(bt.antigens);
}

public boolean canDonateTo(BloodType recipient) {
    return bt.antigens.containsAll(antigens);
}
```

And they run like a bat out of hell

- An `EnumSet` is actually a long used as a bit vector
- Actual implementation: `return (es.elements & ~elements) == 0;`

Antigen implementation assessment

- A bit longer and conceptually difficult
  - Probably not appropriate for an exam
- Code is illuminating and elegant
- Performance is (probably) a wash
- Not clear whether it's better on balance
  - But a design alternative worth considering

Outline

- “Mixed messages” post-mortem
- “Are you my type” post-mortem
- Permutation generator post-mortem
- Cryptarithm post-mortem

Design comparison for permutation generator

- Template Method pattern
  - Easy to code
  - Ugly to use
- Strategy 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

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 that 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

```
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<>();
    for (List<Integer> perm : Permutations.of(list)) {
        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 i = 0; i <= 10; i++)
        exhaustiveTest(i);
}
```

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

- “Mixed messages” post-mortem
- “Are you my type” post-mortem
- Permutation generator post-mortem
- Cryptarithm post-mortem

A fast, fully functional cryptarithm solver in 6 slides

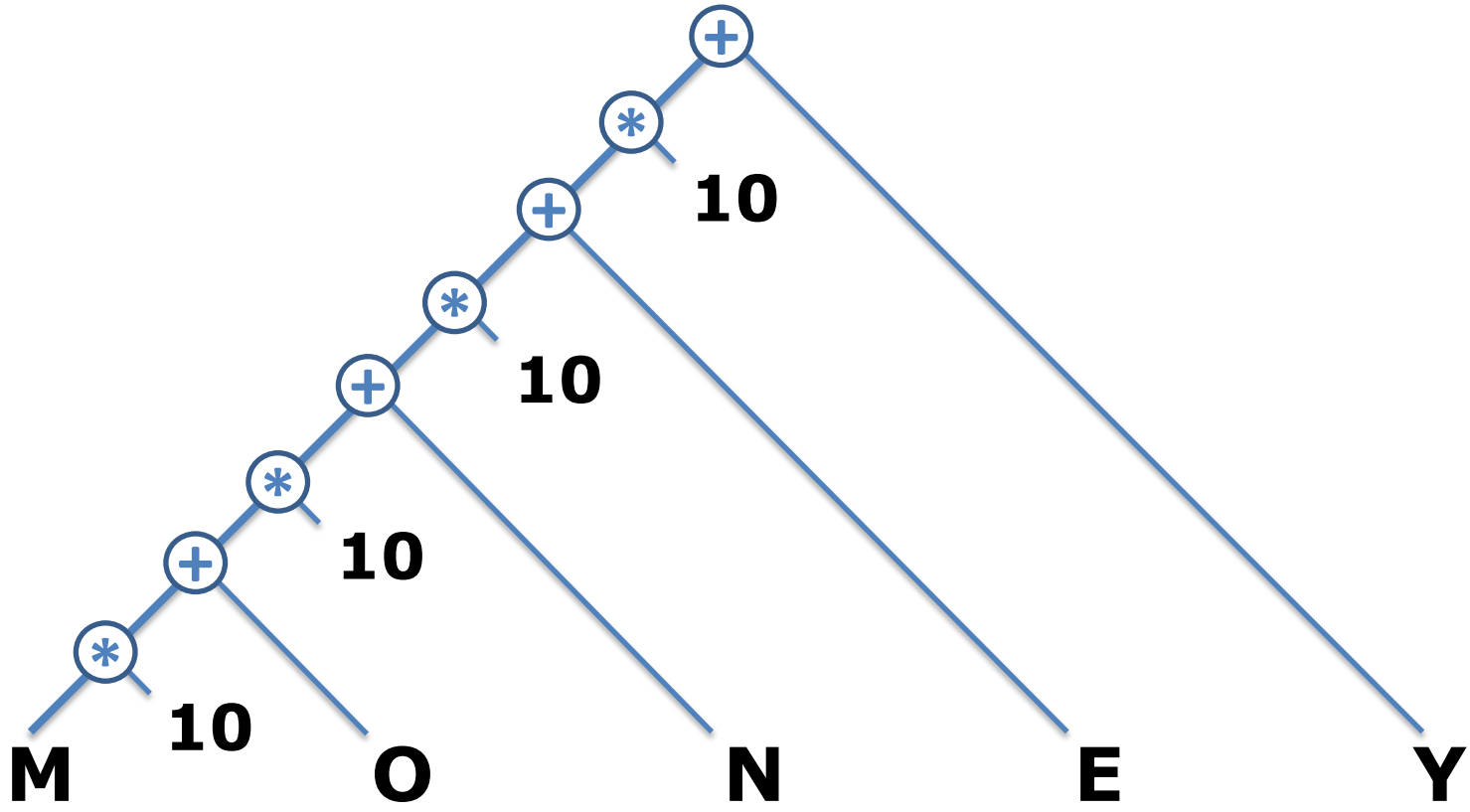
To refresh your memory, here's the grammar

```
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



$$(((M * 10 + O) * 10 + N) * 10 + E) * 10 + Y$$
$$= M * 10^4 + O * 10^3 + N * 10^2 + E * 10 + Y$$

Cryptarithm 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
- Don't do things you know to be wrong
  - such as writing constant hash functions
- Not enough to be merely correct; code must be clearly correct
  - Nearly correct is right out.