Principles of Software Construction

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

Midterm 1 and Homework 3 Post-Mortem

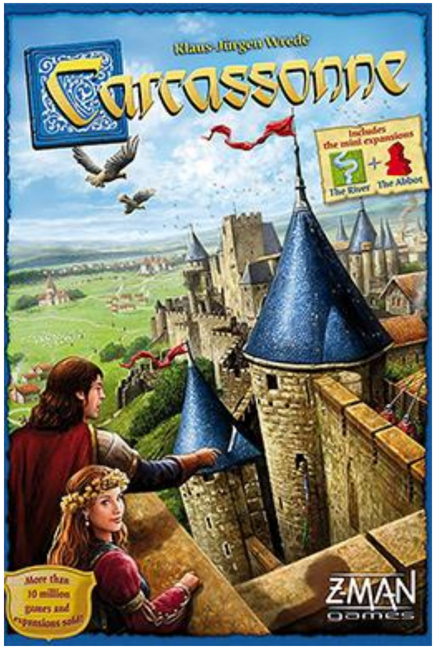
Josh Bloch Charlie Garrod

School of Computer Science

institute for SOFTWARE RESEARCH

Administrivia

- Homework 4a due Thursday, 11:59 p.m.
  - Design review meeting is mandatory

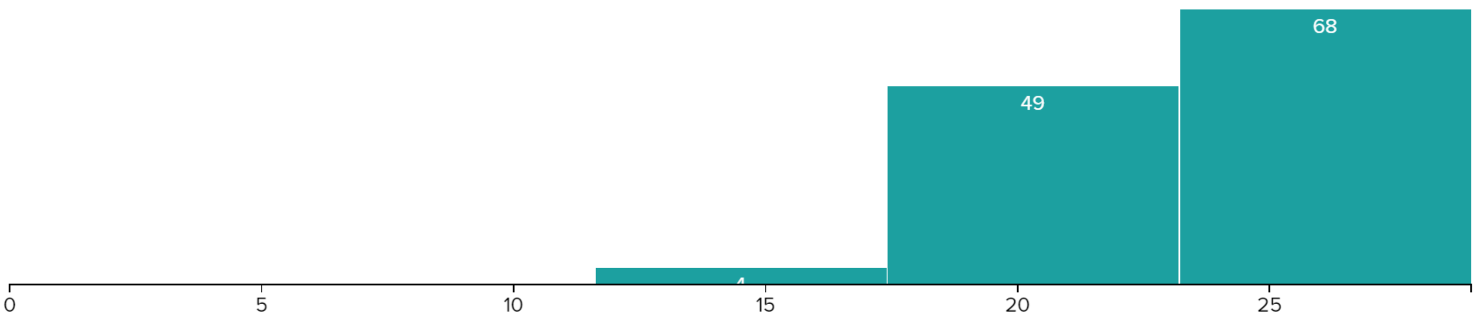


Outline

I. Midterm exam post-mortem – SET problem
II. Permutation generator post-mortem
III. Cryptarithm post-mortem

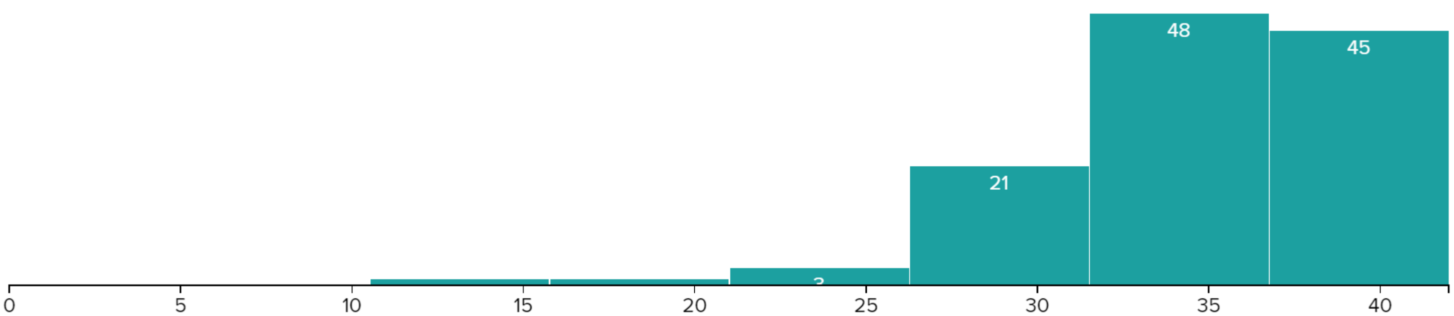
Midterm exam results – histograms of raw scores

Review Grades for **Midterm exam 1** (Excluding SET) ● **GRADES PUBLISHED**



MINIMUM	MEDIAN	MAXIMUM	MEAN	STD DEV
14.0	**24.0**	**29.0**	**23.99**	**3.36**

Review Grades for **Midterm exam 1 -- SET** ● **GRADES PUBLISHED**



MINIMUM	MEDIAN	MAXIMUM	MEAN	STD DEV
14.0	**36.0**	**42.0**	**34.65**	**4.65**

Anyone know a simpler expression for this?

```
if (myDog.hasFleas()) {
    return true;
} else {
    return false;
}
```

Hint: it's not this

```
return myDog.hasFleas() ? true : false;
```

Please do it this way from now on

We reserve the right to deduct points if you don't

```
return myDog.hasFleas();
```

SET problem – `Card` should be immutable

- *Much* safer – value can't change underneath you
- Trivial to use concurrently – no synchronization necessary
- More efficient – can share instances
- **Always make simple value classes immutable!**

What's the best type for a feature?

What's the best type for a SET?

A good, basic solution – features (1/4)

A good, basic solution – features (1a/4)

institute for SOFTWARE RESEARCH

A good, basic solution – fields, constructor, accessors (2/4)

A good, basic solution – Card methods (3/4)

A good, basic solution – `Object` methods (4/4)

API is simple – client code is pretty

Why is this solution ⅓ the length of many we received?

Why is this solution ⅓ the length of many we received?

- **Good choice of representation**
  - Fighting with representation adds verbosity
- **Makes good use of the facilities provided for us by the platform**
  - Object methods on `enum`
  - Utility methods such as `Objects.requireNonNull` and `List.of`
- **Makes good use of itself**
  - Code reuse vs. copy-and-paste

Using **generics** to make a reusable `thirdInSet` method

Generic methods are powerful, but declarations are ugly and complex

```
private static <T extends Enum<T>> T thirdInSet(T first, T second) {
    … // Implementation redacted
}
```

You can call static method directly from `Card.thirdInSet` or dispatch to it from feature enums:

```
public enum Number { ONE, TWO, THREE;
    Number thirdInSet(Number second) {
        return Card.thirdInSet(this, second);
    }
}

public enum Color { RED, GREEN, PURPLE;
    Color thirdInSet(Color second) {
        return Card.thirdInSet(this, second);
    }
}
```

Deep magic: how to "inherit" `thirdInSet` implementation

Default implementations, added in Java 8, are the secret ingredient

```
interface Feature<T extends Enum<T> & Feature<T>> { // 😫😫😫😫😫😫
    default T thirdInSet(T second) {
        … // Implementation redacted
    }
}

public enum Number implements Feature<Number> { ONE, TWO, THREE }
public enum Color implements Feature<Color> { RED, GREEN, PURPLE }
public enum Shading implements Feature<Shading> {OUTLINE, STRIPED, SOLID}
public enum Shape implements Feature<Shape> { DIAMOND, SQUIGGLE, OVAL }
```

A worthwhile performance tweak to Card

You didn't need to do this on the exam, but it's worth it in real life

Replace this

```
public Card(Number num, Color color, Shading shade, Shape shape);
```

with this

```
private Card(Number num, Color color, Shading shade, Shape shape);

private static final List<Card> deck = newDeck();

public static Card of(Number number, Color color, Shading shading,
        Shape shape) {
    return deck.get(index(number, color, shading, shape));
}

private int index(Number num, Color col, Shading shd, Shape shp) {
    return 27*num.ordinal() + 9*col.ordinal() + 3*shd.ordinal() + shp.ordinal();
}
```

- This is called an *instance-controlled class*
  - Only 81 instances ever exist (one per value)

Outline

I. Midterm exam post-mortem – SET problem
II. Permutation generator post-mortem
III. Cryptarithm post-mortem

Design comparison for permutation generator

- Command pattern
  - Easy to code
  - Reasonably pretty to use:

```
PermGen.doForAllPermutations(list, (perm) -> { // lambda
    if (isSatisfactory(perm))
        doSomethingWith(perm);
});
```

- Iterator pattern
  - Tricky to code because algorithm is recursive and Java lacks *generators*
  - ***Really* pretty** to use because it works with `for`-each loop

```
for (List<Foo> perm : Permutations.of(list))
    if (isSatisfactory(perm))
        doSomethingWith(perm);
```

- Performance 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 (consecutive integers 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!

Pros and cons of exhaustive testing

- Pros and cons of exhaustive testing
  - \+ Gives you (nearly) 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

- Midterm exam post-mortem
- Permutation generator post-mortem
- Cryptarithm post-mortem
  - Cryptarithm class (6 slides)
  - CryptarithmWordExpression (2 slides)
  - Main program (1 slide)

`Cryptarithm` class (1/6) – fields

`Cryptarithm` class (2/6) – constructor / parser

Sample input argument: `["send", "+", "more", "=", "money"]`

`Cryptarithm` class (3/6) – word parser

`Cryptarithm` class (4/6) – operator parser

`Cryptarithm` class (5/6) – solver

`Cryptarithm` class (6/6) – solver helper functions

`CryptarithmExpressionContext` class

Naïve version; solves 10-digit cryptarithms in about 1 s.

`CryptarithmWordExpression` class

Naïve version; solves 10-digit cryptarithms in about 1 s.

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, step back and rethink it
  - "A week of coding can often save a whole hour of thought."
- Not enough to be merely correct; code must be clearly correct
  - Try to avoid nearly correct.