Principles of Software Construction

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

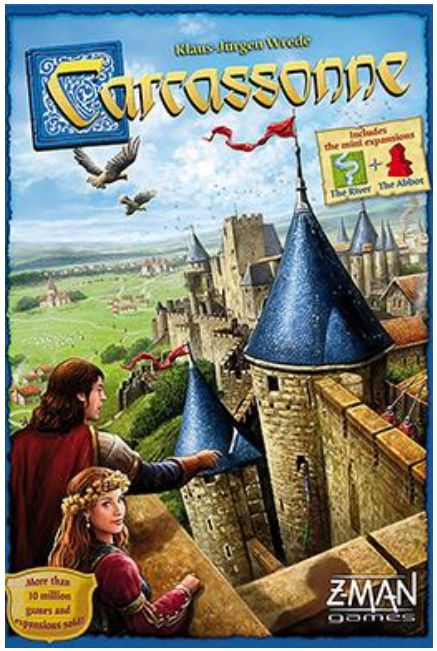
Midterm 1 and Homework 3 Post-Mortem

Josh Bloch Charlie Garrod

School of Computer Science
institute for SOFTWARE RESEARCH

Administrivia

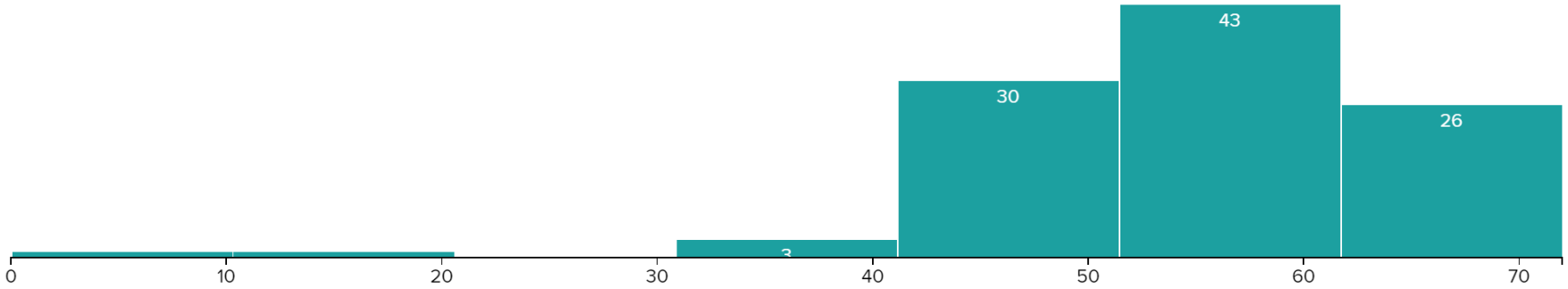
- Homework 4a due Thursday, 11:59 p.m.
  - Design review meeting is mandatory



Outline

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

Midterm exam results



MINIMUM	MEDIAN	MAXIMUM	MEAN	STD DEV
1.0	54.75	72.0	54.86	10.32

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();
```

`DnaStrand` 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 **representation** for a base?

What's the best internal **representation** for a strand?

In a real-world setting, performance concerns might intrude

- The human genome has about 3 billion base pairs
  - Would take up 24 GB with our current representation
  - But each base pair has only 2 bits of actual information
  - So you could cut this down by a factor of 16
- This implies a bit-vector representation
  - Strand would be represented as an array of (say) `int`
  - Where each int represents 16 bases
- But you don't do this sort of thing until you know you have to
  - **Avoid premature optimization**
- It would have been wrong to do this for the exam

What are best input types for constructor (or factory)?

A good, basic solution – Base enum (1/4)

A good, basic solution – field and constructor (2/4)

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

A good, basic solution – complementarity methods (4/4)

API is good – client code is pretty

Why is this solution ¼ the length of many we received?

Why is this solution ¼ the length of many we received?

- **Good choice of internal representation**
  - Fighting with representation adds verbosity
- **Makes good use of the facilities provided for us by the platform**
- **Makes good use of itself**
  - Code reuse vs. copy-and-paste

Outline

I. Midterm exam post-mortem
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) -> {
    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 “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

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

CryptarithmWordExpression

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.