CSC207.01 2014S, Class 34: Implementing Queues with Arrays

Overview

- Preliminaries.
  - Upcoming work.
  - Admin.
  - Questions.
- Wrappers, Adapters, Delegation, and such.
- Design Patterns.
- Lab.

Preliminaries

Upcoming work

- Exam 2 due at the start of class on Friday.
- Today's writeup: Exercise 5. CANCELLED
- Reading for Wednesday: Priority Queues
- Next homework distributed Wednesday.

Admin

- Happy April fools day!
- Review session tonight at 7. Earnest will discuss the wonders of DNF, kth-smallest on arrays, and more.
- Consider running for the SEPC.
  - Send email to sepc-cs@grinnell.edu (look, an email address for spambots!)
  - Give a statement that they can distribute so that people will vote for you.
  - Successful past statements: "I will bake cookies." "I will lobby for a couch."
- Get ready to vote on T-Shirts (the Curried Lamb-Duh T-shirt has disappeared; you should encourage the SEPC to add it).
- Extra credit:
  - Convocation, Wednesday, noon: Elizabeth Kolbert (the Sixth Extinction)
  - Technology in the liberal arts symposium on Thursday.
  - CS table Friday: TBD

Questions on the Exam

Can our skip list iterator call underlying methods of the skip list?

Certainly.

Should we fail fast for the skip list iterator?

It would be nice. It's also fine if you just crash and burn if someone else mutates the list.

Should we fail fast for the filtered list iterator?

It would be nice. It's also fine if you just crash and burn if someone else mutates the list.

Why does my filtered iterator crash and burn when I try to remove an element from a filtered range using delegation?

Because ranges do not implement remove.

That's acceptable behavior. You can't do any better than the underlying iterator.

Wrappers, Adapters, Delegation, and such

- ReportingLinearStructure.java
- Useful for exploring the behavior of linear structures.
- But also reveals some approaches to object-oriented design.
- Reminder: One goal of object-oriented design is that you can "easily" build new objects that have similar behavior in most cases, but add behavior.
  - Inheritance is one way of achieving that.
  - You override the method whose behavior you want to change.
  - If you want to get the behavior of the original, but add behavior

Code

```
public class Utils
{
   public int square(int x)
   {
     return x*x;
   } // square

   public double expt(double x, int n)
   {
     ...;
   } // expt
} // class Utils

public class BetterUtils
  extends Utils
{
```

```
int callsToSquare = 0;

@Override
public int square(int x)
{
  ++this.callsToSquare;
  // And do the old behavior
  return super.square(x);
} // square(int)
```

Problem! What if we want to add/extend anything that meets an interface?

A typical solution appears in the ReportingLinearStructure.java class.

- Match the interface.
- Add a field for the underlying class/object we extend
- Basic implementation of every method
  - method(params) -> base.method(params)
- And then add utility that you want

What is the additional utility that we might want?

- Printing/reporting/logging
- Count number of function calls
- Sometimes just change behavior, e.g., each call to put should put two copies

A more general approach to one of the benefits of inheritance

Design Patterns

- Common problems (e.g., "I want to log every time I do a function call")
- Common ways of solving the problems (see above)
- A nice software engineering effort: List general kinds of design problems that people encounter, give the typical form of a solution, and NAME it.
- At Grinnell, we tend to focus on knowing/using the terms, more than the "look up how to solve the problem."
- What we've done doesn't have a clear pattern name
  - Most people will understand what you're doing if you say "object wrapper"
  - Related pattern: Delegate
  - Related pattern: Adapter

Lab