A Post-Mortem of A2

Prof. Chris Jermaine
cmj4@cs.rice.edu

Prof. Scott Rixner
rixner@cs.rice.edu

Now That We're About To Hand A2 Back…

- Let's look at the space of possible designs
- Perhaps we can put all designs on a spectrum

Nothing
in abstract

Everything
in abstract

Now That We're About To Hand A2 Back…

- Let's look at the space of possible designs
- Perhaps we can put all designs on a spectrum

- Where are appropriate A2 designs?
  - — Probably in here

What Was Almost Mandatory in the Abstract?

```
toString ()
getRoundedItem ()
```

- Why?
  - — Imp seems to not depend in any way on underlying data representation
  - — Can easily imp in terms of other ops in the interface
  - — “toString” ex:

```
try {
  String returnVal = new String ("<");
  for (int i = 0; i < getLength (); i++) {
    Double curItem = getItem (i);
    if (i != 0)
      returnVal = returnVal + ", ";
    returnVal = returnVal + curItem.toString ();
  }
  returnVal = returnVal + ">"; ...
```

Also Should Have Put getLength There

- Since if you don’t you end up maintaining actual length (at least implicitly) in both
- So put it in the abstract
- Set it via a call to “super”

Was This the Best Design?

- Probably not, though no points were taken for this
- What else should have gone in the abstract class?

Was This the Best Design?

- Probably not, though no points were taken for this
- What else should have gone in the abstract class?
  - — The logic to deal with backing values and dividing everyone by a value
- Why?
  - — In the concrete, you end up repeating the same (bug prone!) logic everywhere

What Else To Put in the Abstract

- The backing value/delta, a multiplier, and logic to deal with it
- Would make sense to have the following in abstract:

```
private double delta;
private double mult;
private int len;

// these will be called by the concrete to map/unmap vals

// takes a val from outside world, converts into internal
protected double mapValue (double mapMe) {
  return (mapMe - delta) * mult;}

// takes an internal val, converts into outside world val
protected double unMapValue (double unMapMe) {
  return (mapMe / mult) + delta;}
```

All of the Concrete Ops Now Call Map Funcs

```
public double getItem (int i) throws ... {
  // code here to extract the value at pos i
  ...
  // then un-map it
  return unMapValue (value);
}

public double setItem (int i, double setToMe) throws ... {
  setToMe = mapValue (setToMe);
  // code here to set the value at pos i
  ...
}
```

And addToAll Goes Into Abstract

```
private double backingValue;
private double mult;
private int len;

public void addToAll (void addMe) {
  delta += addMe;
}
```

And addToAll Goes Into Abstract

```
private double backingValue;
private double mult;
private int len;

public void addToAll (void addMe) {
  delta += addMe;
}
```

- Plus, you have a “multAllBy” in abstract so you can implement normalize in the concrete

```
protected void multAllBy (double multiplier) {
  mult /= multiplier;
  delta *= multiplier;
}
```

- An then constructor becomes:

```
protected ADoubleVector (double initVal, int vecLen) {}
```

That Would Have Been a Great Design

- But probably OK to go even further!
- Say you decided only public methods in concrete are “addMyself-ToHim”, “getItem”, and “setItem”
- How to do this? Many ways...
- One is to have a protected abstract “splitResult” routine:

```
protected abstract SplitResult splitSum (double divLine);
```

— This avgs/counts the stored values, partitioning above and below “divLine”

- “SplitResult” has:

```
public double getAvgLo ();
public double getAvgHi ();
public int getCountLo ();
public int getCountHi ();
```

Then l1Norm Is In Abstract

```
public double l1Norm () {
  SplitResult myRes = splitSum (delta * mult);
  return unMapValue (myRes.getAvgHi ()) * myRes.getCountHi () -
         unMapValue (myRes.getAvgLo ()) * myRes.getCountLo () +
         Math.abs ((len - myRes.getCountHi () - myRes.getCountLo ()) *
                   unMapValue (0.0));
}
```

Then l1Norm Is In Abstract

```
public double l1Norm () {
  SplitResult myRes = splitSum (delta * mult);
  return unMapValue (myRes.getAvgHi ()) * myRes.getCountHi () -
         unMapValue (myRes.getAvgLo ()) * myRes.getCountLo () +
         Math.abs ((len - myRes.getCountHi () - myRes.getCountLo ()) *
                   unMapValue (0.0));
}
```

- And so is normalize:

```
public double normalize () {
  SplitResult myRes = splitSum (Double.POSITIVE_INFINITY);
  double tot = unMapValue (myRes.getAvgLo ()) * myRes.getCountLo () +
               Math.abs ((len - myRes.getCountHi () - myRes.getCountLo ()) *
                         unMapValue (0.0));
  mult *= tot;
  delta /= tot;
}
```

- And then “multAllBy” goes away

When Have You Gone Too Far?

- When you find yourself designing methods in the abstract that somehow take into account imps in the concrete
- Obvious example:
  - — You start checking the subclass type to see what you're gonna do
- But it can be more subtle
  - — For example, were my "l1Norm", "normalize" appropriate?
  - — Implementation did leak up a bit, since aware that not all vals will be explicit
  - — Was this a bad design?

Questions?