Inheritance and Type Hierarchies

Prof. Chris Jermaine
cmj4@cs.rice.edu

Prof. Scott Rixner
rixner@cs.rice.edu

We've Argued OO Is About Abstraction

- Type hierarchies: methodology to provide for abstraction in OOP
- Based on ability to declare an "is-a" relationship between types
- Java uses keyword "extends"
  - — But I like to still think about it using "is-a"
- When we say:

```
class DeleteOp extends class IEditOp {...}
```

- We are saying a deletion "is-an" edit operation

So What Does Extends (Is-A) Mean?

- If:

```
class DeleteOp extends class IEditOp {...}
```

  — "DeleteOp" automatically has all of the methods that "IEditOp" has

  — "DeleteOp" automatically has all of the member variables that "IEditOp" has

  — In fact, if you have no more code than this, a "DeleteOp" is exactly the same as an "IEditOp"

- If this is all type hierarchies gave you, would be silly

  — Why have "DeleteOp" if it's exactly the same as "IEditOp"?

- Luckily, this is not what extends/is-a means

  — A "subclass" such as DeleteOp is **constrained** by its superclass

  — But it is **not** exactly the same as its superclass

Over-Riding Superclass Behavior

- Fundamental to type hierarchies in OOP...
  - — Is the ability of a subclass to define its own behavior
  - — But this can only be done in a way that's consistent with the superclass
  - — In Java, this is done by re-defining or **over-riding** superclass methods
- Why have this whole sublcass/superclass thing in OOP?

Over-Riding Superclass Behavior

- Fundamental to type hierarchies in OOP...
  - — Is the ability of a subclass to define its own behavior
  - — But this can only be done in a way that's consistent with the superclass
  - — In Java, this is done by re-defining or **over-riding** superclass methods
- Why have this whole subclass/superclass thing in OOP?
  - — Makes your code more compact, easier to maintain
  - — Because many types can share the same method implementation
  - — Change code in the superclass, all subclasses automatically get update

Over-Riding Superclass Behavior

- Fundamental to type hierarchies in OOP...
  - — Is the ability of a subclass to define its own behavior
  - — But this can only be done in a way that's consistent with the superclass
  - — In Java, this is done by re-defining or **over-riding** superclass methods
- Why have this whole subclass/superclass thing in OOP?
  - — Makes your code more compact, easier to maintain
  - — Because many types can share the same method implementation
  - — Change code in the superclass, all subclasses automatically get update
  - — But the real power of inheritance comes when you add on **polymorphism**
  - — Polymorphism: ability
  - — This is best illustrated with an example...

Edit Distance Revisited

- Say that we want to actually print out the set of edits
  - — Not just compute the edit distance between strings
- Look on the web, you'll see complex pictures of how to do this
- Generally involves computing edit distance DP matrix first
  - — Then backtracking through matrix to see how you got to the end
- Consider the following (quite simple, extensible) code based on inheritance plus polymorphism

Edit Distance Reloaded

- Assume we define an “IEditOp” class
- Four subclasses: “InsertOp”, “DeleteOp”, “XFormOp”, “NoOp”
  - — Correspond to the three edit operations, plus doing nothing

Edit Distance Reloaded

- Goal is to transform string “a” into string “b”
  - — Assume we start by shifting chars over to make a[0], b[0] safe
  - — Then declare a matrix of edit ops rather than scores
  - — And get ourselves an edit op “factory”... what is this?

```
// pre-prend both strings with “ “
a = “ “ + a;
b = “ “ + b;

// allocate the edit op matrix
IEditOp [][]A = new IEditOp [a.length ()][b.length ()];

// allocate the edit op factory
IEditOpFactory myFactory = new IEditOpFactory ();
```

Edit Distance Reloaded

- The loop to compute the edit distance becomes quite simple

```
for (int i = 0; i < a.length (); i++) {   // loop through matrix
  for (int j = 0; j < b.length (); j++) { // loop through matrix
    A[i][j] = null;                       // initialize entry
    ArrayList <IEditOp> IEditOpList =     // get list of edit ops
      myFactory.run (A, i, j, a.charAt (i), b.charAt(j));
    for (IEditOp current : IEditOpList) {
      if (A[i][j] == null ||              // see if this edit op is best
          current.getCostToHere () < A[i][j].getCostToHere ())
        A[i][j] = current;                // if this op is best, use it
    }
  }
}
A[a.length () - 1][b.length () - 1].printEdits ();
```

— Let's examine that inner-most loop in more detail

Edit Distance Reloaded

```
ArrayList <IEditOp> IEditOpList =
  myFactory.run (A, i, j, a.charAt (i), b.charAt(j));

for (IEditOp current : IEditOpList) {
  if (A[i][j] == null ||
     current.getCostToHere () < A[i][j].getCostToHere ())
    A[i][j] = current;
}
```

— This “factory” produces a list of all possible edit operations we can use here

— There are different edit operations, but each “is-an” IEditOp

— So they can go into the same list of generic IEditOp objects

— A[i][j] should store the last op needed to xform “i” chars in “a” to “j” chars in “b”

— This loop checks each edit op in turn to see if it is the best option

The IEditOpFactory

```
class IEditOpFactory {

  public ArrayList <IEditOp> run (IEditOp [][]A,
    int i, int j, char charFromA, char charFromB) {

    ArrayList <IEditOp> myRes = new ArrayList <IEditOp> ();
    myRes.add (new InsertOp (A, i, j, charFromA, charFromB));
    myRes.add (new DeleteOp (A, i, j, charFromA, charFromB));
    myRes.add (new XformOp (A, i, j, charFromA, charFromB));
    myRes.add (new NoOp (A, i, j, charFromA, charFromB));
    return myRes;
  }
}
```

- The “run” method puts all possible edit ops into a list
  - — Each edit operation is aware of its own cost, and how it links to other edit ops
  - — Constructor uses A, i, j, charFromA, and charFromB to compute this
  - — Factories of this sort are common in OOP
  - — Useful ‘cause it abstracts away the task of creating the various edit operations

Before Continuing with the Example

- Why write code like this?

Before Continuing with the Example

- Why write code like this?
  - — In complex programs, much more maintainable and extensible
- We have totally abstracted out the idea of an edit operation
  - — You can add, change an edit op without touching core loop
  - — Just implement a new subclass of IEditOp, then add it to the factory

How Is the IEditOp Class Coded?

```
abstract class IEditOp {

  // this is the previous edit operation in the
  // optimal transform of a into b
  private IEditOp parent = null;

  // this is the total cost to get to this particular
  // edit operation
  private int costToHere = 999999999;

  // print yourself nicely to the screen
  public abstract void printSelf ();
  ...
```

— First, this class is "**abstract**"

— Means you can't ever "new" it. Why?

— Analogy you've never got a generic "color" (you've got black, white, etc.)

How Is the IEditOp Class Coded?

```
abstract class IEditOp {

  // this is the previous edit operation in the
  // optimal transform of a into b
  private IEditOp parent = null;

  // this is the total cost to get to this particular
  // edit operation
  private int costToHere = 999999999;

  // print yourself nicely to the screen
  public abstract void printSelf ();
  ...
```

— Note this class has some abstract methods

— Means we force the subclass to implement them

— Why declare as abstract? Can't print a generic edit operation!

Also Has Three Concrete Methods

```
  // set up the internal stuff--parent and cost
  protected void setup (IEditOp myParent, int cost) {
    parent = myParent;
    costToHere = cost;
  }

  // get the total cost
  public int getCostToHere () {
    return costToHere;
  }

  // print all of the edits needed to get from string a to string b
  public void printEdits () {
    if (parent != null) {
      parent.printEdits ();  // Noooo!  Recursion!!!
    }
    printSelf ();
  }

}
```

Let's Consider printEdits In Detail

```
// print all of the edits needed to get from string a to string b
public void printEdits () {
  if (parent != null) {
    parent.printEdits ();  // Noooo!  Recursion!!!
  }
  printSelf ();
}
```

- Note: uses recursion to print the sequence of edits
- Also note: "printSelf" is abstract... how does this work?
  - — Every IEditOp object is an instance of a subclass of IEditOp (IEditOp is abstract)
  - — When call printSelf, Java invokes printSelf method associated with concrete class
  - — Java is smart enough to figure out which concrete class this object belongs to
  - — This is known is **polymorphism** in OOP...
  - — ...that is, the same call can have different results based on identity of "this"

How to Implement an Actual Edit Op?

```
class InsertOp extends IEditOp {

  // this is the char we are inserting
  private char insertedChar;

  // this is where we do the insert
  private int atWhichPos;

  public InsertOp (IEditOp [][]A, int i, int j,
    char charFromA, char charFromB) { ...}

  public void printSelf () {
    System.out.format ("Inserted %c at pos %d.\n",
        insertedChar, atWhichPos);
  }
}
```

— Note that we can declare member vars not present in parent class

— Also note that we **must** provide an implementation of the printSelf method

What Does the Constructor Look Like?

```
public InsertOp (IEditOp [][]A, int i,
  int j, char charFromA, char charFromB) {

  // make sure string b is not empty (if it is,
  // can't insert to get to it)
  if (j != 0) {
    setup (A[i][j - 1], A[i][j - 1].getCostToHere () + 1);
    insertedChar = charFromB;
    atWhichPos = i;
  }
}
```

- As long as we are not trying to xform i characters of string “a” to 0 chars of “b”...
- ...we can put a non-infinite cost as well as a parent into the InsertOp
- In insertion, parent is at A[i][j - 1] (match first j - 1 chars of “b”, do an insert)
- We also record the insertion associated with this operation
- And the position at which the insertion takes place

How About the XformOp Class?

```
class XformOp extends IEditOp {

  private int charFrom;
  private int charTo;
  private int atWhichPos;

  public XformOp(IEditOp [][]A, int i, int j, char charFromA, char charFromB){

    // if either string has no characters, can't do a xform
    if (i != 0 && j != 0) {
      setup (A[i - 1][j - 1], A[i - 1][j - 1].getCostToHere () + 1);
      charFrom = charFromA;
      charTo = charFromB;
      atWhichPos = i;
    }
  }

  public void printSelf () {
    System.out.format ("Transformed %c at pos %d to %c.\n",
         charFrom, atWhichPos, charTo);
  }
}
```

So What Does All of This Do?

```
    String a = " " + "asxaaxdsfaayyahhhhzzzjj";
    String b = " " + "asaadsfaaaahhhzzzjjj";
```

```
output is:
Deleted x from pos 3.
Deleted x from pos 6.
Transformed y at pos 12 to a.
Deleted y from pos 13.
Deleted h from pos 18.
Inserted j at pos 23.
```

Some Take-Home Points

- Abstract classes... why to use ‘em?
  - — You’ll almost always need an abstract class sitting on top of the hierarchy
  - — Why?
  - — So, get in the habit of putting them there (as a placeholder) even when you think you don’t, whenever you have a hierarchy

Some Take-Home Points

- Abstract classes... why to use ‘em?
  - — You’ll almost always need an abstract class sitting on top of the hierarchy
  - — Why?
  - — So, get in the habit of putting them there (as a placeholder) even when you think you don’t, whenever you have a hierarchy
- How is inheritance different from polymorphism?
  - — Inheritance is a mechanism by which classes can share code
  - — Polymorphism is a mechanism by which a single method call in a piece of code could have different results, depending upon the actual type of the object
  - — Closely related, but not the same!

Some Take-Home Points

- Is all of this worth it? Factories, abstract classes, polymorphism...
  - — If you want to get an edit distance computation working quickly, honest answer:
  - — Probably not
  - — But: for a problem of even intermediate complexity...
  - — ...carefully thinking about how to carve up a problem in this way is invaluable
  - — Greatly simplifies debugging, maintenance, and even coding
  - — We have used inheritance/polymorphism to create a number of simple, small parts that can be understood/implemented/tested/maintained independently
  - — For more complicated problems, invaluable

Some Take-Home Points

- Is all of this worth it? Factories, abstract classes, polymorphism...
  - — If you want to get an edit distance computation working quickly, honest answer:
  - — Probably not
  - — But ...y...
  - — ...ca ...em in this way is invaluable
  - — Gre ...ven coding
  - — We ...a number of simple, small parts that ...ined independently
  - — For more complicated problems, invaluable

It's all about

ABSTRACTION!!

Why Does This Design Give Us Flexibility?

- Super-easy to add new edit ops (or take existing ones out)
- Say we want the ability to insert repeating values for free
- This defines a new edit op
  - — Distance between “abcd” and “aaabcdd” is now zero
  - — Repeat the first “a” twice, then repeat the final “d”
- To code this up, two minor changes

Change #1: Define the RepeatOp Class

```
class RepeatOp extends IEditOp {

  // this is the char we are repeating
  private char repeatedChar;

  // this is where we do the repeat
  private int atWhichPos;

  public RepeatOp (IEditOp [][]A, int i, int j,
    char charFromA, char charFromB) { ...}

  public void printSelf () {
    System.out.format ("Repeated %c at pos %d.\n",
        insertedChar, atWhichPos);
  }
}
```

Change #1: Define the RepeatOp Class

```
public RepeatOp (IEditOp [][]A, int i,
  int j, char charFromA, char charFromB) {

  // make sure string b is not empty (if it is,
  // can't repeat to get to it)
  if (j != 0 && charFromA == charFromB) {
    setup (A[i][j - 1], A[i][j - 1].getCostToHere ());
    repeatedChar = charFromB;
    atWhichPos = j;
  }
}
```

Change #2: Modify the Factory

```
class IEditOpFactory {

  public ArrayList <IEditOp> run (IEditOp [][]A,
    int i, int j, char charFromA, char charFromB) {

    ArrayList <IEditOp> myRes = new ArrayList <IEditOp> ();
    myRes.add (new InsertOp (A, i, j, charFromA, charFromB));
    myRes.add (new DeleteOp (A, i, j, charFromA, charFromB));
    myRes.add (new XformOp (A, i, j, charFromA, charFromB));
    myRes.add (new NoOp (A, i, j, charFromA, charFromB));
    myRes.add (new RepeatOp (A, i, j, charFromA, charFromB));
    return myRes;
  }
}
```

Change #2: Modify the Factory

```
class IEditOpFactory {

  public ArrayList <IEditOp> run (IEditOp [][]A,
    int i, int j, char charFromA, char charFromB) {

    ArrayList <IEditOp> myRes = new ArrayList <IEditOp> ();
    myRes.add (new InsertOp (A, i, j, charFromA, charFromB));
    myRes.add (new DeleteOp (A, i, j, charFromA, charFromB));
    myRes.add (new XformOp (A, i, j, charFromA, charFromB));
    myRes.add (new NoOp (A, i, j, charFromA, charFromB));
    myRes.add (new RepeatOp (A, i, j, charFromA, charFromB));
    return myRes;
  }
}
```

- That's it!!

Questions?