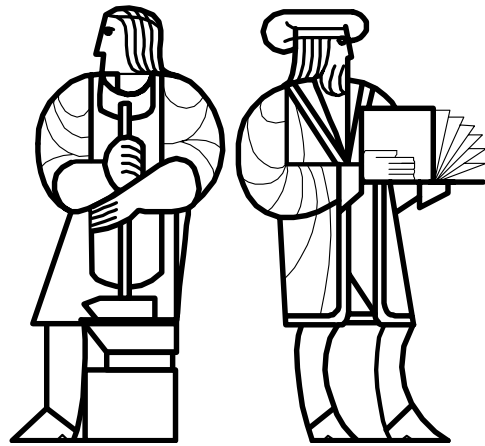

Subclassing



6.170

Michael Ernst

Saman Amarasinghe *

* your host this week

6.170 specifications

The “**precondition**”: constraints that hold before the method is called (if not, all bets are off)

requires – spells out any obligations on client

The “**postcondition**”: constraints that hold after the method is called (if the precondition held)

modifies – lists objects that may be affected by method; any object not listed is guaranteed to be untouched

throws – lists possible exceptions

effects – gives guarantees on the final state of modified objects

returns – describes return value

An example of a non-conforming client

```
static void Uniqueify(List<Integer> lst)
```

requires
modifies
effects
returns

lst is not null.
lst
all the elements of the lst are unique, repetitions are removed
none

```
static void Uniqueify(List<Integer> lst) {  
    lst.sort();  
    for(int i=0; i < lst.size()-1; i++)  
        while(lst.get(i) == lst.get(i+1)) {  
            lst.remove(i+1);  
            if(lst.size() <= i+1) break;  
        }  
}
```

Aha! The list is also sorted

Client:

```
Uniqueify(l1);  
i = BinarySearch(l1, j);
```

An example of a non-conforming client

```
static void Uniqueify(List<Integer> lst)
```

requires
modifies
effects
returns

lst is not null.
lst
all the elements of the lst are unique, repetitions are removed
none

```
static void Uniqueify(List<Integer> lst) {  
  
    for(int i=0; i < lst.size()-1; i++)  
        for(int j=lst.size()-1; j > i; j--)  
            if(lst.get(i) == lst.get(j))  
                lst.remove(j);  
}
```

Client:

```
Uniqueify(l1);  
i = BinarySearch(l1, j);
```

An example of unnecessarily strong spec

static void Uniqueify(List<Integer> lst)

requires
modifies
effects
returns

lst is not null.

lst

lst is sorted and all the elements of the lst are unique, repetitions are removed

none

```
static void Uniqueify(List<Integer> lst) {  
    lst.sort();  
    for(int i=0; i < lst.size()-1; i++)  
        while(lst.get(i) == lst.get(i+1)) {  
            lst.remove(i+1);  
            if(lst.size()<=i+1) break;  
        }  
}
```

Point example

```
class Point {
    private double x, y;
    public Point(double x, double y)
        { this.x=x; this.y=y; }
    public double x() { return x; }
    public double y() { return y; }
    public double r() { return Math.sqrt(x*x+y*y); }
    public double theta() { return Math.atan2(y,x); }
}
```

```
class SingleCalcPoint {
    private double x, y;
    private double r, theta;
    public SingleCalcPoint(double x, double y)
        { this.x=x; this.y=y;
          r = Math.sqrt(x*x+y*y);
          theta = Math.atan2(y,x);
        }
    public double x() { return x; }
    public double y() { return y; }
    public double r() { return r; }
    public double theta() { return theta; }
}
```

Changes are cumbersome

```
class Graphics {  
    void drawLine( Point p1,  
                  Point p2) { ... }  
    ...  
}  
  
...  
  
screen.drawLine( new Point(5,10) ,  
                  new Point(10,5) );
```

Changes are cumbersome

```
class Graphics {  
    void drawLine( SingleCalcPoint p1,  
                  SingleCalcPoint p2) { ... }  
    ...  
}  
  
...  
  
screen.drawLine(new SingleCalcPoint(5,10),  
                new SingleCalcPoint(10,5));
```

Point example

```
class Point {
    private double x, y;
    public Point(double x, double y)
        { this.x=x; this.y=y; }
    public double x() { return x; }
    public double y() { return y; }
    public double r() { return Math.sqrt(x*x+y*y); }
    public double theta() { return Math.atan2(y,x); }
}
```

```
class SingleCalcPoint extends Point {
    private double r, theta;
    public SingleClacPoint(double x, double y)
        { super(x, y);
          r = Math.sqrt(x*x+y*y);
          theta = Math.atan2(y,x);
        }
    public double r() { return r; }
    public double theta() { return theta; }
}
```

reducing impact of extension

```
class Graphics {  
    void drawLine(Point p1, Point p2) { ... }  
    ...  
}  
  
...  
  
screen.drawLine(new Point(5,10),  
                new Point(10,5));
```

reducing impact of extension

```
class Graphics {  
    void drawLine(Point p1, Point p2) { ... }  
    ...  
}  
  
...  
  
screen.drawLine(new SingleCalcPoint(5,10),  
                new SingleCalcPoint(10,5));
```

Only the constructors need to be changed!

Reuse of Code

```
class SingleCalcPoint {  
    private double x, y;  
    private double r, theta;  
    public SingleCalcPoint(double x, double y)  
    { this.x=x; this.y=y;  
      r = Math.sqrt(x*x+y*y);  
      theta = Math.atan2(y,x);  
    }  
    public double x() { return x; }  
    public double y() { return y; }  
    public double r() { return r; }  
    public double theta() { return theta; }  
}
```

Old members need to be redeclared, even if no change

Old methods need to be reimplemented, even if no change

```
class SingleCalcPoint extends Point {  
    private double r, theta;  
    public SingleCalcPoint(double x, double y)  
    { super(x, y);  
      r = Math.sqrt(x*x+y*y);  
      theta = Math.atan2(y,x);  
    }  
    public double r() { return r; }  
    public double theta() { return theta; }  
}
```

Only the additional members

Put the old constructor to good use

Only new and changed methods

Point example

```
class CachingPoint {
    private double x, y;
    public CachingPoint(double x, double y)
        { this.x=x; this.y=y; polarized = false; }
    public double x() { return x; }
    public double y() { return y; }

    private double r, theta;
    private boolean polarized;
    private void polarize() {
        if (!polarized) {
            r = Math.sqrt(x*x+y*y);
            theta = Math.atan2(y,x);
            polarized = true;
        }
    }
    public double r() { polarize(); return r; }
    public double theta() { polarize(); return theta; }
}
```

Point example

```
class CachingPoint extends Point {
    private double r, theta;
    private boolean polarized;
    public CachingPoint(double x, double y) {
        super(x,y);
        polarized = false;
    }
    private void polarize() {
        if (!polarized) {
            double x = x();
            double y = y();
            r = Math.sqrt(x*x+y*y);
            theta = Math.atan2(y,x);
            polarized = true;
        }
    }
    public double r() { polarize(); return r; }
    public double theta() { polarize(); return theta; }
}
```

Use an Interface?

```
public interface Point {  
    public double x();  
    public double y();  
    public double r();  
    public double theta();  
}
```

```
class MultiCalcPoint implements Point {  
    private double x, y;  
    ...  
}
```

Pros: compact size

Cons: r and theta is calculated every time

```
class SingleCalcPoint implements Point {  
    private double x, y;  
    private double r, theta;  
    ...  
}
```

Pros: r and theta is calculated only once

Cons: r and theta is calculated even if they are not used

```
class CachingPoint implements Point {  
    private double x, y;  
    private double r, theta;  
    private boolean polarized;  
    ...  
}
```

Pros: r and theta is calculated only once and only if they are ever used

Cons: Most amount of space

Can you hide the constructors?

```
class Graphics {  
    void drawLine(Point p1, Point p2) { ... }  
    ...  
}
```

No need to know the implementation

ERROR! Cannot instantiate an interface.
Need to know the implemented class

...

```
screen.drawLine(new Point(5,10),  
                Point(10,5));
```

What class to use?
MultiCalcPoint, SingleCalcPoint or CachingPoint?

What about if the implementer comes up with a better implementation?

All constructors in the client code need to be changed

Factory Methods

```
class PointProcs {  
    public static Point makePoint() { ... }  
    ...  
}  
  
...  
  
screen.drawLine(PointProcs.makePoint(5,10),  
                PointProcs.makePoint(10,5));
```

Create a “factory method” instead of a constructor

- Only a single place to modify to change the class

- All objects will be of the same class

This is a “Design Pattern”

interfaces and specifications

```
public interface Matrix {  
    int get(int row, int col);  
    void set(int row, int col, int val);  
    int rows();  
    int cols();  
}
```

interfaces and specifications

```
public interface Matrix {  
    // specfield width : integer  
    // specfield height : integer  
    // specfield cells : map from integer pair to integer  
  
    // returns: cells[<row,col>] if it exists, otherwise 0  
    int get(int row, int col);  
  
    // modifies: this.cells  
    // effects: cells[<row,col>] = val  
    void set(int row, int col, int val);  
  
    // returns: height  
    int rows();  
  
    // returns: width  
    int cols();  
}
```

Omitted bounds checking to
save screen space

interfaces and specifications

```
public interface Matrix {
    // specfield width : integer
    // specfield height : integer
    // specfield cells : set of 3-tuples of integers

    // returns: if for some v <row,col,v> is in cells,
    //           then return v; otherwise return 0
    int get(int row, int col);

    // modifies: this.cells
    // effects:
    //   cells = {<r,c,v> in \old(cells) | (r!=row || c!=col)}
    //           + <row,col,val>
    void set(int row, int col, int val);

    // returns: height
    int rows();

    // returns: width
    int cols();
}
```

Omitted bounds checking to
save screen space

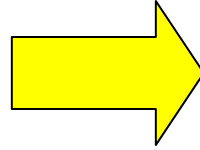
flat array implementation

```
public class OneDimArrayMatrix implements Matrix {  
  
    private int[] data;  
    private int width;  
  
    public OneDimArrayMatrix(int r, int c) {  
        data = new int[r*c];  
        width = c;  
    }  
  
    public int get(int r, int c)  
        { return data[r*width+c]; }  
  
    public void set(int r, int c, int v)  
        { data[r*width+c] = v; }  
  
    public int rows() { return data.length/width; }  
    public int cols() { return width; }  
}
```

Flat Array Representation

Matrix

1	2	3
4	5	6



data

1	2	3	4	5	6
---	---	---	---	---	---

width 3

Abstraction Function (AF)

Says how representation should be interpreted

Tells you how to get from the values of the *rep fields* (data) to the values of the *spec fields* (width, height, cells)

Representation Invariant (RI)

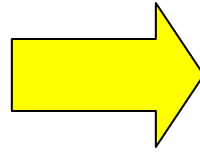
Some possible values of rep fields may not correspond to a valid state of the object

The representation invariant gives conditions that must be satisfied by the rep fields for interpretation to be possible

Flat Array Representation

Matrix

1	2	3
4	5	6



data

1	2	3	4	5	6
---	---	---	---	---	---

width 3

Abstraction Function:

`specfield width = width`

`specfield height = data.length/width`

`specfield cells = {<r,c,data[r*width+c]>`

`| 0 ≤ r < height,`
`0 ≤ c < width}`

Representation Invariant:

data should be non-null

data.length should be a multiple of width

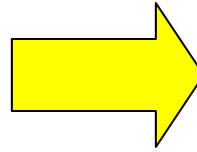
2D array implementation

```
public class TwoDimArrayMatrix implements Matrix {  
  
    private int[][] data;  
  
    public TwoDimArrayMatrix(int r, int c)  
        { data = new int[r][c]; }  
  
    public int get(int r, int c)  
        { return data[r][c]; }  
  
    public void set(int r, int c, int v)  
        { data[r][c] = v; }  
  
    public int rows() { return data.length; }  
    public int cols() { return data[0].length; }  
}
```

2D Array Representation

Matrix

1	2	3
4	5	6



data

1	2	3
4	5	6

Abstraction Function:

???

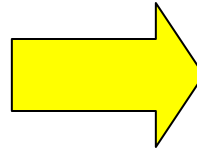
Representation Invariant:

???

2D Array Representation

Matrix

1	2	3
4	5	6



data

1	2	3
4	5	6

Abstraction Function:

`specfield width = data.length`

`specfield height = data[0].length`

`specfield cells = {<r,c,data[r][c]> | 0 ≤ r < height,
0 ≤ c < width}`

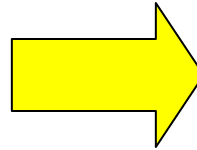
Representation Invariant:

???

2D Array Representation

Matrix

1	2	3
4	5	6



data

1	2	3
4	5	6

Abstraction Function:

`specfield width = data.length`

`specfield height = data[0].length`

`specfield cells = {<r,c,data[r][c]> | 0 ≤ r < height,
0 ≤ c < width}`

Representation Invariant:

data should be non-null

data[i] should be non null for all i, $0 \leq i < \text{data.length}$

data[i].length = data[j].length for all i, j

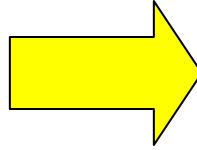
map implementation

```
public class HashMatrix implements Matrix {
    private Map<Pair,Integer> data;
    private int height, width;
    public HashMatrix(int r, int c) {
        data = new HashMap<Pair,Integer>();
        height = r;
        width = c;
    }
    public int get(int r, int c) {
        Integer v = data.get(new Pair(r,c));
        if (v==null) return 0;
        return v;
    }
    public void set(int r, int c, int v) {
        data.put(new Pair(r,c),v);
    }
    public int rows() { return height; }
    public int cols() { return width; }
}
```

Map Representation

Matrix

1	2	3
4	5	6



???

Abstraction Function:

???

Representation Invariant:

???

why interfaces instead of classes

Java design decisions:

A class has exactly one superclass

A class may implement multiple interfaces

An interface may extend multiple interfaces

Observation:

multiple superclasses are difficult to use and to implement

multiple interfaces, single superclass gets most of benefit

potential benefits of subclassing

Implementation reuse

Implementations need not repeat unchanged fields and methods – instead reuse from the superclass

Specification reuse

Clients who understand the superclass specification need only study novel parts of subclass

Ability to substitute new implementations

Clients may not need to change their code when new subclasses are developed, if they only handle

Many pitfalls! We'll see them through the term

Lists and Lists and Lists...

```
public class ListOfIntegers {  
    private Integer lst[];  
    public void add(int, Integer) { ..... }  
    public Integer get(int) { ..... }  
    ...  
}
```

```
public class ListOfObjects {  
    private Object lst[];  
    public void add(int, Object) { ..... }  
    public Object get(int) { ..... }  
    ...  
}
```

```
public class ListOfLists {  
    private List lst[];  
    public void add(int, List) { ..... }  
    public List get(int) { ..... }  
    ...  
}
```

.....
.....

Too many Lists!!

How about creating a list of what you defined?

List as an Interface

```
public Interface List {  
    public void add(int, ???);  
    public ??? get(int);  
    ...  
}
```

What is the type???

List as an Interface

```
public Interface List {
    public void add(int, Object);
    public Object get(int);
    ...
}
public class ListOfIntegers implements List {
    private Integer lst[];
    public void add(int, Object) { ... }
    public Object get(int) { ... }
    ...
}

List lst = new ListOfIntegers();
Integer fst = lst.get(0);
```

Error: Type mismatch
Object cannot be assigned to an Integer

Generic Types

```
public class List<T> {  
    private T lst[];  
    public void add(int, T) { ... }  
    public T get(int) { ... }  
    ...  
}
```

```
List<Integer> lst = new List<Integer>();  
Integer fst = lst.get(0);
```