Subtyping

Rob Miller
Fall 2007
Benefits and Risks of Subclassing

Implementation reuse
- If class B extends class A, then B inherits A’s fields and method bodies
- This is certainly better than reusing implementation by copying and pasting into B. (Why?)
- But this benefit has a downside: **subclassing breaks encapsulation**
  - A and B can’t be analyzed and changed independently anymore
  - We’ve mentioned this before, and will see more examples today

Client reuse
- Java type system allows B objects to be used wherever A objects are used
- So all client code written for A can use B as well
- This benefit also has a downside: **subclasses must be true subtypes**
  - If B objects don’t fulfill the same contract as A objects, then client code will break
  - We’ll see examples later, and define what we mean by subtyping
Subclassing Breaks Encapsulation #1

An example from the Java library: java.util.Properties

class Properties extends Hashtable {
    // Hashtable is an old library class that implements Map<Object, Object>,
    // so Properties inherits methods like:
    Object get(Object key) {...}
    void put(Object key, Object value) {...}

    // Rep invariant of Properties:
    // all keys and values are Strings
    String getProperty(String key) { return (String) this.get(key); }
    void setProperty(String key, String value) { this.put(key, value); }
}

Inherited superclass methods can break the subclass’s rep invariant!

© Robert Miller 2007
Subclassing Breaks Encapsulation #2

Example adapted from Bloch’s *Effective Java* (item 14)

class CountingList<E> extends ArrayList<E> {
    private int eltsAdded = 0; // total number of elements ever added
    @Override void add(E elt) {
        ++eltsAdded;
        super.add(elt);  }
    @Override void addAll(Collection c) {
        eltsAdded += c.size();
        super.addAll(c);  }
}

- What if ArrayList.addAll() works by calling add() \(n\) times?
- What if addAll(c) sometimes calls add() \(n\) times, and sometimes does it a different way, depending on the type of the collection \(c\)?

When a subclass overrides superclass methods, it may depend on how the superclass uses its own methods.
Subclassing Breaks Encapsulation #3

An example from our photo organizer

class Catalog { // version 1.0 of the photo organizer, no filter caching
    Set<Photo> photos;
    public void addNewPhoto(Photo photo) { photos.add(photo); }
}

class MyCatalog extends Catalog { // my subclass of the catalog
    public void removePhoto(Photo photo) { photos.remove(photo); }
}

class Catalog { // version 2.0 of the photo organizer, with filter caching
    Set<Photo> photos;
    Map<Filter, Set<Photo>> filterToMatches;
    // Rep invariant: all photos in cached matching sets are stored in the catalog
    // all f: Filter | f.filterToMatches in photos
}

When a class is subclassed, either it must freeze its implementation forever, or all its subclasses must evolve with its implementation.
Design Advice

Use composition rather than subclassing

- When B contains A as a field, the abstraction barrier between B and A is preserved
  ```java
class Properties extends Hashtable { ... }
class Properties {  private Hashtable table; ... }
```

- ```java
class CountingList<E> extends ArrayList<E> { ... }
class CountingList<E> implements List<E> {  
    private List<E> list;
    public CountingList<E>(List<E> list) { this.list = list; }
    ...
}
```

- CountingList is an instance of the **Wrapper** pattern
  - A wrapper modifies the behavior of another class without subclassing
  - Also decouples wrapper from the specific class it wraps – CountingList could wrap an ArrayList, LinkedList, even another CountingList

© Robert Miller 2007
Design Advice

**When subclassing is necessary, design for it**

- Define a *protected* API for your subclasses, in the same way you define a *public* API for clients
- Document how you use your own methods (e.g. does addAll() call add()?)
- Don’t expose your rep to your subclasses, or depend on them to maintain your rep invariant

© Robert Miller 2007
Subtyping vs. Subclassing

“B is a subtype of A” means “every B object is also an A object”

- Recall that a type is a set of values or objects, so subtyping is simply the subset relation
- In terms of specifications: “every B object satisfies the specification for A”

Subclassing (class B extends class A) is one way to declare a subtyping relationship in Java

- Other ways are implementing an interface (class B implements interface A) or extending an interface (interface B extends interface A)
- When we declare that B is a subtype of A, the Java type system allows us to use a B object wherever an A object is expected
  - i.e., when the declared type of a variable or method parameter is A, the actual runtime type can be B

© Robert Miller 2007
Substitution Principle

Subtype must be *substitutable* for supertype

- In particular, subtype must fulfill the same contract as the supertype, so that clients designed for the supertype can use subtype objects safely
- Subtype won’t surprise clients by failing to meet guarantees made by the supertype (e.g., postconditions)
- Subtype won’t surprise clients by making stronger demands than the supertype does (e.g. preconditions)

**B is a subtype of A if B’s specification is at least as strong as A’s specification**

- Java compiler guarantees part of this requirement automatically
  - For example, it ensures that every method in A appears in B, with a compatible type signature
- But Java doesn’t check every aspect of the specification
  - Preconditions, postconditions, and other properties are not checked!
Subclassing Requires Subtyping #1

Rat represents an immutable rational number

```java
class Rat {
    Rat plus(Rat that);
    Rat minus(Rat that);
}
```

MutableRat extends it to make a mutable rational

```java
class MutableRat extends Rat {
    public void addTo(Rat) {
        ...
    }
    public void subtractFrom(Rat) {
        ...
    }
}
```

➢ This declares to Java that MutableRat is a subtype of Rat ... but isMutableRat truly a subtype of Rat?
Is MutableRat Substitutable For Rat?

Clients that depend on the immutability of Rat may fail when given MutableRat values
- An immutable expression tree that contains Rat values
- A function that memoizes previously computed values in \( \text{HashMap<Rat,Rat>} \)
- Multithreaded code that uses the same Rat values in different threads

MutableRat fails to meet guarantees made by Rat
- Specifically, that the value of the object will never change

Mutable counterparts of immutable classes should not be declared as subtypes
- If you want a mutable rational class (e.g. for performance reasons), then it should not be a subtype of Rat
- String and StringBuffer/StringBuilder offer an example of how to do it right

© Robert Miller 2007
Subclassing Requires Subtyping #2

**BigNat** represents an immutable natural number ($\geq 0$)

```java
class BigNat {
    public BigNat plus(BigNat that);
    public BigNat minus(BigNat that);
}
```

**BigInt** represents an immutable integer (negatives too)

```java
class BigInt extends BigNat {
    private boolean isNegative;
}
```

> BigInt just adds a sign bit to BigNat. Makes sense, right?
Is BigInt Substitutable for BigNat?

Abstractly, it doesn’t make any sense

- Not every integer is a natural!
- The abstract type of BigInt isn’t a subset of the abstract type of BigNat, so it’s semantic nonsense to declare BigInt a subtype of BigNat

Practically, it’s risky

- A function declared to take a BigNat parameter has an implicit a precondition that its parameter is $\geq 0$
  
  double sqrt(BigNat n)
- But it can be passed a negative BigInt! What will happen?

BigInt fails to make guarantees made by BigNat

- Specifically, that the integer value is not negative
Subclassing Requires Subtyping #3

Immutable Square is a subtype of Immutable Rectangle

class Rectangle {
    public Rectangle(int w, int h);
    public int getWidth();
    public int getHeight();
}

class Square extends Rectangle {
    public Square(int side) { super(side, side); }
}

What about MutableSquare and MutableRectangle?
Is a `MutableSquare` a `MutableRectangle`?

class `MutableRectangle` {
    ... 
    /** @effects sets this rectangle’s dimensions to w x h */
    public void setSize(int w, int h);
} 

class `MutableSquare` extends `MutableRectangle` {
    /** @requires w = h 
     * @effects sets all edges to given size */
    void setSize(int w, int h); 

    /** @effects sets all edges to given size, 
        but throws BadSizeException if w != h */
    void setSize(int w, int h) throws BadSizeException;  

    /** @effects sets all edges to given size */
    void setSize(int side);

Declared subtypes should truly be subtypes

- When you declare to Java that “B is a subtype of A”, you should ensure that B actually satisfies A’s contract
  - B should guarantee all properties that A does, e.g. immutability
  - B’s methods should have the same or weaker preconditions, and the same or stronger postconditions

- This applies whether the declaration was made using subclassing (class B extends class A) or interface implementation (class B implements interface A)
Summary

Risks of subclassing
- Subclassing breaks encapsulation
- Use subclassing only for true subtyping

Substitution principle
- B is a true subtype of A if and only if B objects are substitutable for A objects anywhere that A is used
- Equivalently, the specification of B must imply the specification of A
- Preconditions of subtype must be weaker, postconditions stronger

Composition
- Wrapper pattern is an alternative to subclassing that preserves encapsulation