CSE 331
Software Design & Implementation

Hal Perkins
Winter 2013
Subtypes and Subclasses
(Slides by Mike Ernst and David Notkin)
What is subtyping?

Sometimes every B is an A

In a library database:
- every book is a library holding
- every CD is a library holding

Subtyping expresses this

B is a subtype of A means:

"every object that satisfies interface B also satisfies interface A"

Goal: code written using A's specification operates correctly even if given a B

Plus: clarify design, share tests, (sometimes) share code
Subtypes are substitutable

Subtypes are **substitutable** for supertypes
- Instances of subtype won't surprise client by failing to satisfy the supertype's specification
- Instances of subtype won't surprise client by having more expectations than the supertype's specification

We say that B is a **true subtype** of A if B has a stronger specification than A
- This is **not** the same as a **Java subtype**
- Java subtypes that are not true subtypes are confusing and dangerous
Subtyping and subclassing

Substitution (subtype) — a specification notion
  B is a subtype of A iff an object of B can masquerade as an object of A in any context
  Similarities to satisfiability (behavior of P is a subset of S)

Inheritance (subclass) — an implementation notion
  Abstract out repeated code
  To create a new class, just write the differences
  Every subclass is a Java subtype
    But not necessarily a true subtype

Outline of this lecture:
  Specification
  implementation (& Java details)
Subclasses support inheritance
Inheritance makes it easy to add functionality

Suppose we run a web store with a class for Products...

class Product {
    private String title;
    private String description;
    private float price;
    public float getPrice() { return price; }
    public float getTax() {
        { return getPrice() * 0.095f; }
    // ...  
    }
}

... and we need a class for Products that are on sale
Code copying is a bad way to add functionality

We would never dream of cutting and pasting like this:

class SaleProduct {
    private String title;
    private String description;
    private float price;
    private float factor;
    public float getPrice(){ return price*factor; }  
    public float getTax()
    {
        return getPrice() * .095;  
    }

    ...
}

Inheritance makes small extensions small

It’s much better to do this:

```java
class SaleProduct extends Product {
    private float factor;
    public float getPrice() {
        return super.getPrice()*factor;
    }
}
```
Benefits of subclassing & inheritance

Don’t repeat unchanged fields and methods

In implementation
  Simpler maintenance: just fix bugs once
In specification
  Clients who understand the superclass specification
  need only study novel parts of the subclass
Modularity: can ignore private fields and methods of superclass (if properly defined)
Differences are not buried under mass of similarities

Ability to substitute new implementations
  Clients need not change their code to use new subclasses
Subclassing can be misused

Poor planning leads to muddled inheritance hierarchy
  Relationships may not match untutored intuition
If subclass is tightly coupled with superclass
  Can depend on implementation details of superclass
  Changes in superclass can break subclass
    “fragile base class problem”

Subtyping and implementation inheritance are orthogonal
  Subclassing gives you both
 Sometimes you want just one (interfaces, composition)
 Subtyping is the source of most benefits of subclassing
interface Rectangle {
    // effects: fits shape to given size
    // this.post.width = w, this.post.height = h
    void setSize(int w, int h);
}

interface Square implements Rectangle {…}

Which is the best option for Square.setSize()?

1. // requires: w = h
    // effects: fits shape to given size
    void setSize(int w, int h);

2. // effects: sets all edges to given size
    void setSize(int edgeLength);

3. // effects: sets this.width and this.height to w
    void setSize(int w, int h);

4. // effects: fits shape to given size
    // throws BadSizeException if w != h
    void setSize(int w, int h) throws BadSizeException;
Square and rectangle are unrelated (Java)

Square is not a (true subtype of) Rectangle:
Rectangles are expected to have a width and height that can be changed independently
Squares violate that expectation, could surprise client

Rectangle is not a (true subtype of) Square:
Squares are expected to have equal widths and heights
Rectangles violate that expectation, could surprise client

Inheritance isn't always intuitive
Benefit: it forces clear thinking and prevents errors

Solutions:
Make them unrelated (or siblings under a common parent)
Make them immutable
Inappropriate subtyping in the JDK

Properties class stores string key-value pairs. It extends Hashtable functionality. What’s the problem?

```java
class Properties extends Hashtable<Object, Object> {  // simplified
    // modifies: this
    // effects: associates the specified value with the specified key
    public void setProperty(String key, String val) { put(key, val); }

    // returns: the string with which the key is associated
    public String getProperty(String key) { return (String) get(key); }
}
```

```java
Hashtable tbl = new Properties();
tbl.put("One", new Integer(1));
tbl.getProperty("One"); // crash!
```
Violation of superclass specification

Properties class has a simple rep invariant:
  keys and values are Strings
But client can treat Properties as a Hashtable
  Can put in arbitrary content, break rep invariant
From Javadoc:

Because Properties inherits from Hashtable, the put and putAll methods can be applied to a Properties object. ...
If the store or save method is called on a "compromised" Properties object that contains a non-String key or value, the call will fail.

Also, the semantics are more confusing than we've shown

getProperty("prop") works differently than
get("prop")!
Solution 1: Generics

Bad choice:

```java
class Properties extends Hashtable<Object, Object> { ... }
```

Better choice:

```java
class Properties extends Hashtable<String, String> { ... }
```

JDK designers deliberately didn’t do this. Why?
(postpone for now – we’ll get to generics shortly)
Solution 2: Composition

class Properties { // no "extends" clause!
    private HasTable<Object, Object> hashtable;   // the "delegate"

    // requires: key and value are not null
    // modifies: this
    // effects: associates specified value with specified key
    public void setProperty (String key, String value) {
        hashtable.put(key, value);
    }

    // effects: returns string with which key is associated
    public String getProperty (String key) {
        return (String) hashtable.get(key);
    }

    ...
Substitution principle

If B is a subtype of A, a B can always be substituted for an A.
Any property guaranteed by the supertype must be guaranteed by the subtype.

The subtype is permitted to strengthen & add properties.
Anything provable about an A is provable about a B.
If instance of subtype is treated purely as supertype – i.e., only supertype methods and fields queried – then result should be consistent with an object of the supertype being manipulated.

No specification weakening.
No method removal.
An overriding method has
  a weaker precondition
  a stronger postcondition.
Substitution principle

Constraints on methods

For each method in supertype, subtype must have a corresponding overriding method
may also introduce new methods

Each overriding method must:

Ask nothing extra of client ("weaker precondition")

Requires clause is at most as strict as in the supertype method

Guarantee at least as much ("stronger postcondition")

Effects clause is at least as strict as in the supertype method
No new entries in modifies clause
Substitution: spec weakening

Method inputs:
Argument types may be replaced with supertypes ("contravariance")
This places no extra demand on the client
Java forbids any change (Why?)

Method results:
Result type may be replaced with a subtype ("covariance")
    This doesn't violate any expectation of the client
No new exceptions (for values in the domain)
Existing exceptions can be replaced with subtypes
    This doesn't violate any expectation of the client
Substitution exercise

Suppose we have a method which, when given one product, recommends another:

```java
class Product {
    Product recommend(Product ref); }
```

Which of these are possible forms of method in `SaleProduct` (a true subtype of `Product`)?

```java
Product recommend(SaleProduct ref); // bad
SaleProduct recommend(Product ref); // OK
Product recommend(Object ref); // OK, but is Java overloading
Product recommend(Product ref) throws NoSaleException; // bad
```

Same kind of reasoning for exception subtyping, and modifies clause
JDK example: not a stronger spec

```java
class Hashtable {
    // class is somewhat simplified (generics omitted)
    // modifies: this
    // effects: associates the specified value with the specified key
    public void put (Object key, Object value);

    // returns: value with which the
    // specified key is associated
    public Object get (Object key);
}

class Properties extends Hashtable {
    // modifies: this
    // effects: associates the specified value with the specified key
    public void put (String key, String val) {
        super.put(key, val);
    }

    // returns: the string with which the key is associated
    public String get (String key) {
        return (String)super.get(key);
    }
}
```

Arguments are subtypes
Stronger requirement = weaker specification!

Result type is a subtype
Stronger guarantee = OK

Can throw an exception
New exception = weaker spec!
Java subtyping

Java types:

  Defined by classes, interfaces, primitives

Java subtyping stems from \( B \ extends \ A \) and \( B \ implements \ A \) declarations

In a Java subtype, each corresponding method has:

  same argument types
  
  if different, \textit{overloading}: unrelated methods
  
  compatible (covariant) return types
  
  a (somewhat) recent language feature, not reflected in (e.g.) \texttt{clone}

  no additional declared exceptions
Java subtyping guarantees

A variable’s run-time type (= the class of its run-time value) is a Java subtype of its declared type

    Object o = new Date();  // OK
    Date d = new Object();   // compile-time error

If a variable of *declared* (compile-time) type T holds a reference to an object of *actual* (runtime) type T', then T' is a (Java) subtype of T

Corollaries:

    Objects always have implementations of the methods specified by their declared type

    If all subtypes are true subtypes, then all objects meet the specification of their declared type

This rules out a huge class of bugs
Inheritance can break encapsulation

```java
public class InstrumentedHashSet<E> extends HashSet<E> {
    private int addCount = 0; // count attempted insertions
    public InstrumentedHashSet(Collection<? extends E> c) {
        super(c);
    }
    public boolean add(E o) {
        addCount++;
        return super.add(o);
    }
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getAddCount() { return addCount; }
}
```
Dependence on implementation

What does this code print?

```java
InstrumentedHashSet<String> s =
    new InstrumentedHashSet<String>();
System.out.println(s.getAddCount());  // 0
s.addAll(Arrays.asList("CSE", "331"));
System.out.println(s.getAddCount());  // 4!
```

• Answer depends on implementation of addAll() in HashSet
  – Different implementations may behave differently!
  – HashSet.addAll() calls add() ⇒ double-counting

• AbstractCollection.addAll specification states:
  – “Adds all of the elements in the specified collection to this collection.”
  – Does not specify whether it calls add()

• Lesson: designers should plan for their classes to be extended
Solutions

1. Change spec of HashSet
   Indicate all self-calls
   Less flexibility for implementers of specification
2. Eliminate spec ambiguity by avoiding self-calls
   a) “Re-implement” methods such as addAll
      Requires re-implementing methods
   b) Use a wrapper
      No longer a subtype (unless an interface is handy)
      Bad for callbacks, equality tests, etc.
Solution 2b: composition

```java
public class InstrumentedHashSet<E> {
    private final HashSet<E> s = new HashSet<E>();
    private int addCount = 0;
    public InstrumentedHashSet(Collection<? extends E> c) {
        this.addAll(c);
    }
    public boolean add(E o) {
        addCount++;
        return s.add(o);
    }
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return s.addAll(c);
    }
    public int getAddCount() { return addCount;
    }
    // ... and every other method specified by HashSet<E>
}
```

The implementation no longer matters

Delegate
Composition (wrappers, delegation)

Implementation **reuse** without **inheritance**

- Easy to reason about; self-calls are irrelevant
- Example of a “wrapper” class
- Works around badly-designed classes

Disadvantages (may be a worthwhile price to pay):
- May be hard to apply to callbacks, equality tests
- Tedious to write (your IDE will help you)
- Does not preserve subtyping
Composition does not preserve subtyping

**InstrumentedHashSet** is not a **HashSet** anymore
  So can't easily substitute it
It may be a true subtype of **HashSet**
  But Java doesn't know that!
  Java requires declared relationships
  Not enough to just meet specification
Interfaces to the rescue
  Can declare that we implement interface **Set**
  If such an interface exists
public class InstrumentedHashSet<E> implements Set<E> {
    private final Set<E> s = new HashSet<E>();
    private int addCount = 0;
    public InstrumentedHashSet(Collection<? extends E> c) {
        this.addAll(c);
    }
    public boolean add(E o) {
        addCount++;
        return s.add(o);
    }
    public boolean addAll(Collection<? extends E> c) {
        addCount += c.size();
        return s.addAll(c);
    }
    public int getAddCount() { return addCount; }
    // ... and every other method specified by Set<E>
}

Avoid encoding implementation details

What about this constructor?
InstrumentedHashSet(Set<E> s) {
    this.s = s;
    addCount = s.size();
}
Interfaces and abstract classes

Provide interfaces for your functionality
  The client codes to interfaces rather than concrete classes
  Allows different implementations later
  Facilitates composition, wrapper classes
    Basis of lots of useful, clever tricks
    We'll see more of these later
Consider providing helper/template abstract classes
  Can minimize number of methods that new implementation must provide
  Makes writing new implementations much easier
  Using them is entirely optional, so they don't limit freedom to create radically different implementations
Java library interface/class example

// root interface of collection hierarchy
interface Collection<E>

// skeletal implementation of Collection<E>
abstract class AbstractCollection<E>
    implements Collection<E>

// type of all ordered collections
interface List<E> extends Collection<E>

// skeletal implementation of List<E>
Abstract class AbstractList<E> extends
    AbstractCollection<E> implements List<E>

// an old friend...
class ArrayList<E> extends AbstractList<E>
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 the benefit
Pluses and minuses of inheritance

Inheritance is a powerful way to achieve code reuse

Inheritance can break encapsulation

A subclass may need to depend on unspecified details of the implementation of its superclass
  e.g., pattern of self-calls

Subclass may need to evolve in tandem with superclass
  Safe within a package where implementation of both is under control of same programmer

Authors of superclass should design and document self-use, to simplify extension
  Otherwise, avoid implementation inheritance and use composition instead