Subtypes

CSE 331
University of Washington

Michael Ernst
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 specification B also satisfies specification 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 (preconditions *and* postconditions)
  - If code is written to handle a Shape, it works if supplied a Circle
- 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
  – Any fact about A objects is true about B objects
  – Similarities to satisfiability (behavior of P is a subset of S)

• Inheritance (subclass) — an implementation notion
  – Factor 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() * 0.101; }
    ...
}

... 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() * 0.101; }
    ...
}
Inheritance makes small extensions small

It’s much better to do this:

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 a mass of similarities

Ability to substitute new implementations
  Clients can use new subclasses without changing their code
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: subtyping without inheritance
      • Composition: reuse implementation without subtyping
Every square is a rectangle  (elementary school)

interface Rectangle {
    // effects: fits shape to the given size
    // this.post.width = w, this.post.height = h
    void setSize(int w, int h);
}
interface Square implements Rectangle {...}

Which of these options are permissible 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 \( w \neq h \)
   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 width and height
  Rectangles violate that expectation, could surprise client

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

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

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

class HasTable<K,V> { 
    // modifies: this
    // effects: associates the specified value with the specified key
    public void put(K key, V value);

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

// Keys and values are strings.

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 getPropertyValue(String key) { return (String)get(key); } 
}

Properties p = new Properties();
Hashtable tbl = p;
tbl.put("One", new Integer(1));
p.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 I've shown

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

Bad choice:
class Properties extends Hashtable<Object,Object> { ... }

Better choice:
class Properties extends Hashtable<String,String> { ... }

Why didn’t the JDK designers make this choice?
Backward compatibility
- Properties was defined before Java had generics
- Only Hashtable<Object,Object> is compatible with all clients that might exist
class Properties {  // no "extends" clause!

private Hashtable<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 for classes

- 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:
  - Anything provable about an A is provable about a B.
  - If an instance of a subtype is treated purely as supertype (only supertype methods and fields queried) then result should be consistent with an object of the supertype being manipulated.
- Subtype may strengthen the spec:
  - May add new methods.
  - An overriding method must have a stronger or equal spec.
- Subtype may not weaken the spec:
  - No method removal.
  - No overriding method with a weaker spec.
A stronger spec’s Java signature

• Method **inputs**: weaker precondition
  – Parameter types of \( A.\texttt{foo}() \) may be replaced by supertypes in \( B.\texttt{foo}() \) in the subclass (“contravariance”)
  – This places no extra demand on the client
  – Java **forbids** any change (Why?)

• Method **results**: stronger postcondition
  – Result type of \( A.\texttt{foo}() \) may be replaced by a subtype in \( B.\texttt{foo}() \) in the subclass (“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
class Hashtable {

    // 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, OR
    // throws a ClassCastException
    public String get(String key) {
        return (String)super.get(key);
    }
}
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, *overloading*: unrelated methods
  – compatible (covariant) return types
    • not reflected in (e.g.) `clone` which predates this Java capability
  – 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
  
  \[
  \text{Object } o = \text{new Date(); // OK} \\
  \text{Date } d = \text{new Object(); // compile-time error}
  \]
  
  - If a variable of declared (compile-time) type \( T_c \) holds a reference to an object of actual (runtime) type \( T_r \), then \( T_r \) is a (Java) subtype of \( T_c \)

- 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
Clients can infer implementation details

• Client use of \(==\) can reveal library caching
  – Return existing immutable value, rather than creating a new value
• Client use of iterator can reveal whether library stores data in sorted order
• Client use of subclassing can reveal self-calls in implementation

• Lesson: Don’t do that!
• Clients should not observe behavior not promised by the spec
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!
  – If `HashSet.addAll()` calls `add()`, then 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()`

• Specification made no promises about implementation details

• Clients shouldn’t assume unspecified implementation behavior
How to get a count of insertions

1. Change spec of **HashSet** (eliminate ambiguity)
   - Strengthen the spec
     - Indicate all self-calls (maybe indicate none are made)
   - Less flexibility for implementers of specification
     - May require re-implementing methods
   - Most clients don’t care

2. Use a wrapper
   - No dependence on **HashSet** spec
   - No longer a subtype (might be able to use an interface)
     - Bad for callbacks, equality tests, etc.
Solution 2: Composition (wrapper)

```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>
}
```
Composition (wrappers, delegation)

Implementation *reuse without inheritance*

- Easy to reason about; self-calls are irrelevant
- Enables control of implementation details
  - Can also work around badly-designed classes
- Disadvantages (might be a worthwhile price to pay):
  - Does not preserve subtyping
  - May be hard to apply to callbacks, equality tests
  - Tedious to write (your IDE will help you)
Composition does not preserve subtyping

- **InstrumentedHashSet** is **not a HashSet** anymore
  - So can't 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 the class implements interface **Set**
  - Requires that such an interface exists
Interfaces reintroduce Java subtyping

```java
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>
}
```

What’s bad about this constructor?

```java
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, etc.

• Consider providing helper/template abstract classes
  – Abstract class:
    • Declared with `abstract class`
    • Does not implement all methods
    • Cannot be instantiated
  – Concrete subclass only implements missing methods
  – Using an abstract class optional; doesn’t limit freedom to create different implementations of an interface
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>

class ArrayList<E> extends AbstractList<E>
Interfaces add flexibility to Java

• Java design decisions:
  – A class has exactly one superclass
  – A class may implement multiple interfaces
  – An interface may extend multiple interfaces

• Justification for Java design decisions:
  – 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
• A subclass can observe unspecified implementation details
  – Example: pattern of self-calls
• If a class needs to control implementation details:
  – Author of superclass may design and document self-use, to simplify this type of extension
  – Client can avoid inheritance and use composition instead
Concrete, abstract, or interface?

Telephone
   $10 corded, speakerphone, cellphone, Skype, VOIP phone

TV
   CRT, Plasma, DLP, LCD

Table
   Dining table, Desk, Coffee table

Coffee
   Espresso, Frappuccino, Decaf, Iced coffee

Computer
   Laptop, Desktop, Server, Cloud, Phone

CPU
   x86, AMD64, ARM, PowerPC

Professor
   Ernst, Notkin
Type qualifiers

A way of using subtyping when you can’t write a new class
  – Can express more than Java’s built-in type system

Date is a type
  – @Nullable Date is a type
  – @NonNull Date is a type

```java
Date d;               // 7/4/1776, 1/14/2011, null, …
@Nullable Date nbleD; // same values as Date
@NonNull Date nnD;    // 7/4/1776, 1/14/2011, …

nbleD.getMonth();    // compile-time (not run-time!) error
nnD.getMonth();      // OK
```
Nullness subtyping relationship

- Which type hierarchy is best?

- A subtype has fewer values
- A subtype has more operations
- A subtype is substitutable
- A subtype preserves supertype properties