CSE 331
Software Design & Implementation

Hal Perkins
Winter 2012
Subtypes and Subclasses
(Slides by David Notkin and Mike Ernst)
Very quick recap: satisfies

- Procedural specification and implementations that satisfy these specifications
  - For specification $S$ and program $P$, $P$ satisfies $S$ iff
    - Every behavior of $P$ is permitted by $S$
    - “The behavior of $P$ is a subset of $S$”

- Abstract data type specification and implementations that satisfy such specifications – more complicated, but the same idea

- These are approaches for defining, reasoning about, testing and implementing software that satisfy specific expectations
Similarity

• Sometimes it is valuable to take advantage of existing specifications and/or implementations to develop a similar piece of software

• That is, we’d like to develop a similar artifact (specification or implementation) not entirely from scratch, but rather as a delta from the original
  \[ A' = A + \Delta A' \]

• Describing the differences and sharing the similarities can simplify development, increase confidence in the properties of the artifact, help in understanding the problem space, etc.
Similarity in software development

- The field has many ways to exploit this notion of similarity – examples include
  - Procedures with parameters – share the algorithm, differ in the data
  - Object-oriented subclassing
  - Object-oriented subtyping
  - Monads in functional programming
  - And many more…

- Just like similarity is confusing in the world, it can be confusing – but very valuable – in software development

These are related but distinct; and the distinctions are often confusing and confused.
Substitutability

- The notion of satisfiability was about whether an implementation met the expectations of a specification
- Substitutability will be the key issue in subtyping – can one specification (and its satisfying implementation) be substituted for another specification (and its satisfying implementation)?
Subtyping and Substitutability

- Subtyping uses substitutability to express the “is-a” relationship
  - A circle is-a shape; a rhombus is-a shape
  - A platypus is-a mammal; a mammal is-a vertebrate animal
  - A `java.math.BigInteger` is-a `java.lang.Number` is-a `java.lang.Object`
- When a programmer declares \( B \) to be a subtype of \( A \) that it means "every object that satisfies the specification of \( B \) also satisfies the specification of \( A \)"
  - Sometimes we call this a true subtype relationship
  - see next slide
Be careful!!!!!

• We are still talking about specifications, not implementations!
  – `java.math.BigInteger` might share absolutely positively no code at all with `java.lang.Object`

• Java subtypes/subclasses are not necessarily true subtypes
  – No type system, including Java’s, can determine the behavioral properties that would be needed to ensure this
    • Details beyond 331
  – Java subtypes that are not true subtypes are confusing at best and dangerous at worst
Subclassing

• Subclassing uses inheritance to share code – take advantage of the similarity of parts of the implementation – enables incremental changes to classes

• Every Java subclass is a Java subtype but is not necessarily a true subtype

• Checking for true subtypes requires full specifications (and deeper checking, again beyond the scope of type systems)
Java subtypes

- Java types are defined by classes, interfaces, and primitives
- **B** is Java subtype of **A** if there is a declared relationship (**B extends A**; **B implements A**)
- Compiler checks that, for each corresponding method in a Java subtype:
  - same argument types
  - compatible result types
  - no additional declared exceptions
- Again: *not* the same as checking for a true subtype! No semantic behavior is considered
Compiler guarantees

- Objects are guaranteed to be Java subtypes of their declared type
  - If a variable of declared \textit{(compile-time, static)} type $T$ holds a reference to an object of actual \textit{(runtime, dynamic)} 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

- Rules out a huge class of bugs
Adding 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()*0.05; }
        // ...
    }

• ... and we decide we want another class for Products that are on sale
We could cut-and-paste

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() * 0.05; }
    //...
}

• Good idea? Bad idea? Why?
Inheritance makes small extensions small

- The code for the extension is in some sense comparable in size to the extension
- 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
  – Implementation: simpler maintenance, fix bugs once
  – Specification: clients who understand the superclass specification need only study novel parts of 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 just want one
  - Subtyping is source of most benefits of subclassing
Every square is a rectangle

interface Rectangle {
    // effects: fits shape to given size
    // thisPost.width = w, thisPost.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

- **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 – it does encourage clear thinking and prevents errors
  - Possible solution might be to make them incomparable (perhaps as siblings under a common parent, say **Shape**)  
  - Why isn’t the elementary school “every square is a rectangle” true when we think about them as true subtypes?

(im)mutability!
Substitution principle Revisited

• If B is a subtype of A, a B can always be substituted for an A
• Any property guaranteed by supertype must be guaranteed by 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 – only supertype methods and fields used – 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: redux

Constraints on methods
- For each method in a supertype, the subtype must have a corresponding (overriding) method
  - Also may 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’s 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: specification weakening

• Method inputs
  – Argument types may be replaced with supertypes ("contravariance")
  – This doesn't place any 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:
  
  \begin{verbatim}
  Product recommend(Product ref);
  \end{verbatim}

• Which of these are possible forms of method in a true subtype?
  
  - Product \textit{recommend}(SaleProduct ref); \  ☐  bad
  - SaleProduct \textit{recommend}(Product ref); \  ☐  OK
  - Product \textit{recommend}(Object ref); \  ☐  OK (overloading)
  - Product \textit{recommend}(Product ref)
    throws NoSaleException; \  ☐  bad

• Same kind of reasoning for exception subtyping and for \textit{modifies} clause
Interfaces and abstract classes

• Provide interfaces for your functionality
  – Lets client write code to satisfy interfaces rather than to satisfy concrete classes
  – Allows different implementations later
  – Facilitates composition, wrapper classes – design patterns we’ll see more about later

• Consider providing helper/template abstract classes for important interfaces – classes with partial or full implementations, designed for extension
  – Can minimize number of methods that new implementation must provide
  – Makes writing new implementations much easier
  – Using them is optional, so they don't limit freedom to create radically different implementations