# Subtypes and Subclasses

6.170 Lecture 14

Fall - ocw.mit.edu

Reading: Chapter 7 of *Program Development in Java* by Barbara Liskov

1   Subtypes

We have used closed arrows in module dependence diagrams and object models. In MDDs, a closed arrow from an implementation *ArrayList* to a specification *List* means that *ArrayList* "meets" the specification of *List*. In an object model, a closed arrow denotes a subset relationship between the two sets.

We can also draw a closed arrow from A to B in an MDD, where A and B are implementation parts as shown below.
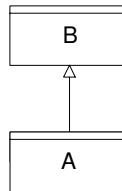
Here, the specifications of A and B are not drawn, but are implied in the MDD. A and B are modules with objects and associated methods. We say that that *A is a B* if every A object is also a B object. For instance, every automobile is a vehicle, and every bicycle is a vehicle, and every pogo stick is a vehicle; every vehicle is a mode of transport, as is every pack animal.

This subset relationship is a necessary but not sufficient condition for a *subtyping* relationship. Type A is a subtype of type B when A's specification implies B's specification. That is, any object (or class) that satisfies A's specification also satisfies B's specification, because B's specification is weaker.

The definition of subtyping depends on the definition of strong versus weak specifications. In 6.170, we will define *true subtyping* to mean that anywhere in the code, if you expect a B object, an A object is acceptable.[1] Code written to work with B objects (and to depend on their properties) is guaranteed to continue to work if it is supplied A objects instead; furthermore, the behavior will be the same, if we only consider the aspects of A's behavior that is also included in B's behavior. (A may introduce new behaviors that B does not have, but it may only change existing B behaviors in certain ways; see below.)

---

[1]With no hubris whatsoever, we call the 6.170 notion of subtyping, true subtyping to distinguish it from Java subtypes, which correspond to a weaker notion.

2 Example: Bicycles

Suppose we have a class for representing bicycles. Here is a partial implementation:

```
class Bicycle {
  private int framesize;
  private int chainringGears;
  private int freewheelGears;
  ...

  // returns the number of gears on this bicycle
  public int gears() { return chainringGears * freewheelGears; }
  // returns the cost of this bicycle
  public float cost() { ... }
  // returns the sales tax owed on this bicycle
  public float salesTax() { return cost() * .0825; }
  // effects: transports the rider from work to home
  public void goHome() { ... }
  ...
}
```

A new class representing bicycles with headlamps can accommodate late nights (or early mornings).

```
class LightedBicycle {
  private int framesize;
  private int chainringGears;
  private int freewheelGears;
  private BatteryType battery;
  ...

  // returns the number of gears on this bicycle
  public int gears() { return chainringGears * freewheelGears; }
  // returns the cost of this bicycle
  float cost() { ... }
  // returns the sales tax owed on this bicycle
  public float salesTax() { return cost() * .0825; }
  // effects: transports the rider from work to home
  public void goHome() { ... }
  // effects: replaces the existing battery with the argument b
  public void changeBattery(BatteryType b);
  ...
}
```

Copying all the code is tiresome and error-prone. (The error might be failure to copy correctly or failure to make a required change.) Additionally, if a bug is found in one version, it is easy to forget to propagate the fix to all versions of the code. Finally, it is very hard to comprehend the distinction the two classes by looking for differences themselves in a mass of similarities.

Java and other programming languages use subclassing to overcome these difficulties. Subclassing permits reuse of implementations and overriding of methods.

A better implementation of `LightedBicycle` is

```
class LightedBicycle extends Bicycle {
  private BatteryType battery;
  ...

  // returns the cost of this bicycle
  float cost() { return super.cost() + battery.cost(); }
  // effects: transports the rider from work to home
  public void goHome() { ... }
  // effects: replaces the existing battery with the argument b
  public void changeBattery(BatteryType b);
  ...
}
```

`LightedBicycle` need not implement methods and fields that appear in its superclass `Bicycle`; the `Bicycle` versions are automatically used by Java when they are not overridden in the subclass.

Consider the following implementations of the `goHome` method (along with more complete specifications). If these are the only changes, are `LightedBicycle` and `RacingBicycle` subtypes of `Bicycle`? (For the time being we will talk about subtyping; we'll return to the differences between Java subclassing, Java subtyping, and true subtyping later.)

```
class Bicycle {
  ...
  // requires: windspeed < 20mph && daylight
  // effects: transports the rider from work to home
  void goHome() { ... }
}

class LightedBicycle {
  ...
  // requires: windspeed < 20mph
  // effects: transports the rider from work to home
  void goHome() { ... }
}

class RacingBicycle {
  ...
  // requires: windspeed < 20mph && daylight
  // effects: transports the rider from work to home
  //          in an elapsed time of < 10 minutes
  //          && gets the rider sweaty
  void goHome() { ... }
}
```

To answer that question, recall the definition of subtyping: can an object of the subtype be substituted anywhere that code expects an object of the supertype? If so, the subtyping relationship is valid.

In this case, both `LightedBicycle` and `RacingBicycle` are subtypes of `Bicycle`. In the first case, the requirement is relaxed; in the second case, the effects are strengthened in a way that still satisfies the superclass's effects.

The `cost` method of `LightedBicycle` shows another capability of subclassing in Java. Methods can be overridden to provide a new implementation in a subclass. This enables more code reuse; in particular, `LightedBicycle` can reuse `Bicycle`'s `salesTax` method. When `salesTax` is invoked on a `LightedBicycle`, the `Bicycle` version is used instead. Then, the call to `cost` inside `salesTax` invokes the version based on the runtime type of the object (`LightedBicycle`), so the `LightedBicycle` version is used. Regardless of the declared type of an object, an implementation of a method with multiple implementations (of the same signature) is always selected based on the run-time type.

In fact, there is no way for an external client to invoke the version of a method specified by the declared type or any other type that is not the run-time type. This is an important and very desirable property of Java (and other object-oriented languages). Suppose that the subclass maintains some extra fields which are kept in sync with fields of the superclass. If superclass methods could be invoked directly, possibly modifying superclass fields without also updating subclass fields, then the representation invariant of the subclass would be broken.

A subclass may invoke methods of its parent via use of `super`, however. Sometimes this is useful when the subclass method needs to do just a little bit more work; recall the `LightedBicycle` implementation of `cost`:

```
class LightedBicycle extends Bicycle {
  // returns the cost of this bicycle
  float cost() { return super.cost() + battery.cost(); }
}
```

Suppose the `Rider` class models people who ride bicycles. In the absence of subclassing and subtypes, the module dependence diagram would look something like this:
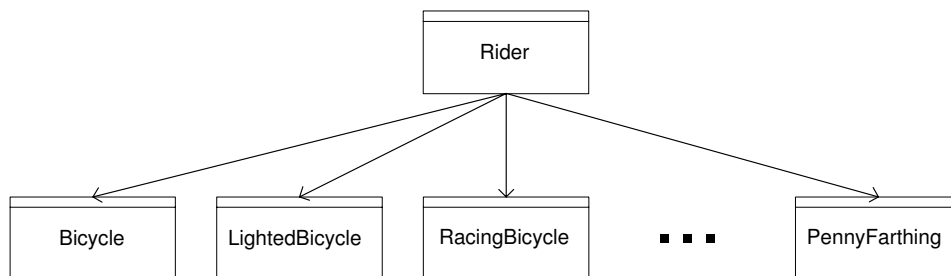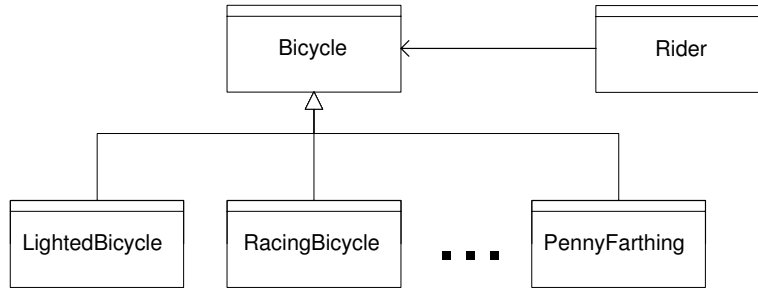

The code for `Rider` would also need to test which type of object it had been passed, which would be ugly, verbose, and error-prone.

With subtyping, the MDD dependences look like this:
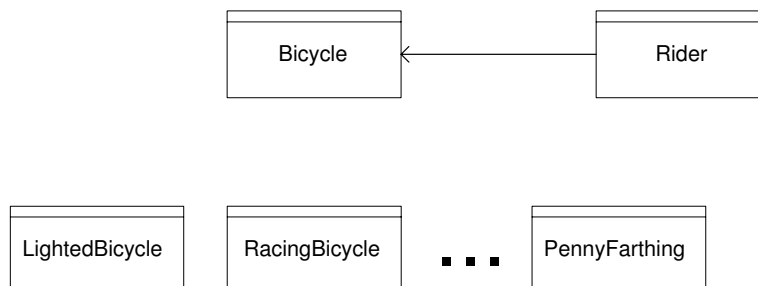
The many dependences have been reduced to a single one.

When subtype arrows are added, the diagram is only a bit more complicated:

Even though there are just as many arrows, this diagram is much simpler than the original one: dependence edges complicate designs and implementations more than other types of edge.

3   Substitution principle

The *substitution principle* is the theoretical underpinning of subtypes; it provides a precise definition of when two types are subtypes. Informally, it states that subtypes must be substitutable for supertypes. This guarantees that if code depends on (any aspect of) a supertype, but an object of a subtype is substituted, system behavior will not be affected. (The Java compiler also requires that the `extends` or `implements` clause names the parent in order for subtypes to be used in place of supertypes.)

The methods of a subtype must hold certain relationships to the methods of the supertype, and the subtype must guarantee that any properties of the supertype (such as representation invariants or specification constraints) are not violated by the subtype.

methods There are two necessary properties:

1. For each method in the supertype, the subtype must have a corresponding method. (The subtype is allowed to introduce additional, new methods that do not appear in the supertype.)
2. Each method in subtype that corresponds to one in the supertype:
   - requires less (has a weaker precondition)
     - there are no more "requires" clauses, and each one is no more strict than the one in the supertype method.
     - the argument types may be supertypes of the ones in the supertype. This is called contravariance, and it feels somewhat backward, because the arguments to the subtype method are supertypes of the arguments to the supertype method. However, it makes sense, because any arguments passed to the supertype method are sure to be legal arguments to the subtype method.
   - guarantees more (has a stronger postcondition)
     - there are no more exceptions
     - there are no more modified variables
     - in the description of the result and/or result state, there are more clauses, and they describe stronger properties
     - the result type may be a subtype of that of the supertype. This is called covariance: the return type of the subtype method is a subtype of the return type of the supertype method.

(The above descriptions should all permit equality; for instance, "requires less" should be "requires no more", and "less strict" should be "no more strict". They are stated in this form for ease of reading.)

The subtype method should not promise to have more or different results; it merely promises to do what the supertype method did, but possibly to ensure additional properties as well. For instance, if a supertype method returns a number larger than its argument, a subtype method could return a prime number larger than its argument.

As an example of the type constraints, if `A` is a subtype of `B`, then the following would be a legal overriding:

```
Bicycle B.f(Bicycle arg);
RacingBicycle A.f(Vehicle arg);
```

Method `B.f` takes a `Bicycle` as its argument, but `A.f` can accept any `Vehicle` (which includes all `Bicycle`s). Method `B.f` returns a `Bicycle` as its result, but `A.f` returns a `RacingBicycle` (which is itself a `Bicycle`).

properties Any properties guaranteed by a supertype, such as constraints over the values that may appear in specification fields, must be guaranteed by the subtype as well. (The subtype is permitted to strengthen these constraints.)

As a simple example from the book, consider `FatSet`, which is always nonempty.

```
class FatSet {
  // Specification constraints: this always contains at least one element

  ...

  // effects: if this contains x and this.size > 1, removes x from this
  void remove(int x);
}
```

Type `NotSoFatSet` with additional method

```
// effects: removes x from this
void reallyRemove(int x)
```

is not a subtype of `FatSet`. Even though there is no problem with any method of `FatSet` — `reallyRemove` is a new method, so the rules about corresponding methods do not apply — this method violates the constraint.

If the subtype object is considered purely as a supertype object (that is, only the supertype methods and fields are queried), then the result should be the same as if an object of the supertype had been manipulated all along instead.

In Section 7.9, the book describes the substitution principle as placing constraints on

- signatures: this is essentially the contravariant and covariant rules above. (A procedure's signature is its name, argument types, return types, and exceptions.)
- methods: this is constraints on the behavior, or all aspects of a specification that cannot be expressed in a signature
- properties: as above

6

4   Java subtypes

Java types are classes, interfaces, or primitives. Java has its own notion of subtype (which involves only classes and interfaces). This is a weaker notion than the true subtyping described above; Java subtypes do not necessarily satisfy the substitution principle. Further, a subtype definition that satisfies the substitution principle may not be allowed in Java, and will not compile.

In order for a type to be a Java subtype of another type, the relationship must be declared (via Java's `extends` or `implements` syntax), and the methods must satisfy two properties similar to, but weaker than, those for true subtyping:

1. for each method in the supertype, the subtype must have a corresponding method. (The subtype is allowed to introduce additional, new methods that do not appear in the supertype.)
2. each method in subtype that corresponds to one in the supertype
    - the arguments must have the same types
    - the result must have the same type
    - there are no more declared exceptions

Java has no notion of a behavioral specification, so it performs no such checks and can make no guarantees about behavior. The requirement of type equality for arguments and result is stronger than strictly necessary to guarantee type-safety. This prohibits some code we might like to write. However, it simplifies the Java language syntax and semantics.

4.1   Example: Square and rectangle

We know from elementary school that every square is a rectangle. Suppose we wanted to make `Square` a subtype of `Rectangle`, which included a `setSize` method:

```
class Rectangle {
  ...

  // effects: sets width and height to the specified values
  //          (that is, this.width' = w && this.height' = h)
  void setSize(int w, int h);
}

class Square extends Rectangle {
  ...
}
```

Which of the following methods is right for `Square`?

```
  // requires: w = h
  void setSize(int w, int h);

  void setSize(int edgeLength);

  // throws BadSizeException if w != h
  void setSize(int w, int h) throws BadSizeException;
```

The first one isn't right because the subclass method requires more than the superclass method. Thus, subclass objects can't be substituted for superclass objects, as there might be code that called `setSize` with non-equal arguments.

The second one isn't right (all by itself) because the subclass still must specify a behavior for `setSize(int, int)`; that definition is of a different method (whose name is the same but whose signature differs).

The third one isn't right because it throws an exception that the superclass doesn't mention. Thus, it again has different behavior and so `Square`s can't be substituted for `Rectangle`s. If `BadSizeException` is an unchecked exception, then Java will permit the third method to compile; but then again, it will also permit the first method to compile. Therefore, Java's notion of subtype is weaker than the 6.170 notion of subtype.

There isn't a way out of this quandary without modifying the supertype. Sometimes subtypes do not accord with our intuition! Or, our intuition about what is a good supertype is wrong.

One plausible solution would be to change `Rectangle.setSize` to specify that it throws the exception; of course, in practice only `Square.setSize` would do so. Another solution would be to eliminate `setSize` and instead have a

```
void scale(double scaleFactor);
```

method that shrinks or grows a shape. Other solutions are also possible.

5   Java subclassing

Subclassing has a number of advantages, all of which stem from reuse:

- Implementations of subclasses need not repeat unchanged fields and methods, but can reuse those of the superclass

- Clients (callers) need not change code when new subtypes are added, but can reuse the existing code (which doesn't mention the subtypes at all, just the supertype

- The resulting design has better modularity and reduced complexity, because designers, implementers, and users only have to understand the supertype, not every subtype; this is specification reuse.

A key mechanism that enables these benefits is overriding, which specializes behavior for some methods. In the absence of overriding, any change to behavior (even a compatible one) could force a complete reimplementation. Overriding permits part of an implementation to be changed without changing other parts that depend on it. This permits more code and specification reuse, both by the implementation and the client.

A potential disadvantage of subclassing is the opportunities it presents for inappropriate reuse. Subclasses and superclasses may depend on one another (explicitly by type name or implicitly by knowledge of the implementation), particularly since subclasses have access to the protected parts of the superclass implementation. These extra dependences complicate the MDD, the design, and the implementation, making it harder to code, understand, and modify.