Interface Inheritance: Behavioral Subtyping

Lecture 11
Intuition

- Some interfaces have significant overlap in functionality
  - bicycles and vehicles
    - both have owners and both can move
  - students and persons
    - both have names and both can be selected for juries
  - rectangles and shapes
    - both have a color
- These are all examples of an “is a” relationship
  - This is a common (but poor) intuitive litmus test
- Interfaces define types, ie sets of possible values

Every bicycle is a vehicle
Extending Interfaces

- One interface can extend another

```
interface X extends A, B { . . . }
```

- X implicitly includes all methods declared in A, B, and transitively above A and B
Recall: Narrowing vs Widening

- **Recall primitive types (eg long, int)**

- **Widening**
  - Assign a “small” value to a variable of “big” type
  - This is always ok and so can be done implicitly
    ```
    void f(int i) {
        long x = i; //widening: always ok
    }
    ```

- **Narrowing**
  - Assign a “big” value to a variable of “small” type
  - The correctness of this cannot be checked by compiler and so requires an explicit cast
    ```
    void f(long x) {
        int i = x; //narrowing: compile error
        int j = (int)x; //ok? programmer promise!
    }
    ```
Narrowing and Widening Objects

- Subinterfaces are “smaller” types than superinterfaces

Every student is a person
Narrowing and Widening Objects

- **Widening**
  - Assign a *subinterface* (declared type) to a variable of *superinterface* (declared) type
  - This is always ok and so can be done implicitly
    ```java
    void f(Student s) {
        Person p = s; //widening: always ok
    }
    ```

- **Narrowing**
  - Assign a *superinterface* (declared type) to a variable of *subinterface* (declared) type
  - This can not be checked by the compiler and so requires an explicit cast
    ```java
    void f(Person p) {
        Student s = p; //compiler complains
        Student s = (Student)p; //ok? prg promise!
    }
    ```
Argument Passing

- Method argument declared types must match signature
  ```java
  interface Course {
    void enroll(Student s) { ... }
  }
  interface Jury {
    void select(Person p) { ... }
  }
  ```
- Automatic (implicit) widening
  ```java
  Student s = ...;
  cse421.enroll(s); //ok (exact match)
  someJury.select(s); //ok (automatic widening)
  ```
- Cast for (explicit) narrowing
  ```java
  Person p = ...;
  someJury.select(p); //ok (exact match)
  cse421.enroll(p); //compiler complains (narrowing)
  cse421.enroll((Student)p); //ok? programmer promise!
  ```
Simple Rule

- A variable / parameter of declared type T can refer to an object of dynamic type “at or below” T

```java
void f(Creature c) {
    . . .
    int a = c.getAge();
    . . .
}
```
Behavioral Subtyping

- Informally, A is a *behavioral subtype* of B when it does everything B does (and maybe more)
  - Everywhere a B is expected, an A can be used instead
- Must satisfy the Substitution Principle:
  - *Any* correct client that uses a B is *still correct* when given an A instead
- Example:
  - A class uses Creature (eg void f(Creature c))
  - Actual argument might be a Creature, Person, Student, or Undergrad
  - Implementation of f() should still be correct!
- Note: This is a requirement on the component provider (of A), *not* on the client
Substitution Principle

□ If Undergrad is a subtype of Student
  □ Any correct client of Student is still correct when given an Undergrad

□ If Undergrad not a subtype of Student
  □ There exists some correct client of Student that is no longer correct when given an Undergrad
Behavioral Subtyping Rules

- Subtype constraint $\Rightarrow$ supertype constraint
  - Hence the informal “is a” litmus test
  - This condition alone, however, is not sufficient

- Each method in subinterface:
  - Requires less than in superinterface
    - Add disjuncts (or) to requires clause
    - Must work under more conditions
    - Contravariance of argument types
  - Ensures more than in superinterface
    - Add conjuncts (and) to the ensures clause
    - Must guarantee more to client
    - Covariance of return types
A is Narrower than B (A is-a B)

- A’s invariant is “stronger”
  - $\text{Inv}_A \implies \text{Inv}_B$
- For each method, A “requires less”
  - $\text{Prem}_A \leq \text{Prem}_B$
  - $\text{Pren}_A \leq \text{Pren}_B$
- For each method, A “ensures more”
  - $\text{Post}_m^A \implies \text{Post}_m^B$
  - $\text{Post}_n^A \implies \text{Post}_n^B$
- Aside:
  - Omitted requires/ensures stands for true
  - Anything $\implies$ true
A is Narrower than B

//@mathmodel M
//@constraint Inv_A ==>//@constraint Inv_B
interface A {
    //@requires Prem
    //@ensures Postm
    int m(int x, int y);
    //@requires Pren
    //@alters this ==>
    //@ensures Postn
    void n(String s);
}

//@mathmodel M
//@constraint Inv_B
interface B {
    //@requires Prem
    //@ensures Postm
    int m(int x, int y);
    //@requires Pren
    //@alters this
    //@ensures Postn
    void n(String s);
}
Visualization: Spec of m()

Requires

\((x == 0) \lor (y == 0)\)

\(x \times y \geq 0\)

Ensures

\(0 < m\)

\(10 < m < 100\)
Example: BigNatural & BigInteger

- Should BigNatural extend BigInteger?
- For behavioral subtyping, ask:
  - Is BigNatural’s invariant stronger?
  - Do all BigNatural methods require less?
  - Do all BigNatural methods ensure more?
BigNatural Extends BigInteger?

```java
//@mathmodel n integer//@mathmodel n integer
//@constraint n >= 0//@constraint
interface BigNatural {
    //@alters n     //@alters n
    //@ens n = #n+1//@ens n = #n+1
    void increment(); void increment();
}

//@mathmodel n integer//@mathmodel n integer
//@constraint interface BigInteger {
    //@alters n//@alters n
    //@ens n = #n+1//@ens n = #n+1
    void increment(); void increment();
    //@alters n//@alters n
    //@ens n=max(0,#n-1)//@ens n = #n-1
    void decrement(); void decrement();
}
```
Example: BigNatural & BigInteger

- Should BigNatural extend BigInteger?
- Is invariant stronger?  **Yes!**
  - BigNatural invariant is \( n \geq 0 \)
  - BigInteger invariant is true
- Do methods require less?  **Yes!**
  - increment() requires the same (true) in both
  - decrement() requires the same (true) in both
- Do methods ensure more?  **No!**
  - BigNatural decrement() ensures \( n > 0 \implies n = n-1 \)
  - BigInteger decrement() ensures \( n = n-1 \)
- Example client code that illustrates the problem
  ```java
  BigInteger noop(BigInteger i) {
      i.decrement();
      i.increment();
      return i;
  }
  ```
  - noop() is correct for BigInteger, but not for BigNatural
Example: Square & Rectangle

- These interfaces have similar abstract state (mathematical model)
  - two components: length, width

- These interfaces have similar public behavior (methods)
  - getArea(): returns the area (ie length * width)
  - widthStretch(): changes width of figure
  - lengthStretch(): changes length of figure

- Should we use inheritance?
  - Square extends Rectangle?
  - Rectangle extends Square?
Square Extends Rectangle?

```java
//@mathmodel l,w
//@constraint l = w
interface Square {

    //@ens getArea=l*w
    float getArea();

    //@alters l,w
    //@ens w = i*#w (&& l = i*#l)
    void widthStretch (int i);
}

//@mathmodel l,w
//@constraint
interface Rectangle {

    //@ens getArea=l*w
    float getArea();

    //@alters w
    //@ens w = i*#w (&& l = #l)
    void widthStretch (int i);
}
```
Example: Square is a Rectangle?

- Is invariant stronger? Yes!
  - Square invariant is length = width and both are >= 0
  - Rectangle invariant is length and width both >= 0
- Do methods require less? Yes!
  - all methods require true in both classes
- Do methods ensure more? No!
  - Square widthStretch(s) ensures length = #length * s
  - Rectangle widthStretch() ensures length = #length
- Example client code that illustrates the problem

  ```java
  Rectangle alwaysTrue(Rectangle r) {
    double initialArea = r.getArea();
    double finalArea = r.widthStretch(2).getArea();
    return (finalArea == 2*initialArea);
  }
  ```

  - alwaysTrue is correct for Rectangle, but not for Square
Rectangle Extends Square?

interface Rectangle {
    float getArea();
    void widthStretch(int i);
}

interface Square {
    float getArea();
    void widthStretch(int i);
}
Is invariant stronger? **No!**
- Square invariant is length = width and both are >= 0
- Rectangle invariant is length and width both >= 0

Do methods require less? **Yes!**
- all methods require true in both classes

Do methods ensure more? **No!**
- Square widthStretch(s) ensures length = #length * s
- Rectangle widthStretch() ensures length = #length

Example client code that illustrates the problem
```
Square alwaysTrue(Square s) {
    double intialArea = s.getArea();
    double finalArea = s.widthStretch(2).getArea();
    return(finalArea == 4*intialArea);
}
```
- alwaysTrue is correct for Square, but not for Rectangle
Java Support for Subtyping

- Java does not enforce behavioral contracts
- Support for behavioral subtyping limited to very weak promises, such as:
  - If B has a visible method m(), A has a visible method m() with same signature
  - A can not *decrease* visibility of m()
  - Parameter types must match exactly
    - Real contravariance would allow A.m’s parameter types to be supertypes of B.m’s parameter types
  - Return type *can be* a subtype (covariance)
  - If B’s method m() can not throw an exception of type E, neither can A’s m()
  - A can not *increase* the list of possible exceptions (we’ll talk about exceptions later...)

Real contravariance would allow A.m’s parameter types to be supertypes of B.m’s parameter types.

Summary

- Interface extensions
  - Declaration syntax
  - Vocabulary: super/sub, base/derived, parent/child
  - Widening (up) is automatic
  - Narrowing (down) requires explicit cast

- Behavioral subtyping
  - Substitution principle

- Subtyping rules
  - Strengthen the constraint
  - Weaken the requires of each method
  - Strengthen the ensures of each method

- Java rules (syntax)
  - Does not allow contravariance of argument types
  - Does allow covariance of return type