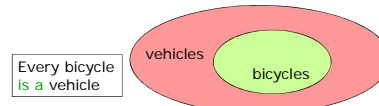


Interface Inheritance: Behavioral Subtyping

Lecture 11

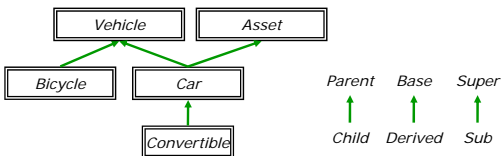
Intuition

- 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



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

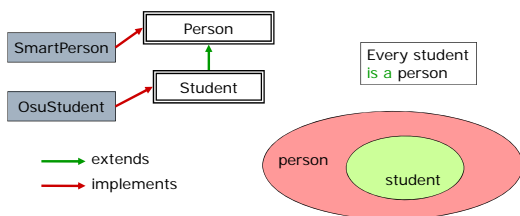
- Recall primitive types (eg long, int)
- Widening
 - Assign a “small” value to a variable of “big” type
 - This is always ok and so can be done implicitly

```
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



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

```
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

```
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

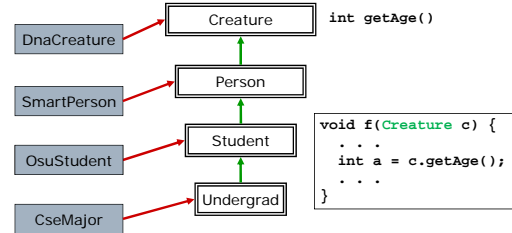

```
interface Course {
    void enroll(Student s) { . . . }
}
interface Jury {
    void select(Person p) { . . . }
}
```
- Automatic (implicit) widening


```
Student s = ...;
cse421.enroll(s); //ok (exact match)
someJury.select(s); //ok (automatic widening)
```
- Cast for (explicit) narrowing


```
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



The Problem

- Compiler enforces only:
 - Method *signatures* match in sub/super interfaces
- Not enforced by compiler:
 - Method *specifications* match
 - Mathematical models* match
 - Constraint* match
- That is, a subinterface can change the description of a method's behavior!

BigNatural and BigInteger

```
//@mathmodel integer n // @mathmodel integer n
//@constraint n >= 0 // @constraint
interface BigNatural { 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();
} }
```

BigNatural With Inheritance

```
//@mathmodel integer n // @mathmodel integer n
//@constraint n >= 0 // @constraint
interface BigNatural { interface BigInteger {
    extends BigInteger

    // @alters n
    // @ens n = #n+1
    void increment();

    // @alters n // @alters n
    // @ens n=max(0,#n-1) // @ens n = #n-1
    void decrement(); void decrement();
} }
```

Why Is This a Problem?

- These changes can break client code!
- Consider:


```
// @ensures i = #i
void skip(BigInteger i) {
    i.decrement();
    i.increment();
}
```
- The skip method
 - Is correct for BigInteger
 - Is not correct for BigNatural
- But Java allows skip to be called with a BigNatural argument (if BigNatural is a subinterface of BigInteger)

Dealing With This Problem

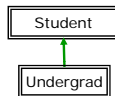
- Resolve/C++ approach:
 - Extensions *can only* add new operations
 - Extensions *can not* change the model or specs of extended component
- Alternative: Behavioral subtyping
 - Allow changes to method specs
 - Require new specs to be consistent with specs of extended component
- Beware:
 - Getting behavioral subtyping right is tricky and, even then, subtle problems can arise
 - More conservative approach is safer

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 foo(Creature c))
 - Actual argument might be a Creature, Person, Student, or Undergrad
 - Implementation of foo() 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
 - Must work under more conditions
 - Add disjuncts (or) to requires clause
 - “contravariance of arguments”
 - Ensures *more* than in superinterface
 - Must guarantee more to client
 - Add conjuncts (and) to the ensures clause
 - “covariance of return”

More Formally...

- A’s constraint is “stronger”
 - $Inv_A \Rightarrow Inv_B$
- For each method, A “requires less”
 - $Pre_A^f \leq Pre_B^f$
 - $Pre_A^g \leq Pre_B^g$
- For each method, A “ensures more”
 - $(Post_A^f \wedge Pre_B^f) \Rightarrow Post_B^f$
 - $(Post_A^g \wedge Pre_B^g) \Rightarrow Post_B^g$
- Asides:
 - Omitted requires/ensures stands for true
 - Anything \Rightarrow true

A is a Behavioral Subtype of B if...

```

//@mathmodel M_A == //@mathmodel M_B
//@constraint Inv_A ==> //@constraint Inv_B
interface A { interface B {

    //@requires Pre_A^f <== //@requires Pre_B^f
    //@ensures Post_A^f ==> //@ensures Post_B^f
    int f(int x, int y); int f(int x, int y);

    //@requires Pre_A^g <== //@requires Pre_B^g
    //@alters this ==> //@alters this
    //@ensures Post_A^g ==> //@ensures Post_B^g
    void g(String s); void g(String s);
} }
    
```

A is a Behavioral Subtype of B if...

```

Computer Science and Engineering @ The Ohio State University

M_A == M_B
Inv_A ==> Inv_B

interface A {                interface B {

    Pre_A^f <== Pre_B^f
    Post_A^f ^ Pre_B^f ==> Post_B^f
    int f(int x, int y);      int f(int x, int y);

    Pre_A^g <== Pre_B^g
    this SubsetOf this
    Post_A^g ^ Pre_B^g ==> Post_B^g
    void g(String s);        void g(String s);
}                             }

```

Example: A is Behavioral Subtype of B

```

Computer Science and Engineering @ The Ohio State University

//@mathmodel integer m      //@mathmodel integer m
//@cons m mod 10 = 0 ==> //@cons m is even

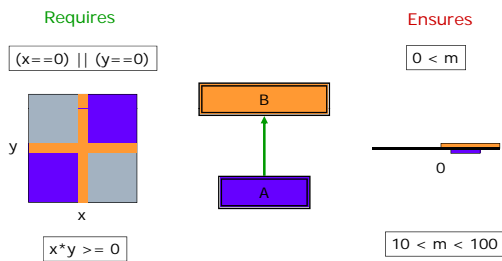
interface A {                interface B {

    //@req x*y >= 0 <== //@req x = 0 or y = 0
    //@ens 10 < m < 100 ==> //@ens 0 < m
    int f(int x, int y);      int f(int x, int y);

    void g(String s);        void g(String s);
}                             }

```

Visualization: Spec of f()



Example: BigInteger & BigNatural

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

BigInteger Extends BigInteger?

```

Computer Science and Engineering @ The Ohio State University

//@mathmodel integer n      //@mathmodel integer n
//@constraint n >= 0        //@constraint
interface BigNatural {     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: BigInteger & BigInteger

- Should BigInteger extend BigInteger?
- Is invariant stronger? **Yes!**
 - BigInteger 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!**
 - BigInteger decrement() ensures $n = \max(0, \#n-1)$
 - BigInteger decrement() ensures $n = \#n-1$
- Example client code that illustrates the problem


```

boolean alwaysTrue(BigInteger i) {
    String oldi = i.toString();
    i.decrement();
    i.increment();
    return (oldi.equals(i.toString()))
}

```

 - alwaysTrue is correct for BigInteger, 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?

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

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

    //@alters l,w        //@alters w
    //@ens w = i*#w      //@ens w = i*#w
    // and l = i*#l     // and l = #l
    void widthStretch   void widthStretch
        (int i);        (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 = $i * \#length$
 - Rectangle `widthStretch()` ensures length = $\#length$
- Example client code that illustrates the problem

```
boolean 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?

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

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

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

Example: Rectangle is a Square?

- 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 = $i * \#length$
 - Rectangle `widthStretch()` ensures length = $\#length$
- Example client code that illustrates the problem

```
boolean alwaysTrue(Square s) {
    double initialArea = s.getArea();
    double finalArea = s.widthStretch(2).getArea();
    return(finalArea == 4*initialArea);
}
```

 - `alwaysTrue` is correct for Square, but not for Rectangle

Java Support for Subtyping

- Java does not enforce behavioral contracts
 - Type checking ensures only that arguments match
- Support for behavioral subtyping limited to very weak promises, such as:
 - If B has a visible method `f()`, A has a visible method `f()` with same signature
 - A can not *decrease* visibility of `f()`!
 - Arguments must match exactly (too much!)
 - Return type *can be* a subtype (covariance)
 - If B's method `f()` can not throw an exception of type E, neither can A's `f()`
 - A can not *increase* the list of possible exceptions
 - We'll talk about exceptions later...

Summary

Computer Science and Engineering @ The Ohio State University

- 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