Substitution Principle

How do we know if saying $B$ is a subtype of $A$ is safe?

**Substitution Principle:** If $B$ is a subtype of $A$, everywhere the code expects an $A$, a $B$ can be used instead and the program still satisfies its specification.

**Signature Rule**

class A {
    public R m (P p) ;
}
class B extends A {
    public R m (P p) ;
}

$R_B$ must be a subtype of $R_A$: $R_B \leq R_A$

$P_B$ must be a super type of $P_A$: $P_B \geq P_A$

covariant for results, contravariant for parameters

**Subtype Condition 1: Signature Rule**

We can use a subtype method where a supertype methods is expected:
- Subtype must implement all of the supertype methods
- Argument types must not be more restrictive
- Result type must be at least as restrictive
- Subtype method must not throw exceptions that are not subtypes of exceptions thrown by supertype

**Subtype Condition 2: Methods Rule**

- Precondition of the subtype method must be weaker than the precondition of the supertype method. $m_A.pre \Rightarrow m_B.pre$
- Postcondition of the subtype method must be stronger than the postcondition of the supertype method. $m_B.post \Rightarrow m_A.post$

**Method Example**

class A {
    public int f (a A, x X) {
        // REQUIRES: a is initialized
        // EFFECTS: returns a.value * x.value
        return a.m (x);
    }
}

public class B extends A {
    // An A may be initialized or uninitialized.
    // An initialized A has an associated int value.
    public int m (x X) {
        // REQUIRES: this is initialized
        public class B extends A {
            // A B may be initialized or uninitialized.
            // A B may be awake or asleep.
            // An initialized B has an associated int value.
            public int m (x X) {
                // REQUIRES: this is initialized and awake
                Can't make the precondition stronger! The callsite might not satisfy it.
public int f (a A, x X) {
    // REQUIRES: a is initialized
    // EFFECTS: returns a.value * x.value
    return a.m (x); // Requires this is initialized
}

public class A {
    // An A may be initialized or uninitialized.
    // An initialized A has an associated int value.
    public int m (x X) {
        // REQUIRES: this is initialized
    }
}

public class B extends A {
    // A B may be initialized or uninitialized.
    // A B may be awake or asleep.
    // An initialized B has an associated int value.
    public int m (x X) {
        // REQUIRES: nothing
    }
}

Okay, precondition is weaker

Properties Example

public class StringSet {
    // Overview: An immutable set of Strings.
}

public class MutStringSet extends StringSet {
    // Overview: A mutable set of Strings.
    MutStringSet cannot be a subtype of StringSet, since it does not satisfy unchangable property.

Substitution Principle Summary

- Signatures: subtype methods must be type correct in supertype call sites: result is a subtype (covariant), parameters are supertypes (contravariant)
- Methods: subtype preconditions must be weaker than supertype preconditions (covariant); subtype postconditions must be stronger than supertype postconditions (contravariant)
- Properties: subtype must preserve all properties specified in supertype overview

Substitution Mystery

... (in client code)
MysteryType1 mt1;
MysteryType2 mt2;
MysteryType3 mt3;
...
    (anything could be here)
mt1 = mt2.m (mt3);

If the Java compiler accepts this code, which of these are guaranteed to be true:

a. The apparent type of mt2 is MysteryType2
b. At the last statement, the actual type of mt2 is MysteryType2
c. MysteryType2 has a method named m
d. The MysteryType2.m method takes a parameter of type MysteryType3
e. The MysteryType2.m method returns a subtype of MysteryType1
f. After the last statement, the actual type of mt1 is MysteryType1

Subtype Condition 3: Properties

Subtypes must preserve all properties described in the overview specification of the supertype.

StringSet could be a subtype of MutStringSet according to the properties rule.

...but couldn’t satisfy methods rule
Subtyping Rules

class A {
    public RA m (PA p) ;
}
class B extends A {
    public RB m (PB a);    mt1 = mt2.m (mt3);
}
If B <= A, what do we know about RB and PB?
RB must be a subtype of RA: RB <= RA
PA must be a subtype of PB: PB >= PA

Substitution Principle

Is this the only way?

Demystifying Subtyping

class A {
    ... (in client code)
    public RA m (PA p) ;
    MysteryType1 mt1;
    MysteryType2 mt2;
    MysteryType3 mt3;
    ... (anything could be here)
    mt1 = mt2.m (mt3);
}

a. The apparent type of mt2 is MysteryType2
   TRUE: the apparent type is obvious from the declaration.
b. At the last statement, the actual type of mt2 is MysteryType2
   FALSE: we only know the actual type <= MysteryType2
c. MysteryType2 has a method named m
   TRUE

d. The MysteryType2.m method takes a parameter of type MysteryType3
   FALSE: we only know it takes a parameter >= MysteryType3
e. The MysteryType2.m method returns a subtype of MysteryType1
   TRUE: the assignment type checking depends on this
f. After the last statement, the actual type of mt1 is MysteryType1
   FALSE: we only know that the actual type <= MysteryType1

RA must be a subtype of MysteryType1:
RA <= MysteryType1
MysteryType3 must be a subtype of PA:
PA >= MysteryType3

Substitution Principle Summary

Param Types    Psub >= Psuper     contravariant
Preconditions  pre_sub ⇒ pre_super for inputs
Result Type    Rsup ≤ Rsub        covariant
Postconditions post_sub ⇒ post_sup for outputs
Properties     properties_sub ⇒ properties_super

These properties ensure if sub is a subtype of supertype, code that is correct using an object of supertype is correct using an object of subtype.

Eiffel’s Rules

(Described in Bertrand Meyer paper out today)
Eiffel Rules

The types of the parameters in the subtype method may be subtypes of the supertype parameters.

How can Girl override set_roommate?

set_roommate (Girl g)
set_roommate (Boy b)

Opposite of substitution principle!

Substitution Principle / Eiffel

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Psub ≥ Psuper</th>
<th>Eiffel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preconditions</td>
<td>pre_sub ⇒ pre_super</td>
<td>Psub ≤ Psuper</td>
</tr>
<tr>
<td>Result</td>
<td>Rsup ≤ Rsuper</td>
<td>pre_sub ⇒ pre_super</td>
</tr>
<tr>
<td>Postconditions</td>
<td>post_sup ⇒ post_super</td>
<td></td>
</tr>
</tbody>
</table>

Meyer's paper is all about the contortions Eiffel needs to deal with non-substitutable subtypes.

Eiffel and I Can’t Get Up?

s: skier; g: girl; b: boy;

s := g;

... s.set_roommate (b);

Charge

Must it be assumed that because we are engineers beauty is not our concern, and that while we make our constructions robust and durable we do not also strive to make them elegant?

Is it not true that the genuine conditions of strength always comply with the secret conditions of harmony?

Gustav Eiffel