

Exercise 4

Behavioral Subtyping and Inheritance

October 17, 2014

Task 1

Consider the example in Slide 57 of the lecture 2:

```
class Number {
    int n;

    /// requires true
    /// ensures n == p
    void set(int p) { n = p; }
}

class UndoNaturalNumber extends Number {
    int undo;

    /// requires 0 < p
    /// ensures n == p && undo == old(n)
    void set(int p) { undo = n; n = p; }

    /// requires true
    /// ensures n == undo && undo == old(undo)
    void reset() { n = undo; }
}
```

where the invariants have been removed. Class `UndoNaturalNumber` is not a behavioral sub-type of `NaturalNumber`. One solution is to use specification inheritance. What are the effective pre/post-conditions of method `UndoNaturalNumber.set` according to the rules of Slides 67 and 70?

Solution

The effective precondition is equal to `true`. The effective postcondition is given by

$$(\text{old}(\text{true}) \implies n == p) \ \&\& \ (\text{old}(0 < p) \implies n == p \ \&\& \ \text{undo} == \text{old}(n))$$

which, if we make the simplifying assumption that formal parameters cannot be assigned to, i.e., `old(p) == p`, is equivalent to

$$n == p \ \&\& \ (0 < p \implies \text{undo} == \text{old}(n)).$$

Task 2 Behavioral Subtyping

Assume the following types in Java:

```
enum Shift {DayShift, NightShift, SpecialShift}

interface PostalWorker {
    boolean sick();
}
```

```

    ///ensures sick()
    void catchDisease();

    ///requires when == SpecialShift || when == DayShift
    ///requires !sick()
    int work(Shift when);
}

interface Bartender {
    boolean sick();

    ///ensures sick()
    void catchDisease();

    ///requires when == SpecialShift || when == NightShift
    ///requires !sick()
    int work(Shift when);
}

```

The `work()` method can be called in order to request that the corresponding person work the requested shift. The value returned by `work()` is the average hourly wage that was earned during the working shift including tips.

A

Now we introduce another interface:

```

interface HardWorker extends PostalWorker, Bartender {
    ///requires true
    int work(Shift when);
}

```

Assuming the improved rule for specification inheritance discussed in the course, what is the effective precondition of the `work()` method of the `HardWorker` interface?

— solution —

```

///requires
    (!sick() && (when == SpecialShift || when == DayShift))
|| (!sick() && (when == SpecialShift || when == NightShift))
|| true

```

which is equivalent to

```

///requires true

```

B

Now we add postconditions to all `work()` methods. Everything else remains as before.

```

interface PostalWorker {
    ...
    ///ensures result ≥ 15 && result ≤ 25
    int work(Shift when);
}

interface Bartender {
    ...
    ///ensures result ≥ 20 && result ≤ 30
    int work(Shift when);
}

```

```

interface HardWorker extends PostalWorker, Bartender {
    ...
    ///ensures result  $\geq$  25 && result  $\leq$  50
    int work(Shift when);
}

```

Assuming the improved rules for specification inheritance, what is the effective postcondition of the `work()` method of `HardWorker`?

— solution —

```

///ensures
( old(!sick() && (when == SpecialShift || when == DayShift))
   $\Rightarrow$  (result  $\geq$  15 && result  $\leq$  25) )
&& ( old(!sick() && (when == SpecialShift || when == NightShift))
   $\Rightarrow$  (result  $\geq$  20 && result  $\leq$  30) )
&& ( old(true)
   $\Rightarrow$  (result  $\geq$  25 && result  $\leq$  50) )

```

which is equivalent to

```

///ensures
( old(!sick() && when != NightShift)
   $\Rightarrow$  result == 25 )
&& ( old(!sick() && when == NightShift)
   $\Rightarrow$  (result  $\geq$  25 && result  $\leq$  30) )
&& ( old(sick())
   $\Rightarrow$  (result  $\geq$  25 && result  $\leq$  50) )

```

C

Consider the following code:

```

///requires worker != null
///requires !worker.sick()
int foo(HardWorker worker) {
    return worker.work(Shift.SpecialShift);
}

```

What is the range of possible return values of the `foo()` method?

— solution —

Only 25 is a possible return value.

D

Change the body of method `foo()` such that it calls the `work()` method of `worker` in a way that makes it possible for this call to return 50.

— solution —

```

int foo(HardWorker worker) {
    worker.catchDisease();
    return worker.work(Shift.SpecialShift);
}

```

Task 3

Investigate the behavior of the following Java code:

```
interface I {};  
  
class C {};  
  
public class E2_1  
{  
    public static void main(String [] argv)  
    {  
        C c = new C();  
        I i = (I) c;  
    }  
}
```

Try to compile it. If it compiles, try to execute it. What happens? Why?

Solution

The compiler allows the code to go through although it can't prove that `c` implements `I`. The reason is that there might be a subclass `D` of `C` such that `D` implements `I` and `c` might be an object of `D`. Here Java opts for the flexibility of dynamic type checking.

When the code executes a runtime exception is thrown, because `c` does not implement `I` and this is caught by the runtime check.

Task 4

Suppose that we have a database, for which we want an “automated key generation” feature. This means that each time the user inserts a new tuple, a unique key is automatically generated for the tuple by the system. An obvious way to do that is to write a counter, which increments by 1 the value that it returns each time it is called. The method that generates a new key is called `generate`.

1. Write a Java class `IncCounter` and an accompanying specification for such a counter.
2. Annotate the following Java class with specifications and show that it is not a behavioural subtype of `IncCounter`.

```
class DecCounter  
{  
    int key;  
    DecCounter () { key = 0; }  
    int generate () { return key--; }  
}
```

3. Write an abstract class `GenerateUniqueKey` together with a specification, such that both `IncCounter` and `DecCounter` are behavioural subtypes of `GenerateUniqueKey`. In the specification, you may use helper methods and fields.

Solution

1.

```
class IncCounter  
{  
    /// constraint old(key) <= key  
    int key;  
  
    IncCounter () { key = 0; }  
  
    /// ensures key=old(key)+1 ∧ result=old(key)  
    int generate () { return key++; }  
}
```

2. The postcondition for generate is $\text{key} = \text{old}(\text{key}) - 1 \wedge \text{result} = \text{old}(\text{key})$ and it is easy to see that it does not refine the postcondition of `IncCounter.generate`. The history constraint is $\text{old}(\text{key}) \geq \text{key}$ and also does not strengthen the one of `IncCounter`.
3. The abstract parent class can be declared using a helper pure method `boolean used(int)`. Informally, the helper method returns true if `x` has been used as a key before. Furthermore, the correctness of the class relies on the property that once a number is used, it never becomes unused again. This can be expressed with a two-state history constraint.

The definitions of the classes follow:

```
abstract class GenerateUniqueKey
{
    /// constraint  $\forall x:\text{int} \mid (\text{old}(\text{used}(x)) \Rightarrow \text{used}(x))$ 
    abstract boolean used(int);

    /// ensures  $\neg \text{old}(\text{used}(\text{result})) \wedge \text{used}(\text{result})$ 
    abstract int generate ();
}

class IncCounter // ... and similarly for DecCounter
{
    int key;
    IncCounter () { key = 0; }

    boolean used (int x)
    { return x < key; }

    /// ensures  $\text{key} == \text{old}(\text{key}) + 1 \wedge \text{result} == \text{old}(\text{key})$ 
    int generate () { return key++; }
}
```

Task 5

Consider the following Java classes and interfaces:

```
public interface IA
{
    IA g(IA x);
}

public interface IB extends IA
{
    IB g(IA x);
    IA g(IB x);
}

public interface IC extends IA
{
    IC g(IB x);
}

class B implements IB
{
    public IB g(IA x){System.out.print("B1");return null;}
    public IC g(IB x){System.out.print("B2");return null;}
}

class C implements IC
{
    public IC g(IA x){System.out.print("C1");return null;}
}
```

```

    public C g(IB x){System.out.print("C2");return null;}
}

class Main{
    public static void main(String[] args) {
        B b = new B();
        C c = new C();
        IA a1 = b;
        IA a2 = c;

        IA r1 = a1.g(a2);
        IA r2 = a2.g(b);
        IC r3 = b.g(b);
        IA r4 = c.g(a2);
        C r5 = c.g(b);
    }
}

```

What is the output of the execution of method main in class Main? Explain your answer.

Solution

The code will print B1 C1 B2 C1 C2:

a1 is of static type IA and dynamic type B, a2 is of static type IA: a1.g(a2) maps to IA.g(IA), which is overridden in IB as IB.g(IA) and then in B as B.g(IA).

a2 is of static type IA and dynamic type C, b is of static type B: a2.g(b) maps to IA.g(IA), which is overridden in IC as IC.g(IA) and then in C as C.g(IA).

b is of static type B and dynamic type B: b.g(b) maps to B.g(IB) (more specific than B.g(IA) - overload resolution).

c is of static type C and dynamic type C, a2 is of static type IA: c.g(a2) maps to C.g(IA).

c is of static type C and dynamic type C, b is of static type B: c.g(b) maps to C.g(IB) (more specific - overload resolution).

Task 6

Consider two classes Stack and Queue, implementing the standard LIFO/FIFO data structures, both of which have methods with the following signatures:

```

void push(Object o);
Object pop();
bool isEmpty();
int size();
void reverse();

```

- Despite having identical signatures, these two classes cannot be behavioral subtypes of one another. Why not?
- When implementing these two classes, is there any possibility of code reuse? If so, give details.
- Describe at least one way of reusing the code in one class by the other - which programming language features are needed for this to work?

Solution

- The intended behavior is that a Stack is LIFO, while a Queue is FIFO. Therefore, the pop and push have different behavior and so neither can be considered a behavioral subtype of the other.

- Depending on the internal representation, either the `pop()` or the `push()` method (but not both) could be reused, from one implementation to the other. For example, if one implements a Queue by pushing to the end of a linked list, and popping from the beginning, then a Stack could be implemented either by pushing on the beginning of the list and reusing the `pop()` method, or by reusing the `push()` method and popping from the end of the list. Furthermore, it's likely that the `isEmpty()`, `size()` and `reverse()` methods could all be reused.
- Any mechanism which allows code reuse without subtyping, e.g., private inheritance in C++ or aggregation. In both cases it would make sense to have a “common super class” used by both implementations. This super-class, however, would either be too wide (allowing insertion/removal at both ends) or rather thin (allowing only insertion on one side). In the wide case we could use a kind of linked list, for example, that can insert/remove at the beginning and end, and use private inheritance to expose only the relevant operations to the clients of each data structure.