ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 4

Behavioral Subtyping and Inheritance

October 18, 2013

Task 1

Let `C` be a class with an integer field `x` and a method `m`. Let `m` have

- Precondition `x>0`
- Postcondition `x<1`

Suppose now that there is a class `D` with an integer field `x` and a method `m`. In which of the following cases does the specification of `m` in `D` permit `D` to be a behavioral subtype of `C`?

a) Pre `x>0` Post `x<-1`

b) Pre `x>0` Post `x<2`

c) Pre `x>-1` Post `x<1`

d) Pre `x>2` Post `x<1`

e) Pre `x>-4` Post `x<-old(x)*old(x)`

f) Pre `true` Post `false`

Solution

	$\text{Pre}_{super} \Rightarrow \text{Pre}_{sub}$	$\text{Post}_{sub} \Rightarrow \text{Post}_{super}$	Behavioral subtyping
a	yes	yes	yes
b	yes	no	no
c	yes	yes	yes
d	no	yes	no
e	yes	yes	yes
f	yes	yes	yes

Task 2

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 subtype 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? Treat the formal parameter as constant (i.e. `old(p)==p`).

Solution

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

```
(old(true)==> n==p)&& (old(0<p)==> n==p && undo==old(n))
```

which is equivalent to

```
n==p && (0<p ==> undo==old(n)).
```

Task 3

Assume a language with structural subtyping, contravariant arguments, and covariant return types. Is it possible to create the classes `A`, `B`, and `C` that meet all of the following requirements?

1. `B` is a structural subtype of `A`, and `C` is a structural subtype of `B`.
2. `B` is not a behavioral subtype of `A`.
3. `C` is a behavioral subtype of both `A` and `B`.
4. The signatures of any two methods of `A`, `B`, or `C` should be different. For this exercise the signature is the combination of return type, method name, and argument order and types. Note that different signatures do not preclude structural subtyping.
5. The classes do not have any fields.

If it is possible to meet all of above requirements, write the classes `A`, `B`, and `C`.

If it is not possible to meet all requirements, explain why not. Then pick one requirement and remove it. Write down the classes `A`, `B`, and `C` that meet the remaining four requirements.

In both cases specify the behavior of the classes using contracts. You do not need to provide method bodies. You may use existing Java classes in your solution, if you want to.

Solution

All requirements can be met. Here are the corresponding classes:

```
class A {

   ///requires a > 0
   ///ensures result > 0
   Number foo(Integer a)
}

class B {

   ///requires a > 10
   ///ensures result > 0
   Number foo(Number a)
}
```

```
class C {
   ///requires true
   ///ensures result == 10 \/ result == 20
   Integer foo(Object o)
}
```

Task 4

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 5

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
{
  int key;
  IncCounter () { key = 0; }

  /// ensures key=old(key)+1 ∧ result=old(key)
  int generate () { return key++; }
}
```

2. The postcondition for `generate` is `key=old(key)-1 ∧ result=old(key)` and it is easy to see that it does not refine the postcondition of `IncCounter.generate`.

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 ∀ x:int | (old(used(x)) ⇒ used(x))
   abstract boolean used(int);

   /// ensures ¬old(used(result)) ∧ used(result)
   abstract int generate ();
}

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

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

   /// ensures key == old(key)+1 ∧ result == old(key)
   int generate () { return key++; }
}
```

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 FIFO, while a Queue is LIFO. 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.