Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Exercise 4

Behavioral Subtyping and Inheritance

October 16, 2015

Task 1

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 2

Consider the example in Slide 58 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 68 and 72?

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, 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 ==> undo==old(n)).
```

Task 3 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 ≥ 25 && result ≤ 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))
     ⇒ (result ≥ 15 && result ≤ 25) )
&& ( old(!sick() && (when == SpecialShift || when == NightShift))
     ⇒ (result ≥ 20 && result ≤ 30) )
&& ( old(true)
     ⇒ (result ≥ 25 && result ≤ 50) )
```

which is equivalent to

```
///ensures
   ( old(!sick() && when != NightShift)
     ⇒ result == 25 )
&& ( old(!sick() && when == NightShift)
     ⇒ (result ≥ 25 && result ≤ 30) )
&& ( old(sick())
     ⇒ (result ≥ 25 && result ≤ 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 4

Consider the following Java method and its pre- and postcondition, where $[0..n)$ denotes a right-open interval that includes 0 but excludes n (reminder: `final` parameters cannot be assigned to):

```
/// requires 0 < n ∧ xs ≠ null ∧ 2*n < xs.length
/// ensures ∀ i ∈ [0..n) :: xs[2*i] ≠ 0
void foo(final int n, final int[] xs)
```

Assume that method `foo` is overridden in a subclass, and that we do not use specification inheritance. Which of the following pre- and postconditions would **not** be allowed if the subclass should be a behavioral subtype?

(a)
```
requires 0 < n ∧ xs ≠ null ∧ 2*n < xs.length
ensures ∀ i ∈ [0..n) :: xs[2*i] ≥ n
```

(b) **CORRECT:**
```
requires 0 < n ∧ xs ≠ null
ensures ∀ i ∈ [0..n) :: xs[2*i] ≥ i
```

(c)
```
requires 0 ≤ n ∧ xs ≠ null ∧ 2*n ≤ xs.length
ensures ∀ i ∈ [0..n) :: xs[2*i] ≥ i+n
```

(d)
```
requires 0 ≤ n ∧ xs ≠ null ∧ 2*n ≤ xs.length
ensures ∀ i ∈ [0..n) :: xs[2*i] ≥ n
```

(e) All of the above would be allowed

Task 5

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();
```

A) Despite having identical signatures, these two classes cannot be behavioral subtypes of one another. Why not?

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.

B) When implementing these two classes, is there any possibility of code reuse? If so, give details.

solution

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.

C) 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

Any mechanism which allows code reuse without subtyping, e.g., private inheritance in C++ or aggregation.

Another option would be 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.

Task 6

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`.

A) Write a Java class `IncCounter` and an accompanying specification for such a counter.

solution

```
class IncCounter
{
   /// constraint old(key) <= key
  int key;

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

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

B) 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--; }
}
```

solution

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`. The history constraint is `old(key) >= key` and also does not strengthen the one of `IncCounter`.

C) 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

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++; }
}
```