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`

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`).

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.

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?

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.

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?