Tutorial 02 Type hierarchy

1 Chapter exercises

The figures below are scanned images of the relevant chapter exercises that are listed in the MD for this tutorial. Refer to the updated exercise notes below the figures.

Exercises

7.1 Define and implement a subtype of `IntList` (see Figures 7.11 and 7.12) that provides methods to return the smallest and largest elements of the list. Be sure to define the rep invariant and abstraction function, and to implement `repOk`.

7.8 Consider a type `Counter` with the following operations:

```
public Counter ( ) // EFFECTS: Makes this contain 0.
public int get ( ) // EFFECTS: Returns the value of this.
public void incr ( )
    // MODIFIES: this
    // EFFECTS: Increments the value of this.
```

Complete the specification of `Counter` by providing the overview section. Be sure to identify all properties of `Counter` objects.

7.9 Now consider a potential subtype of `Counter`, `Counter2`, with the following extra operations:

```
public Counter2 ( ) // EFFECTS: Makes this contain 0.
public void incr ( )
    // MODIFIES: this
    // EFFECTS: Makes this contain twice its current value.
```

Is `Counter2` a legitimate subtype of `Counter`? Explain by arguing that either the substitution principle is violated (for a non-subtype) or that it holds (for a subtype). Discuss how each operation of `Counter2` either upholds or violates the substitution principle.

7.10 Now consider another potential subtype of `Counter`, `Counter3`, with the following extra operations:

```
public Counter3 (int n) // EFFECTS: makes this contain n
public void incr (int n)
    // MODIFIES: this
    // EFFECTS: If n > 0 adds n to this.
```

Is `Counter3` a legitimate subtype of `Counter`? Explain by arguing that either the substitution principle is violated (for a non-subtype) or that it holds (for a subtype). Discuss how each operation of `Counter3` either upholds or violates the substitution principle.

7.11 Consider a type `IntBag`, with operations to `insert` and `remove` elements, as well as all the observers of `IntSet`. Bags are like sets except that elements can occur multiple times in a bag. Is `IntBag` a legitimate subtype of `IntSet`? Explain by arguing that either the substitution principle is violated (for a non-subtype) or that it holds (for a subtype).

Ex1.1. Ex 7.1

Change from `IntList` in the requirement to `java.util.List` and replace rep invariant and abstraction function by abstract properties. That is, you are to specify and implement a sub-type of `java.util.List` that provides methods for returning the smallest and largest elements of the list. Let's call this sub-type by the name `MaxMinIntList`.

<u>Note:</u> Interface `List` has a number of sub-types already provided in Java

1. Read the API documentation of interface `List`
2. Draw a UML class diagram of `List` and complete it with the essential members.
3. What are the class sub-types of `List`? As you identify these sub-types, add them to the UML class diagram of `List` to create a type hierarchy.
4. Choose a class sub-type of `List` that you are familiar with and use it to write a simple Java program to create a `List` containing the following integers 1, 5, 3, 2, 7, 9, 10, 4 and to display the list content to the standard console.
5. Decide whether to define `MaxMinIntList` as a direct sub-type of `List` or a sub-type of one of its class subtypes. Try to think of at least two designs for `MaxMinIntList`.

Ex1.2. Ex 7.8-11

To do these exercises, you need to refer to the slides about the meaning of sub-type in the lecture slides. Note that you need to **use the header rule** instead of *signature rule* (mentioned in the text book) as part of your discussion of the validity of the subtypes.

2 Additional Exercises

These exercises are designed to help you review the core type hierarchy concepts and techniques. Some of these exercises are based on the `Vehicle` type hierarchy example that was used in the lecture.

Ex2.1. Inheritance

1. Update the two classes `Bus` and `Car` so that their weight constraints are as follow:
   1.1. `Bus.weight` is in the range [5000.0, 20000.0] (kgs)
   1.2. `Car.weight` is in the range [1000.0, 2000.0] (kgs)
2. Update the two classes `Bus` and `Car` so that they now have the following constraints on the length dimension:
   2.1. `Bus.length` is in the range [4.0, 10.0] (meters)
   2.2. `Car.length` is in the range [1.5, 3.5] (meters)
3. Update class `Vehicle` to have a new attribute called `registrationNumber`. Based on your practical understanding of this attribute, decide a suitable data type and domain constraint for it.

   Note: you must update and/or define the operations that are relevant to the new attribute.
4. Update the two classes `Bus` and `Car` so that they each have a different domain constraint for the

attribute `registrationNumber` from the domain constraint defined in the class `Vehicle`. For example, if `Vehicle.registrationNumber`'s domain constraint specifies that the attribute can contain up to 12 alpha-numerical characters then `Bus.registrationNumber`'s and `Car.registrationNumber`'s could specify that it only contains up to 8 and (respectively) 6 such characters.

5. Update the three classes `Vehicle`, `Bus` and `Car` so that the `toString` method can be removed from `Bus` and `Car`, and that the inherited `toString` method from the class `Vehicle` now provides the accurate class label for not only `Vehicle` but also for `Bus` and `Car`.

   Hint: In Java, you can use the following statement in a method of a class to obtain the name of the run-time (actual) class of an object of that class: `this.getClass().getSimpleName()`.

6. Define in the class `Vehicle` a new operation named `travel`, which travels from a given point *A* to another point *B*. This operation can simply print a message to the standard console detailing the travelling (e.g. the type of vehicle, the two points, and the number of seats (i.e. passengers)). However, it must use a specialised symbol for the type of vehicle object that this operation is being invoked on. For instance, if you invoke it on a `Bus` object then the message must use a specialised symbol for `Bus`. This feature requires you to also update the two sub-types `Bus` and `Car` of `Vehicle`.

   Hint: In Java, you can use Unicode characters as text symbols. These characters are written in the form '\uXXXX' where 'XXXX' are hexadecimal characters. A searchable database of Unicode characters is available at this web site: http://unicode-table.com/en/.

7. Define a two new subtypes of Vehicle: `Motorbike` and `Boat`. Use your practical understandings of these two types to add at least one attribute to each with suitable domain constraints.

Ex2.2. Method Overriding

1. Improve the three classes `Vehicle`, `Bus`, and `Car` from Ex2.1.4 so that each class has a method named `validateRegistrationNumber` (which validates the value of attribute `registrationNumber`), and the two methods of `Bus` and `Car` override that of `Vehicle`.

Ex2.3. Sub-types with extra attributes

1. Update the class `vehiclesextra.Bus` so that it now has the following constraint on the attribute `routes`:

   1.1. `Bus.routes` only contain values in the range [1,100]

2. Update the class `vehiclesextra.Car` so that it now has the following constraint on the attribute `owner`:

   2.1. `Car.owner` must not be more than 255 characters in length

3. Design and code a new sub-type of `Vehicle` called `IronSuit`, which models the iron suit that is worn by Mr. Stark in the movie named Iron Man (https://en.wikipedia.org/wiki/Iron_Man_(2008_film)). Class `IronSuit` must have at least one distinctive attribute and it must be completed with all the necessary operations. One essential operation, for instance, is `fly`, which should carry the person wearing the suit from a point *A* to another point *B* in the air.

In this basic version of the class `IronSuit`, operation `fly` should simply print out a message on the standard console about these flying facts: the two points and the distance between them. You need to decide how to the class `IronSuit` knows about the distances between the pre-defined points.

4. Update the operation `IronSuit.fly` in Ex2.3.3, so that it can simulate on the standard console the actual fly path from point *A* to point *B*. The longer the distance, the longer the path and the travel time. You need to decide how to measure the fly time based on the distance and need to choose a suitable symbol that best represents the path.

   Hint: In Java, you can use the following code to cause a program to pause for a given number of milli-seconds:

```
int millies = 1000; // 1 second
try {
  Thread.sleep(millis);  // pause
  // wakes up: do something (e.g. print a console message)
} catch (InterruptedException e) {
   // no problems: ignore
}
```

Ex2.4. Interface

1. Update class `IronSuit` to implement the interface named `Flyable`. You must create this interface to have the operation named `fly` (introduced in Ex2.3.3) defined in it.

Ex2.5. Multiple implementations

1. Design and code the following type hierarchy, which is used to provide implementations of different sorting algorithms. You must decide on the suitable attributes and operations that need to be defined in the super-type and in each of the sub-types.

   Super type: `SortingAlg`

   Sub-types: `BubleSortAlg`, `QuickSortAlg`, `MergeSortAlg`, …

Ex2.6. Object sorting

`java.lang.Comparable, java.util.Comparator`

Read the Java API about these interfaces and answer the following questions:

1. What are the above interfaces used for?
2. What operation(s) must a subclass implement?
3. Update class `Vehicle` to implement the `Comparable` interface:
   3.1. to compare `Vehicle` objects based their names
4. * How do you design `Vehicle` to support both ASC and DESC sorting orders?

3 Submission

Submit your report to the home work submission box of this tutorial on the FIT portal. Name the file as follows: ***student-id_class*_hwk02.zip**, where ***student-id*** is your student id, ***class*** is the code of the class that you attend.