CSCI 3155: Principles of Programming Languages

Exercise sheet #13

2007

Group name:__________________________

Object-Oriented Programming II

This exercise is largely a lab exercise.
You may use Eclipse (which is preinstalled on our machines) or EMACS/vim on axon.cs.colorado.edu for the programming exercises. Submit the resulting programs via e-mail (to creichen@machine.cs.colorado.edu) or leave them in your home directory on axon, indicating (on the exercise sheet) where you stored them.

Exercise 1. Java's instanceof feature allows us to inspect the dynamic type of an object. Specifically, a instanceof A is true iff $a : T$ and $T <: A$.

(a) Give an example where such a facility might be useful. You may find it helpful to consider instanceof in the proximity of explicit narrowing conversions.

(b) Assume that you only want to test instanceof for one or two classes in your program (though possibly for many objects). Further assume that all of the classes you care about are (direct or indirect) subclasses of C. How could you extend C to achieve an effect similar to instanceof, albeit limited to only one or two types?

Exercise 2. Object-oriented programming has many practical uses. However, one of its most pleasant attributes is that it maps neatly to many aspects of the "physical world".

Illustrate your prowess in Object-Oriented Programming by implementing a text adventure game[1]. The instructor has provided you with the parser part (both as C++ and as Java code), so you can focus on the game content.

You may implement the system either in C++ or in Java. If in doubt, use Java.

The following contains a step-by-step instruction to implement the game. Be advised that the steps become less and less concrete and require you to make more decisions by yourself towards the end.

(a) Set up your development environment to use the skeleton program for the language you picked. Make sure that it compiles. Use the command "quit" to exit.

(b) Examine the source code. You will find a class AdvParser, which parses the input and calls various handler methods (look, go, ...). Create a subclass Game of AdvParser and override the method status to print your own status/welcome message. Make sure to adjust the class Adv to use your newly overridden Game class.

(c) We will populate our game world with *rooms* and *items*. We want to be able to (i) *look* at all three of these (this should print a message describing the object), and (ii) to get the *name* of the object.

    Implement an appropriate inheritance hierarchy that ensures that all three have a common supertype that allows us to *look* at them. Make sure that you use an appropriate "concept" to implement the common supertype and its subtypes.

(d) Rooms should have four exits, to the north, south, east, and west. Implement facilities to set the exits (pointing to other rooms). Implement a single room StartRoom with no exits, and give the room some description. Make sure that the Game object has a reference to the current room (i.e., the StartRoom) and prints the room's name after every command.

    Change the implementation of look to print the room's *description* whenever the string parameter is `null`(Java)/`NULL`(C++).

(e) Implement a second room of your own design and attach it to the StartRoom. Make sure that the exits of the two rooms match up. Try to ensure that your implementation *enforces* that the exits match up.

    Now, implement the method go in class Game that allows you to move around in rooms. Make sure that you can move around among the rooms. Implement this method with a helper method adjacent in Room that tells you what room there is in a given direction.

[1] Or "interactive fiction", as some people prefer to call it.

(f) Let's have a look at items now. Build a class Storage that collects Items. Each object of the class should have a name (initialised by the constructor and retrievable by an appropriate method). The class should have methods for *inserting* and *removing* an item, as well as for printing the names of all items it contains and for *finding* an item by name (if present).

Change the class Item so that it knows which Storage it is contained in. Enforce that each Item is contained in one Storage when it is created.

(g) Ensure that each room has a Storage. Ensure that printing the description of a room will also print the contents of its Storage.

Construct some Item and put it into the Storage of one of the rooms you have constructed. Ensure that everything works.

(h) Allow the player to acquire items, using another Storage object, which we will call the *inventory*.

To implement this, provide a method remove to Item which removes the item from its current Storage and places it in another Storage.

Futher override the Game.get method to give you the desired behaviour.

(i) Ensure that the Game.inventory method prints out the inventory. Further ensure that Game.drop gives you the desired behaviour.

(j) Build a lethal object: if the player tries to pick up the object, the game is over (you can use Adv.loseGame()/Adv::lose_game(void)). Place this object in a new room.

(k) Build a goal room: if the player is inside of the room, the game finishes (you can use Adv.winGame()/Adv::win_game(void)). Connect the goal room to one of your rooms to test it.

(l) Now build a simple puzzle. Build a *salt* object. Build an *ice room* that behaves like regular room, except that one exit is inaccessible unless the player drops the salt in the room (make sure to hide the salt in another room).

Implement this by overriding the method adjacent in your ice room. Note that you can re-use the adjacent implementation for regular Rooms within your overridden implementation.

Ensure that the player must pass the ice room to get to the goal room.

(m) Implement the use method in your Game class. The method should simply invoke a method of the particular item that the player is trying to use. Assume that none of the items so far are useable. Ensure that the player can only use items that are part of the inventory.

(n) Now build another puzzle. Disconnect the goal room from the rest of the game, and instead introduce a *portal room*. (To make the puzzle more reasonable, you might want to add some description to the room that indicates that there is a sealed door/gate/portal on one side of the room).

Introduce a *wand* that the player can find in one of the rooms (not the goal room). The wand should be useable: when the player uses it, it should interactively ask the user for something that it should affect. The user should input a name, leading to an appropriate effect:

i) When applied to the *portal room*, the wand should connect the portal room to the goal room.
ii) When applied to the *wand*, the wand should make itself disappear (making the game potentially unwinnable).
iii) Feel free to add further effects. In particular, ensure that it is easy to add further effects.

Since this is a "magic wand", you don't need to be near the object you affect. Initialise the wand with an array of all the things it can affect. Do not use narrowing conversions. Do not add any new methods (abstract or concrete) to unrelated objects (such as Item).