Comp 401 - Assignment 6: Basic Inheritance and Completing the Bridge Scene

Early Submission Date: Wed Oct 4, 2017 (+5%)

Completion Date: Tue Oct 24, 2017

First Late Date: Fri Oct 27, 2017 (-10%)

Second Late Date: Tue Oct 31, 2017 (-25%)

You will be adding new properties and public methods to the scene. The properties will introduce the bridge, gorge, and two standing areas we saw in my demo. The methods will manipulate the avatars in the scene – the guard and the knights. Each of them will determine which avatar to operate on and then invoke a method on the avatar or one of its components. Thus, you will get much more practice with object composition.

In addition, you will refactor your token classes to use inheritance.

If you have not done so, you may want to also look at my demos at the start of class for what a scene could look like and where we are going:

Bridge Scene - 1st day (long)

Bridge Scene - 2nd day (short)

Inheritance and Collections	PowerPoint PDF YouTube Mix	Docx PDF Drive		Completing the Bridge Scene Checks File	lectures.inheritance Package Git (Inheritance)
Inheritance and variables	PowerPoint PDF YouTube	Docx PDF Drive			lectures.inheritance Package Git (Inheritance)

	Mix				

Gorge with Bridge in the Scene

Add to the scene of your previous assignment a gorge with a bridge. The previous assignment gave you enough experience with object composition, so it is up to you how you add them to your scene. The simplest approach is to create two parallel lines to simulate a gorge, and a rectangle connected to the two lines to simulate a bridge. You are free to give 3-D effects as I have tried to do, or create images for the gorge and/or bridge. For this part, you can add a property called *Gorge* that has a getter for an object that represents a gorge with bridge. It can be an atomic shape such as an image or a composite (bean) shape.

Standing Area Shapes in the Scene

Now add to the scene two additional readonly shape properties, called *KnightArea* and *GuardArea*, defining the areas in which the guard and a knight stand when they communicate with each other. The simplest approach is to make each of these areas a circle, as I have done. You are free to create more sophisticated shapes. There is one standing area for all knights, and a separate one for the guard. The two areas should be on the left side of the bridge and gorge.

Initialization

Write code that is in or called from your constructor(s) that places (a) the knights on the left side of the gorge, with no knight in a standing area, and (b) the guard in the guard standing area. An avatar is in a standing area if at least some part of the avatar in the area. Ideally, both legs of the avatar should be in the standing area, but you may use some image that does not allow that to happen in an aesthetic way

Approach Scene Method (Name: approach)

Implement a public method that takes as an argument an avatar shape and moves the avatar to the knight standing area. When a knight approaches, the knight standing area becomes *occupied*. We do not want two knights to approach simultaneously – so this method should do nothing if the knight area is already occupied.

Name (not tag) this method as: "*approach*".

Occupied Boolean Property (Tag: Occupied)

To help you organize and debug your code, create a Boolean read only property, called *Occupied*, that is true if the knight standing area is occupied. The value of this property

depends on whether a knight has approached and if an approached knight has passed or failed.

Even though Bean conventions allow the getter of a Boolean property P to be called isP (e.g. is isOccupied), to make our grading easier, use the rule of calling it getP (e.g. getOccupied), which is also accepted by Bean conventions.

Say Scene Method (Name: say)

This public method takes a String as an argument. If the knight standing area is not occupied, it does nothing. If the area is occupied, then this method allows the guard and knight to alternate in speaking, with the guard starting the conversation. Thus, the first time it is called after a new knight approaches, it makes the guard say the string; the second time it is called, it makes the knight say something, and so on. Do not put a requirement on the number of questions that have to be asked before a pass or fail. It can be 0 or 4, for example.

Of course, an avatar says a string by setting the text property of its string shape to the string.

Name it, "say".

Knight Turn Boolean Property

Again, to help you organize and debug your code, create a Boolean read only property, called *KnightTurn*, that is true if the knight standing is area is occupied and it is the knight's turn to speak, otherwise it is false.

Interacting Knight Avatar Property

Again, to help you organize and debug your code, create a read only property, called *InteractingKnight*, that returns null if the standing area is not occupied and the knight in the standing area if it is occupied. *Put the @Visible(false) annotation on top of the getter for this method – otherwise we create a DAG.* Do not include it in the PropertyNames annotation.

Passed Scene Method

If the knight standing area is occupied, this method moves the approached knight to some point on the right side of the gorge, beyond the bridge, and makes the standing area unoccupied. The method takes no parameters. A knight can be passed only if it is the guard's turn to speak. Again, it is up to you if you want at most three questions.

Name this method as: "*passed*"

Failed Scene Method

This method puts either the knight or the guard to some point "in the gorge." If the knight standing area is occupied and it is the (a) guard's turn to speak, then the **knight** fails and falls in gorge (b) knight's turn to speak, then the **guard** fails. When an avatar falls, it is neither on the left or right side of the bridge/gorge. Again, the method takes no parameters, and it is up to you if you want at most three questions. When a knight fails, the standing area becomes unoccupied.

Name this method as: "*failed*"

Refactoring Token Code

Refactoring of code is re-implementation of the code without changing its functionality. Refactor your token classes and interfaces to follow the inheritance rules given below.

1. Remove code repetition in both the classes and interfaces through inheritance.
2. Make each class implement a single interface, which can be an extension of multiple interfaces. Each class can of course implement a different interface – the idea here is to replace all interfaces implemented by a class by a single interface, which can be a new interface that extends all of the previous multiple interfaces. Thus, the word class can implement a single interface that extends the common token interface.

A class should duplicate the annotations (*other than tag*) of its super class if they still apply to it.

When refactoring you will need to delete redundant variables and methods in your original code that are now inherited from the superclass. Be sure you do not forget to do so. Forgetting to delete variables will cause subtle bugs that can be caught by turning on the following check in Eclipse:

Window→Preferences→Compiler→Error/Warnings→Name shadowing and conflicts→Field declaration hides another field or variable.

You may want to turn on error checking for this, but make sure you turn on warning.

To put some structure in this part of your assignment: implement a class tagged "Token" that implements the common interface, and make all of the other token classes direct or indirect subclasses of this class. Similarly, the command token classes will be subclasses of the word class and will simply define a constructor that calls the superclass constructor.

We will be refactoring composite and atomic shape classes in the next assignment.

Extra Credit.

1. Allow the whole scene to be scrolled left, right, up, and down, defining a method with two int arguments specifying how much it should be scrolled in the x and y directions respectively. Positive (negative) values of the parameter indicate how much it should scroll right/down (left/up). The scrolling unit is up to you. Tag the method as *"scroll"*. If the scene scrolls in the *positive* (negative) x/y direction, the scene components move in the *negative* (positive) direction.
2. Change the StringHistory and AStringHistory (from lectures) to store tokens rather than strings. Then extend it (and its interface) to support the clear operation (tagged "clear"), which empties the list. Tag the extended class as *"ClearableHistory"*. In your scanner bean, create an *additional* readonly property, *TokenList*, which returns a clearable history rather than an array. You will clear the history every time a string is scanned and add the scanned tokens to it. This is more efficient than the array case, in which you create a new compact array each time a new string is scanned. This means that you will create two *copies* of every token object- one to store in the array, and one to store in the history. Do not put the same token object in the array and list, as that would create a DAG. Use *StructurePatternNames.VECTOR_PATTERN* as the structure pattern for the history classes as AStringHistory follows the naming conventions of the Java class, Vector.

For Fun

1. Rotate or give other effects to an avatar as it falls into the gorge.
2. Animate the movement of a passed avatar as it crosses the bridge.
3. Animate the movement of an approaching Knight.
4. Give 3-D effects to the bridge and/or gorge.

For 1-3, define an init method in the scene that takes an OEFrame as an argument.This is like a setter except that it is expected to be called only once, to initialize the OEFrame, and does not have an associated getter. Do not pass the frame as an argument to the approach, passed and failed methods, which will do the animation. Later you will be removing this method, when we learn better ways to refresh.

Animating/Demoing Main Class

1. To demonstrate your work, write a main class that instantiates the bridge scene, displays it in an ObjectEditor window, and shows each of the scene methods you have implemented working. If you animate the movement of a failed or passed Knight (for fun), the animations will only be visible if you call the scene methods from the main program. In general, the screen will freeze if you call animating code from ObjectEditor. We will see the reason for this later, as well as a solution to this problem. When you display the scene, make sure the two Boolean

properties are displayed – either in the tree view or the main panel. You may have to move the dividers between the panels to see the information properly - the TAs will also do this when they view the data. You can use the setSize() method on an OEFrame to adjust its size.

2. In addition, your main class should redo the scanner animation from assignment 4 to show that your code still works after inheritance.

Constraints

1. Follow the inheritance constraints placed on the token types.
2. If a class declares (without inheriting) all the methods of an interface, then the interface should have the same tag as the class. This is why in this assignment we have a class and interface with the tag "Token".
3. A class or interface should not have more than one tag.
4. Do not tag a subtype with the tags of its super type(s). This rule applies to both classes, which have unique supertypes, and interfaces, which can have multiple super types.
5. Do not tag a supertype with the tags if its subtypes – for instance the interface tagged Token should not have the tag "Word".
6. Be sure to tag your new methods, add them to the scene interface, and put the new property names in the appropriate annotations.
7. Do not use System.exit()..
8. Make your main classes public - otherwise the grader cannot access them.
9. Make sure your image files are included with the submission – they should be in the project directory.
10. Make sure your project is self-contained and does not reference other projects such as recitation and previous assignment submissions.
11. In this and the previous assignment, you will be using several int constants. No magic numbers as discussed in class lecture on style.
12. You also have to deal with Booleans in this assignment. For those using CheckStyle, pay heed to the warnings indicating Boolean expressions are more complicated than necessary – you will not lose points for not doing so, however.
13. When the TAs are grading, there are two user-interfaces, one created by your program and the other is a grader control panel to navigate among your projects and manually fill in data and press buttons. Often your UI covers the grader UI, making it difficult for the TAs. Try and create as small a window as is necessary

to demo and position it in in the upper left corner. Also if you do not need to show the main or tree panel (in this assignment you do need to show one of these), hide them. The user-interface notes tell you how do this. In addition, any textual or graphical property you do not want displayed in the main/drawing panel can be hidden by saying @Visible(false) before its getter.

Submission Instructions

- These are the same is in the previous assignment. The TAs will run the main method to see the test cases animate.
- Be sure to follow the conventions for the name and package of the main class so that its full name is: **main.Assignment6**.

Good luck!