Homework 03

Version: 1.6

Due: 2016-10-20T23:59:59

Download zip (hw3.zip)

Introduction

Civilization (https://en.wikipedia.org/wiki/Civilization_(video_game)) is a turn-based strategy game centered around founding and building a civilization. More commonly referred to as Sid Meier's Civilization, the game has been around since 1991 and has since developed somewhat of a cult following. This semester we are going to be implementing our own version of this game for your homework assignments and by the end of the semester you will have a working game you can show off to your friends! If you are not familiar with Civilization, there is a free version (https://en.wikipedia.org/wiki/Freeciv) that you can play in order to get familiar with the game.

This assignment will get you ramped up on Abstract Classes, Interfaces, Data Abstraction, Inheritance, and Polymorphism. This project will:

- Test your ability to convert a written description into a Java program,
- Require you to write superclasses, and subclasses that extend them,
- Assess your understanding of abstract classes and interfaces,
- Give you an opportunity to practice using the concepts of polymorphism.

Problem Description

Now that you've had some practice working on a project with multiple classes, we're ready to take our game up a notch! For this homework, you'll be implementing several classes representing the controllable units and structures of the game as well as some of the logic behind their interactions. And best of all, you're going to using inheritance and your knowledge of polymorphism to do it!

Before we begin…

- Like all homework descriptions, it is very important you read and understand this entire document.
- It is important to note that you will not need to create a new repository for this homework. You will be working in the same git repository you set up for the first homework.
- This homework will be very focused on details. So make sure you read the description carefully and implement every detail described!
- Note that many of the instance variables will require getter and setter methods (since all of your instance data should be private) that we did not explicitly describe in the homework description. If you complete all of the classes correctly then `CivilizationGame.java` should compile and run no problem! If you receive errors about missing methods, you should add that functionality to your class.
- Most of this code is revised from HW02 code, so you will need to replace any duplicated files in your repository with the new versions.

Compiling and Running

- **This project will not compile in its entirety right off the bat**
- As you work, you will need to individually compile the files you write to make sure that they work! So remember to compile often so that you don't have to deal with a mountain of errors! Keep in mind that when you compile only one file, it does not compile some of the other ones that it references. So, you will probably get alot of cannot find symbol

errors. Don't worry about them, they appear because some dependency classes aren't compiled. You may also get warnings that methods aren't being overridden or something if you use the @Overide tag. Once again, that is because the superclass didn't get compiled if you compiled that file only.
- This project makes use of several different packages to manage its many files. As a result, compiling and running this project is a bit different from projects you have worked on in the past.
- Since our files are in many different packages, we have to make sure they get linked together properly. We do this by setting the class path when we compile and when we run.
- To compile the program in its totality run this command:

```
$ javac -cp src/main/java src/main/java/runner/*.java
```

- If that one doesn't work for some reason, and you are using bash/git bash or something like that, try this one:

```
$ javac -cp src/main/java src/main/java/*/*.java
```

- AND IF THAT ONE DOESN'T WORK, this one should work for sure.

```
$ javac -cp src/main/java src/main/java/runner/*.java src/main/java/model/*.java src/main/java/view/*.java src/main/java/controller/*.java
```

- That command instructs java to compile all of our files while considering src/main/java to be our classpath.
- To run the program, you should run:

```
$ java -cp src/main/java runner.CivilizationGame
```

Checkstyle

- Remember that you may lose checkstyle points for this homework! So, make sure that you are frequently running checkstyle on your code to ensure that you don't lost unnecessary points.
- There are currently two checkstyle errors in the homework that you are not responsible for. These are caused by some provided constructors taking in more than 7 parameters. You will not be penalized for these two checkstyle errors.
- When writing MilitaryUnit.java, the constructor will also take in more than 7 parameters. You will not be penalized for this third checkstyle error either.
- **The checkstyle cap on this assignment is 40 points**, where you lose one point per error.

Solution Description

- Note: When making new classes, remember that we are working with multiple packages. You will have to specify what package each file belongs to at the top of the file. Read through some of the provided files to see how to do this. It is also suggested that you look through the files to understand the interplay between all of the different parts.
- Note: You might notice that all of the classes in the model package, with the exception of Model, do not have an access modifier, making them package private. This is by design to prevent access to these classes from outside the package. Any class you create for this project should have package private access.

Civilization.java

First thing's first! In your second homework, you may remember a good deal of overlap between all of the different Civilization classes (Egypt, RomanEmpire, and QinDynasty). To clean this up and eliminate alot of repeated code, you're going to create a class called Civilization to serve as the super class for Egypt, RomanEmpire, and QinDynasty. We've provided some modified files for you (they are a bit different from what they were in HW02, as some requirements have changed) for these three subclasses. Your job here is to factor out common code between them and move it into Civilization.java to make it cleaner. Once you have done this, there is some additional functionality to add. For now, in each of the three civilization subclasses, override the explore method to provide unique functionality for each subclass. Egypt's explore method should find treasure in its desert, and add the amount of gold found to its treasury. The RomanEmpire should mine coal in its hills and add the resources it finds to its own resources. QinDynasty should also explore its hills, but instead go hunting and add the health of the Game it hunted to its food supply (Don't forget to make sure the Game in the hills does not run out! There is an instance method in Hills to help you with this). There are additional methods that these classes will override, but we will revisit those later on.

- Note: Once you are finished, Egypt, RomanEmpire, and QinDynasty should each have one no-args constructor only. Make sure that the name is set properly.
- Note: It should be possible to create an instance of the Civilization class. As a result, you will need to provide some default functionality for any methods that are overridden in subclasses. Namely, the default functionality for the explore method should just be to add 20 resources to the Civilization's resource pool.

MiltaryUnit.java

Take a look at Unit.java. This class represents a player controllable unit in the game. It extends a class called MapObject. Note what methods you have to work with in these classes, how they work, and what they do (it will help to take a look at model/Model.java, view/UI.java, controller/GameController.java, and runner/CivilizationGame to fully understand how the game works). What you are going to do now, is create a new class called MilitaryUnit that inherits from Unit and represents a class that can attack another MapObject on the game board.

- You should not be able to make an instance of the MilitaryUnit class.
- The class should have a private field called damage, as well as associated accessor methods (assume the normal naming convention for these).
- MilitaryUnit should have one public constructor that takes in an int health, a Civilization object that will serve as its owner, an int baseEndurance, an int pay, an int initialGoldCost, an int initialFoodCost, an int initialResourceCost, and an int damage (in that order to make it work nicely with provided classes). The initialHappinessCost of a MilitaryUnit should be 10. Set the instance data for MilitaryUnit, and remember to make sure that all instance data for superclass gets set as well.
- MilitaryUnit should override the tick method. The method should call the superclass's version of the method, and then set its canAttack boolean equal to true (keep in mind what fields the superclass has!).
- MilitaryUnit should have a method called battle that returns nothing, and takes in a MapObject as a parameter. This method should not have an implementation defined in this class.
- MilitaryUnit should have a method called attack that returns nothing, and takes in a MapObject as a parameter. Whenver a MilitaryUnit battles, it should increase its owner Civilization's strategy level appropriately (recall what methods are available to increase strategy that would make sense during a **battle**), call the MilitaryUnit's battle method, and set the Unit's canAttack boolean equal to false, so that they cannot attack more than once during a turn.
- Finally, MilitaryUnit should **correctly** override the toString method. The toString method should return "Military Unit. " appended to the front of the superclass's toString method.

Symbolizable.java

- Create an interface called Symbolizable. This interface will have one method called symbol. Symbol simply returns a char representing the symbol that represents something on the map.
- Modify the header for MapObject such that it will implement this new interface. Do not provide an implementation for symbol in this class; implementations will be provided at the lowest level.

MilitaryUnit subclasses

Now, we're going to create some concrete subclasses for the MiltaryUnit we just made! The ones you will implement will be in MeleeUnit.java, RangedUnit.java, and SiegeUnit.java. HybridUnit.java has been provided for you, and you should not have to modify it.

- Each of these subclasses should **correctly** override the toString method from MilitaryUnit. The toString should be the name of the unit appended to the front of the toString for the super class. E.g, for MeleeUnit it should be "Melee Unit. " appended to the beginning of the superclass's toString.
- Each of these subclasses should have only one constructor that takes in a Civilization that will represent the owner of the unit. All other values will be default, as described in the individual sections below for each class.
- Each of these subclasses should provide an implementation for the battle method. A battle step consists of the unit attacking whatever MapObject got passed in (i.e, dealing damage to it, recall the methods that a MapObject has that will let you implement this), and then possibly being counterattacked by the enemy unit. Firstly, the enemy unit will only be able to counterattack if it survived the damage it was dealt in the first step. Additional conditions for the enemy unit counterattacking are detailed on a per subclass basis as described in the below sections. SiegeUnit has some special requirements that will be detailed in its section.

MeleeUnit.java

- health: 100
- baseEndurance: 10
- pay: 10
- initialGoldCost: 14
- initialFoodCost: 5
- initialResourceCost: 0
- damage: 30
- Should return the following symbol: M
- Can only be counterattacked if the enemy is a MeleeUnit or if the enemy is a HybridUnit.

Special MeleeUnit subclass: LegionUnit.java

- This is a special MeleeUnit that the RomanEmpire class uses in place of the normal MeleeUnit (RomanEmpire should override a method from Civilization to reflect this).
- Legion units are particularly strong, so when they go to battle, HybridUnits are unable to counterattack them.
- Legion units also have 1.5 times as much damage as a normal MeleeUnit (Truncate because damage is stored as an integer).
- Should return the symbol: L
- Should override the toString method from MeleeUnit, and return "Legion. " appended to MeleeUnit's toString.
- Should have only one constructor that takes in a Civilization that represents the owner.

RangedUnit.java

- health: 100
- baseEndurance: 10
- pay: 10
- initialGoldCost: 14
- initialFoodCost: 5
- initialResourceCost: 0
- damage: 30
- Should return the following symbol: R
- Can only be counterattacked if the enemy is a RangedUnit or if the enemy is a HybridUnit.

Special RangedUnit subclass: WarChariot.java

- This is a special RangedUnit that the Egypt class uses in place of the normal RangedUnit (Egypt should override a method from Civilization to reflect this).
- WarChariot units have twice the baseEndurance of a normal RangedUnit.
- Should return the symbol: W
- Should override the toString method from RangedUnit, and return “War Chariot Unit. “ appended to RangedUnit’s toString.
- Should have only one constructor that takes in a Civilization that represents the owner.

SiegeUnit.java

- health: 200
- baseEndurance: 5
- pay: 10
- initialGoldCost: 14
- initialFoodCost: 5
- initialResourceCost: 10
- damage: 60
- Should return the following symbol: S
- SiegeUnit cannot be counterattacked.
- SiegeUnit will only deal damage to Buildings, and cannot deal damage to anything else.
- Recall that in the attack method for MilitaryUnit, we increase the Civilization’s strategy points as if we were engaging in a battle. However, if a SiegeUnit attacks, we’re in a siege! You should override the attack method to reflect this as well.

Special SiegeUnit subclass: BlackPowderUnit.java

- This is a special SiegeUnit that the QinDynasty class uses in place of the normal SiegeUnit (QinDynasty should override a method from Civilization to reflect this).
- BlackPowderUnits are much more robust than a traditonal SiegeUnit, and have no restrictions on whom they can attack, and can still not be counterattacked.
- Should return the symbol: B
- Should override the toString method from MeleeUnit, and return “Black Powder Unit. “ appended to SiegeUnit’s toString.
- Should have only one constructor that takes in a Civilization that represents the owner.

Building.java and some subclasses

Now that we have all of our trusty defenders implemented, we can safely work on building up some of our civilization’s Buildings! Building.java is provided for you; take a look at that to see what all of our Buildings are going to be able to do, and what it is that they have. Notice that this is an abstract class. This is because you will be extending this class to make some specialized Buildings with additional functionality! The classes you will be implementing will be Farm.java and Landmark.java. FishingShack.java, CoalMine.java, and Settlement.java have been provided for you, but we urge you to take a look at them to stay up to date on changes that have been made to those classes. Both of the subclasses you need to implement will have a one-arg constructor that takes in a Civilization that represents the owner of the Building. Additional values you need to set will be detailed in the sections below for each class. Both classes will also have to provide implementations for the invest method, and provide toStrings. The format of the toString for each class will be the name of the Building, appended to the front of the superclass’s toString. So, Farm would have “Farm. “ appended to the front of the superclass’s toString.

Farm.java

- health: 200
- goldGeneration: 0
- resourceGeneration: 0
- foodGeneration: 10

- techPointGeneration: 0
- philosophyGeneration: 0
- happinessGeneration: 10
- Should return the symbol: +
- The invest method for this class should permanently increase its foodGeneration by 2.

Landmark.java

- health: 200
- goldGeneration: 0
- resourceGeneration: 0
- foodGeneration: 0
- techPointGeneration: 10
- philosophyGeneration: 0
- happinessGeneration: 10
- Should return the symbol: !
- The invest method for this class should permanently increase its techPointGeneration by 5.
- Landmark is also going to have some subclasses. Each of these subclasses will have a single one arg constructor that takes in a Civilization representing the Landmark's owner. They are also all going to have a toString that returns the name of the Landmark appended to the beginning of the superclass's toString. For example, the Pyramid toString would return "Pyramid. " appended to the beginning of the super class's toString.

Special Landmark subclass: Pyramid.java

- Pyramid is a special Landmark used only by Egypt in place of the basic Landmark (Egypt should override a method from Civilization to reflect this!).
- Investing in a Pyramid, while also causing the investment effects of Landmark, will additionally provide a one time philosophy boost of 25 to its Civilization.

Special Landmark subclass: GreatWall.java

- GreatWall is a special Landmark used only by the QinDynasty in place of the basic Landmark (QinDynasty should override a method from Civilization to reflect this!).
- Investing in a GreatWall, while also causing the investment effects of Landmark, will additionally provide a one time strategy boost of 10 to its Civilization.

Special Landmark subclass: Coliseum.java

- Coliseum is a special Landmark used only by the RomanEmpire in place of the basic Landmark (RomanEmpire should override a method from Civilization to reflect this!).
- Investing in a Coliseum, while also causing the investment effects of Landmark, will additionally provide a one time happiness boost of 50 to its Civilization.

Worker unit classes

- At this point we've got our Buildings all set up, but we have nobody to actually build them! Never fear! We have our humble workers to do this work for us! Each of the Worker classes you implement will inherit from unit, and also have the qualities of the Convertable interface, which has been provided for you. The convertable interface has two methods. The first of them, called convert, simply returns the owner's version of the Building that they are going to build. canConvert represents whether or not they can build on the TileType that was passed in. Workers are pretty similar, so when writing their constructors, you can make use of the one arg constructor in Unit. However, they will all have different toStrings. Each of these subclasses should have a toString that returns the name of the unit appended to the beginning of the super class's toString. For example, AnglerUnit would return a value like "Anglers. " appended to the beginning of the super class's toString. Another important thing to note is that these worker units can only build on specific terrains, so you should familiarize yourself with the provided TileType enum. Also keep in mind that Civilizations have methods that return what kind of Building they use.

AnglerUnit.java

- Should return its owner's version of FishingShack.
- Can only build on WATER tiles.
- Should return the symbol: a

CoalMinerUnit.java

- Should return its owner's version of CoalMine.
- Can only build on HILLS tiles.
- Should return the symbol: c

FarmerUnit.java

- Should return its owner's version of Farm.
- Can only build on PLAINS tiles.
- Should return the symbol: f

MasterBuilderUnit.java

- Should return its owner's version of Landmark.
- Can only build on PLAINS tiles.
- Should return the symbol: m

SettlerUnit.java

- Should have an instance variable townName of type String, which stores the name of the town these guys want to settle.
- This class's only constructor should take in a Civilization representing the owner, as well as a String representing the town name, and the instance variable should be set appropriately.
- Should return its owner's version of Settlement.
- Whenever this unit converts, it should increment its owner's number of settlements.
- Can only build on PLAINS tiles.
- Should return the symbol: s
- This toString should, return something like "Settlers of INSERTTOWNNAMEHERE. " appended to the beginning of the super class's toString.

Model.java

- At this point, you should be able to compile your Game, and it should be relatively playable. However, there is one last class that requires some implementation. While most of Model.java has been provided for you, attackSelected has been left blank (it currently returns false; this is only temporary, and you will provide the correct return values). Some important things for you to note while you are working on this method are that the TerrainTile selected variable in Model is the tile on the map that the user has currently selected, and that the Map class has some methods you can make use of. Model has an instance of Map called map.

attackSelected

- Takes in two ints, r and c, which represent the row and column on the map of the MapObject the user is currently trying to attack. It is possible that there is no MapObject at these coordinates.
- select contains the TerrainTile holding the MapObject that the user is trying to attack with. This MapObject being a MilitaryUnit is a precondition to this method being called.
- The attack should only go through if there is a valid MapObject at (r, c) on the map, the owner of this MapObject is not the playerCivilization, and the player's unit is able to attack this turn.
- After the unit attacks, if any of the MapObjects were destroyed during the battle, they should be removed from the map (i.e the TerrainTiles they resided on should no longer have occupants).
- If the attack was successful (i.e it went through as detailed above), the method should return true, otherwise it should return false.

Conclusion

- If everything worked out right, you should now be able to compile and play your game! We'll be iterating on this in future homeworks, so make sure you become familiar with how it works, and all of the different moving parts.

Verifying Your Submission

Please be sure that any code you push compiles and runs through the command line! Pull from your repository and make sure everything is working how you want it!