Eric Myre

Alex Nuccio

Angel Cornejo

1. Design choices
   1. Our choices within our game were made because they were the most sensible and logical choices we could make: We didn’t do anything the hard way unless we had to. Our most notable example is how we use items: we technically don’t. Item, and its subclasses HealthPotion, SpeedShoes, StrengthPotion, and Trap all do not implement a direction based Use() like we originally planned. Instead, they now are field-less classes whose type is used via instance of comparisons within the Unit’s Use() to determine how the shall be used. We figured this was much easier than having bounds checking and occupancy and the like within the Use() method, because then it wouldn’t be a simple handful of lines, and the more complicated the code, the harder it is to follow.
   2. Now, for an example of where there was no other option: the controller. Our controller is long, hard, and not too pretty to look at. The only reason we have for why this java file is so overwhelmingly large is because there was no other way. All of our prompts, our GUI, our popups must be made in the same file, or else we’d lose track of where edits have to go. There is a method to the madness, and on a good day we could tell you what a given part is doing without context clues, but that was the way it had to be.
   3. Lastly, most of the rest of our code was designed to be the most understandable as possible. Maps have 2D-arrays of Tiles, units have HP, the water is blue and the grass is green. We wanted the code to be as understandable as possible, so we only have to explain a few things here or there. I can’t think if there is anything else weird within our code, maybe that says something about our final masterpiece.
2. Patterns
   1. Builder Pattern – Our prompt for adding units, map, and scenario to the board is an example of the Builder Pattern. We set up the map by using a series of prompts from the controller, and each entry into the Java Swing drop down menus responds to values within the game itself. For example if you selected the basic game-mode, then you can choose your team by selecting them one unit at a time, and at each stage the map gets updated. It starts out as a basic map, depending on the scenario it may be changed to a different map type, then you may choose which map you want if you are on a basic mode, then the units are added to each player and the players are added to the game all before the game starts.
   2. Strategy – As per spec requirements, we have multiple strategy levels, depending on your preference. We have your basic Strategy, which just attempts to attack in all directions then moves in random directions if it could not attack, and for those who simply want to enjoy slaughtering defenseless lambs or for people who really want to hate themselves, there is an even easier Practice mode. Practice mode means the enemy units will never attack, just move. Lazy, or genius? With the time delimiter on matches killing an entirely fast all Horse enemy army is harder than it sounds, because on regular difficulty they’d stay still and attack your units, now they are fast little lemmings.
   3. Template – Our Unit class takes part in a template patter. The abstract class Unit, from which every unit is derived, has basic concrete methods that do not change for any subclass, and concrete methods that are true for all but a couple subclasses, so the others simply override the original method while still using the other templates. The figurehead example is of course Attack(), which targets a unit one square away. For Archers and Mages however, their attacks are more complicated, which means they have different implementation both of which are totally unique, and both of which override the Unit Attack method in favor of their clearly superior strengths.
   4. Singleton – The Map used throughout the game is static. It does not change. We never call the constructor again to make a new one, we just pass a reference to it around whenever it is needed. There is only one.
   5. MVC – Probably the most given of all of the required six design patters is the MVC. We have lots of code under the hood in the model, and our controller also creates the view. I’d honestly be surprised if someone was able to make a GUI without an MVC, it’s practically how we learned to do it.
   6. Composite – Our Swing based GUI has JFrames stored within JFrames, so we could store stuff while we stored stuff. The JFrames both contain things that are either other containers, MORE JFRAMES, or of course useful things like JTextFields or JLabels.
3. Extra features

Our most notable extra feature is the Mage unit. The mage is a ranged unit that is not only able a unit that is two tiles away from it, but when it successfully attacks an enemy, it does magical splash damage, hurting every unit that is standing on the adjacent tiles of the damaged unit. This splash damage required a large amount of additions to the basic ranged attack method from the Archer class, essentially tripling the lines used… Besides our amazing fireballing wizard and our beautifully hand drawn masterpieces, there is the fact that we got all of this code working with a solid THREE members. We had the unfortunate pleasure of having our fourth team-member leave before we so much as started writing our code, so all of our work has been done thanks to this legendary trio.