**World of Heros Design Document**

Xu Song, Newcomb Maddux, Ian Cowan

**Purpose**

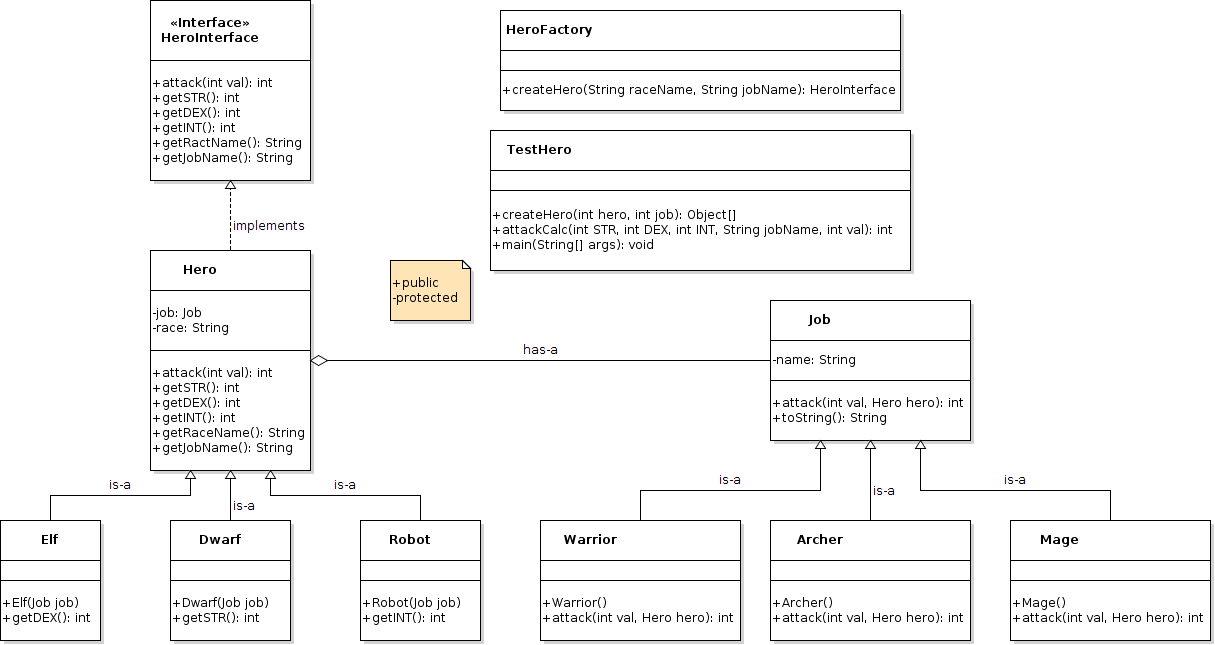
Our team is working on a project for the production of World Of Heroes. The core designers need a way to create heroes that have a job and kill off the top level predators in the ecosystem. They need a simple way to create the hero and get specific information from the hero based on the different combinations of the races and jobs. This needs to be scalable as well because the core designers are planning on adding more races and jobs down the road.

**Specifications**

The designers want to be able to create an object from the HeroInterface which they have provided to us, including the methods they want to be able to call from the object. In order to create the hero object, the core designers want to be able to call HeroFactory.createHero() along with two strings that include the race name and the job name and they want it to return a HeroInterface object. All of the heroes should begin with 20 strength, 20 dexterity, and 20 intelligence. Elves should increase the dexterity by 5, dwarves should increase the strength by 5, and robots should increase their intelligence by 5. Each job, warrior, archer, and mage, have different attack amounts that are defined by the core designers. When creating a hero, if the race or job are not valid the HeroFactory.createHero() should just return null.

Outside of HeroFactory and our unit tests, no if or switch statement are permitted. The code needs to be scalable and have the ability to increase the number of both races and jobs in the future without too much extra work. Finally, numeric data is not allowed to be stored in any of the classes. All of these requirements are in place to increase the efficiency of the program.

**Design Overview**



**Figure 1. Class Diagram for Hero Creation**

In order to create a HeroInterface object, we will create a method within HeroFactory, called createHero(), that creates a hero based on a provided race and job name. This HeroFactory will then first create a job from any of the three jobs. If creation of the job fails, the method will return null at this point. If a job gets created successfully, the factory will continue and create a hero from one of the three hero races. For the methods within Hero, the getDEX(), getSTR(), and getINT() methods will return 20 by default, and are morphed as necessary within each individual hero class to return the appropriate value, if different from 20. The getRaceName() method simply returns the raceName that is set upon creation of the specific hero object. The getJobName() method returns from the created job and returns the job’s getString() method, as created appropriately. Finally the attack() method that takes an integer value doesn’t do anything within the Hero class other than returning the value that’s calculated from the attack() method found within the appropriate Job class. The attack() method that takes an integer value within the appropriate Job class calculates the attack value of a hero dependent on the race’s strength, dexterity, and intelligence values and the value provided by the user. It then calculates the appropriate attack amount and returns an integer value. Finally, all of our unit tests are within the TestHero class and these will run for every combination of race and job combinations.

**Analysis**

An alternative way to create the heroes and jobs would be to create each hero and job as its own class. If all of the heroes and jobs were created as individual classes there would be large amounts of repeated code since there would be no inherited methods from a superclass. The attack(), getSTR(), getDEX(), getINT(), getRaceName(), and getJobName() methods would all have to be copied into each hero and the attack() and toString() methods would need to be copied for all jobs. By following this process, if more heroes or jobs needed to be created all of these methods would need to be rewritten instead of inherited which would waste time.

Another alternative way would have been to create Hero and Job as interfaces. All classes that implement these Hero and Job interfaces would need to use the methods within the interface. Having to use every method in the interface would be unnecessary because not every Hero type or Job type needs every method that would be in the interface.

Our implementation of the heros is linearly scalable, not exponentially, that is, for each job or race that we want to add, we only have to create one new class instead of more than one. For time complexity, all of our classes, except for TestHero, will run with constant time, . For the TestHero class, the time complexity will be non-linear, , which depends on the number of races and jobs.

It is unlikely that one would encounter any risks when utilizing our implementation unless there was a syntax error or an incorrectly assigned variable inherited by subclasses, which could ruin the returns of the observer functions.

**Conclusion**

In this design document, we lay out the purpose of the project and then describe the overall layout of out UML diagram including relationships of classes and objects and how each method is implemented. We also compare and contrast benefits, risks and time complexities of alternative designs.

By completing this lab, we have a better understanding of how a project is done. The designing part is the most important part. We need to choose the most efficient way (the time complexity is O(1)) among all the possibilities to implement objects and various relationships between classes. We also practiced using polymorphism, complements, composition, and inheritance to create multiple hero objects that are required. These objects can be created simply by passing Job object as a parameter into Hero object; they implement all the methods provided in the interface and have distinct attributes and attacking calculation methods According to specific races and jobs. Later on, if the game company creates more characters, all we need to do is add subclasses of new races and jobs, there is no need to create classes for each combination.

To sum up, our design can be considered as successful; it passed the unit testing and fulfills all the requirements that are asked.