Modeling and Programming 2018-2: First lab practice

Luis Daniel Aragon Bermudez 416041271

February 26th, 2017

Contents

Strategy Design Pattern

According to Elisabeth Freeman, Kathy Sierra, the strategy design pattern:

> defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Implementing this pattern is extremely simple in Java; we only need to define an interface for our desired algorithm family and then make the class we want to exhibit this behaviour have a field of this type, this way the class can use any of the interface's implementing classes dynamically to exhibit different behaviors depending on which of the implementing classes is currently "in use".

The application

This application demos a simple implementation of the strategy design pattern.

The demo's specification is as follows:

> You have been commissioned to make an adventure video game, in the game you have a character that can be a king, a queen, a troll or a knight, all the characters have the activities move and attack, the movements are the same for everyone, but the attack depends on the weapon they have, each character can use only one weapon at a time but they can change weapons at any time of the game. The weapons they can use are: axe, knife, hammer and sword.

The project's file tree looks like this:

```
src/main/java/
  mx/
    unam/
      fciencias/
        myp/
          Demo.java
          models/
            characters/
              GameCharacter.java
              King.java
              Knight.java
              Queen.java
```

```
    Troll.java
    package-info.java
  weapons/
    Axe.java
    Hammer.java
    Knife.java
    Sword.java
    Weapon.java
    package-info.java
package-info.java
utils/
  Direction.java
  GameVector.java
  Position.java
  package-info.java
```

Regarding the Strategy design pattern, we wanted all our `GameCharacter`s to change their attacks depending on which weapon was equipped, so we gave `GameCharacter` a `Weapon` field and made `GameCharacter`'s attack method call the attack method on its `Weapon`; then the process of adding weapons was really easy each would implement the Weapon interface and override the attack method in order to get the desired custom behavior. It was also necessary to add the `swapWeapon` method to `GameCharacter` in order to make its attack capabilities dynamic and customizable.

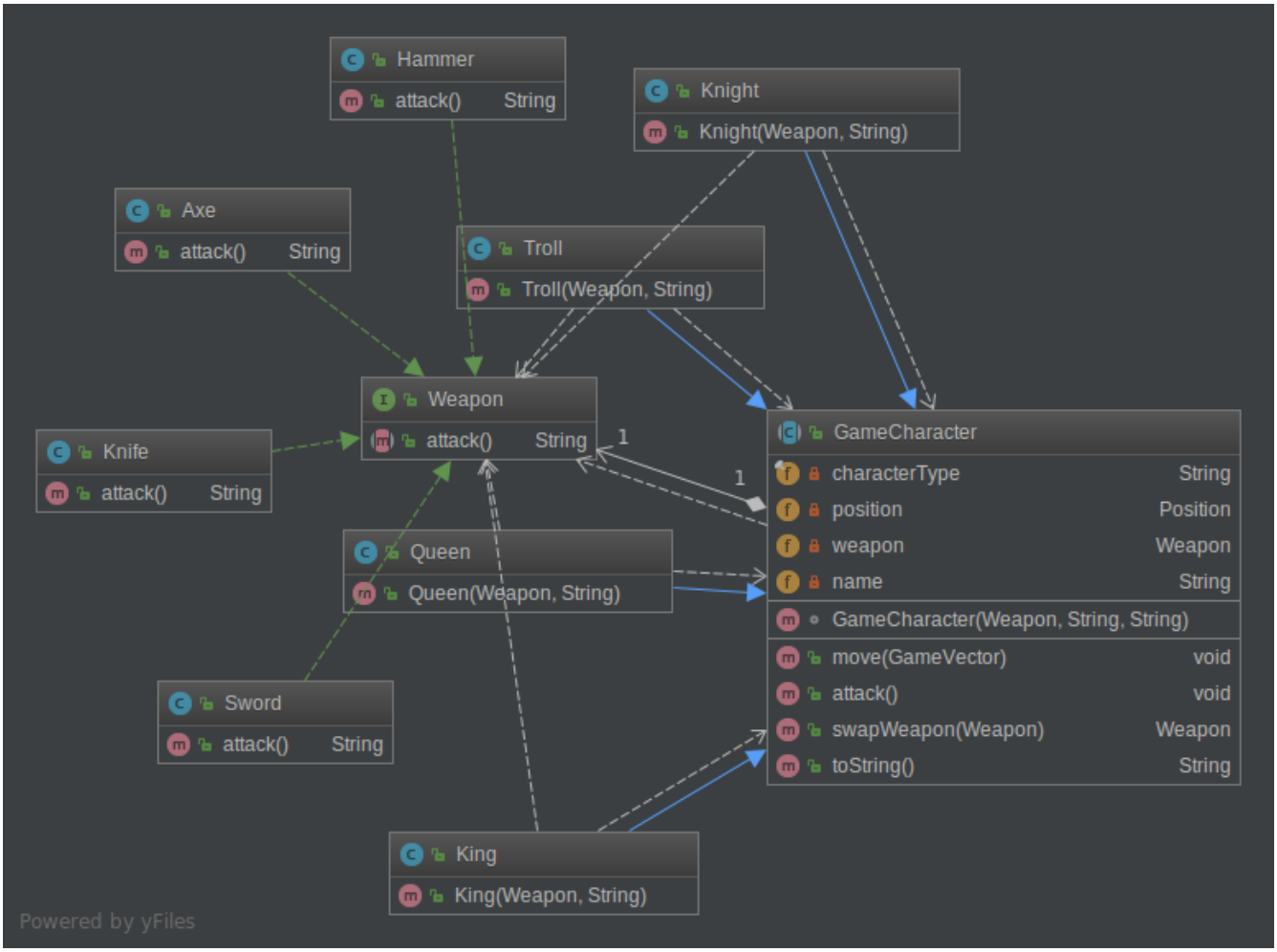

Figure 1: UML diagram showing the details of the Strategy Design pattern implementation.

Building and running the program

The program can be built using gradle, the most common tasks are described bellow, for a full list of available tasks use `./gradlew tasks`. If you're on Linux or Mac then running the following command from the project's main directory will be enough to build and run the program: `./gradlew run`. If you're on windows use `gradlew.bat run` from the command prompt instead.

Some of the most common tasks are:

1. `./gradlew build`, compiles and creates the outputs of this project.
2. `./gradlew javadoc`, generates the program's documentation and puts it inside `build/docs/javadoc`.
3. `./gradlew run`, builds the program and runs the application.
4. `./gradlew clean`, deletes all files and folders generated during the build process (except the .gradle directory).

Acknowledgements

For more information on the tools used to build, create and run this program refer to the following links:

- Pandoc is a Haskell library for converting from one markup format to another, and a command-line tool that uses this library. Pandoc was used to keep this README file consistent.
- Gradle was used to create the build script.
- JetBrains' IntelliJ IDEA was used as the primary editor. Its diagramming utility also came in handy to produce the application's class diagram.