Bilkent University

CS 319 Spring 2020

Iteration 2 - Final Report

Group 1B-SS

Can Cebeci	21703376
Cem Cebeci	21703377
Gizem Karal	21703094
Gökberk Boz	21602558
Sena Sultan Karataş	21604078
Mehmet Ali Altunsoy	21702531

1 Introduction

We started implementing the game, after our iteration one design report had been submitted. IntelliJ IDE environment is used by the developers for the programming of the game. For the GUI implementation, we used javaFX library. In order to commit the changes for the development, github is used.

The user interface is implemented with the music in the background. The game is a single-player game consisting of enemies created according to the different levels of combat vertices. Three characters are implemented as: Ironclad is a warrior-like class that uses high damage cards and blocks, Silent is more like a rogue class tries to combo his cards with energy buffs and extra cards and weakens enemies with Poison effects which damages each turn and lastly Watcher can transform into different forms for temporary buffs. Our game has mainly four vertex types as Rest, Treasure, Merchant and Combat. The game implementation is able to use the relics, potions, cards that the player has. According to different characters, there can be various types of cards used.

2. Design Changes

In Navigation and Run Management Subsystems added several Controller Classes, in order to JavaFX implementation, which are not mentioned in the previous iterations. Controlling all scenes in one controller needs too much space so that we had to divide UI classes.

2.1 Navigation Subsystem

2.1.1 Screen Controller

A class called Screen Controller added. This class basically works as a screen navigator. It stores the .fxml files of Navigation and Run Management Subsystems and changes the scenes on demand.

2.2 Run Management Subsystem

2.2.1 Creating the Map

Since the Map should be formed from several paths which branch to each other we decide to implement the map with NodeList structure. There are 4 paths in the game which are basically NodeLists where each node is a VertexNode. Each Node composed of:
-Vertex type: indicate what type of the Vertex it is.
-Next Vertex: Next Vertex on the path.
-Alternative Next Vertex: Alternative next Vertex to branch with other paths.

All types of vertices have limited chance to be in the map. There will be 7 combat, 2 elite combat, 3 merchant, 3 rest, 3 treasure and 1 boss vertices in the map.
Each path initializes with a Combat Vertex Node. After that 3 random vertices were created.

Paths 1 and 2 merge with the top rest vertex, paths 3 and 4 merge with the bottom rest vertex at the end to one last healing or card upgrade chance. These two rest are merged with the boss vertex and the whole map will be initialized. Map class keeps the data of the current vertex.

2.2.2 New Screens

Quick Map and Deck Screens added to Run Management Subsystem.

2.3 Combat Management Subsystem

2.3.1 Singleton Combat Manager

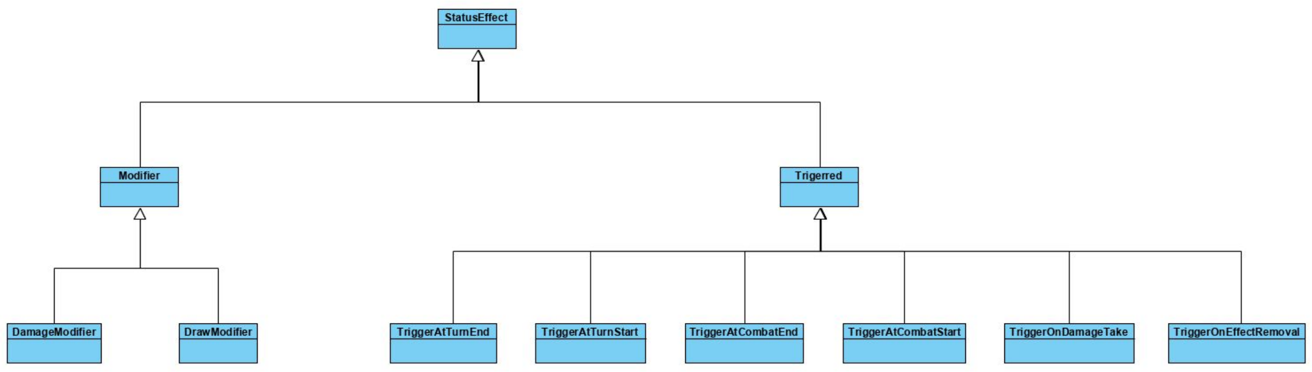
Shortly after we started implementing the combat management subsystem, we realized that in a combat scenario, the CombatManager object needed to be referenced very frequently and references to the CombatManager object needed to be passed to almost every object that participated in combat. The UI classes would also need references to the CombatManager object. On top of that, we would have a single CombatManager object for any combat scenario. To get rid of all of the CombatManager references to the same object, we decided to make CombatManager a singleton class. This way, we can address the dynamic CombatManager object statically using the getInstance() method, solving our referencing problem without forcing any additional restrictions on our design.

2.3.2 Combat Entity Abstract Class

The enemies and the player share a considerable amount of their attributes and functionality. Objects of both classes have their HP, maxHP. They both take damage and are affected by status effect. Therefore, we decided to make both classes extend a common abstract class CombatEntity. This allowed us to reuse a significant portion of our combat logic code. Also, having abstract targets for the status effect made the implementation of the new status effect interfaces (explained in the next section) much easier.

2.3.3 Status Effect Interfaces

When we decided that it was time to implement our first status effect, we realized that having an abstract class StatusEffect was not generic enough to implement a wide range of status effects. Since extendibility was one of our design goals, we decided to change our status effect architecture such that each StatusEffect subclass would implement a number of interfaces, decided by their function and when they need to be triggered. The new interfaces for status effects are the following:



This architecture enables extendibility in two levels. First, the interfaces we have can support a wide variety of status effects because a status effect's function is much more generic than just a method `affect()`, as was first proposed. Status effects can now affect the combat at different stages. Also, with the new architecture, new interfaces can be defined to support new status effects very easily.

2.3.4 Combat UI

One of the decisions made in our design report was to separate the combat UI from the combat logic completely. To do that, we had designed a CombatUIAdapter class that handles the input and the graphical output. At that time, we had no knowledge of which graphics library we would use. After we decided to use JavaFX and learned how the library works, we realized that we would need a number of UI controller classes. We implemented our combat management subsystem's UI in such a way that the CombatManager object and other logical objects need not be aware that these controllers exist. If they need to display something on the screen or make an input request, they simply call the CombatUIAdapter's designated methods, which delegate the functionality to various JavaFX specific UI controller classes. When we designed the CombatUIAdaper, we did not know about the Adapter design pattern yet. But in hindsight we notice how appropriate the name was, the CombatUIAdapter serves as an Adapter class for the complex implementation of the combat UI.

3. Lessons Learnt

3.1 Technical Issues

- Learning JavaFX from scratch is challenging. Should be well organised and consider every case before implementation. If not it can be really hard to edit the previous structure.
- Singleton classes are really useful.
- Images with high resolution may overload your memory.
- Code reviewing is an important part of this project since you may not see the small detail because it is so obvious. An eye from outside can really help.
- Saving and loading a NodeList from a .txt file is a hard design issue to implement, especially NodeLists are branching with each other.

3.2 Teamwork

- Team members should be 100% sure that they are on the same page. If not there may be huge wasted hours. (Implementing same thing or implement a thing for nothing)
- Using git is helpful to see who is doing what and take an action considering other members commit.
- Using diagrams is a helpful way to design the project before the implementation.

4. Users Guide

This section is about how a player could advance through the game from the beginning.

4.1 Main Menu

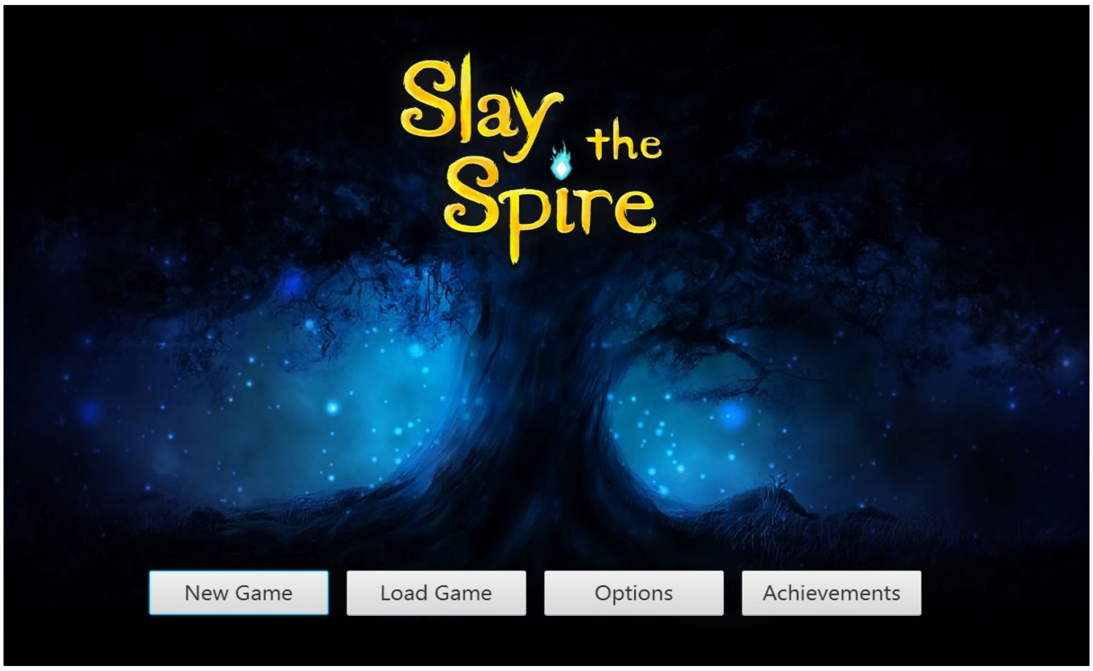
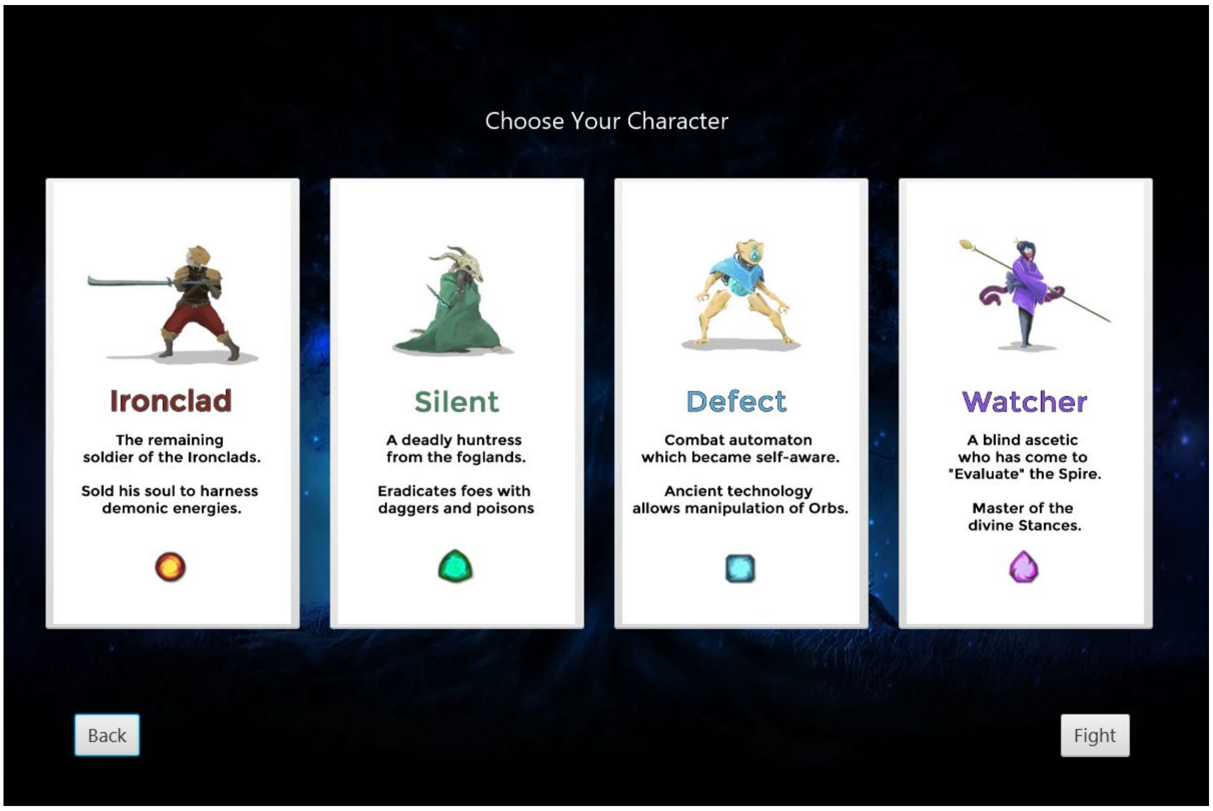


Figure 1: The Main Menu Scene of Slay the Spire game

At first, the player enters the game, the player must select ‘New Game’, since he/she does not have a previous load in the following scenes of ‘New Game’. When the user presses the new game button the name of the user is expected to be entered on the screen. After typing the user name, pressing the resume button will appear the character selection.

4.2 Character Menu


Choose Your Character

Ironclad

The remaining soldier of the Ironclads.

Sold his soul to harness demonic energies.

Silent

A deadly huntress from the foglands.

Eradicates foes with daggers and poisons

Defect

Combat automaton which became self-aware.

Ancient technology allows manipulation of Orbs.

Watcher

A blind ascetic who has come to "Evaluate" the Spire.

Master of the divine Stances.

Figure 2: The Character Menu Scene of Slay the Spire game

Four different characters are displayed. The player first chooses a name then a character. This character selection is one of the most important factors in the game since it determines the play style. If we want to briefly explain, Ironclad is a warrior-like class that uses high damage cards and blocks, Silent is more like a rogue class tries to combo his cards with energy buffs and extra cards and weakens enemies with Poison effects which damages each turn. And lastly Watcher can transform into different forms for temporary buffs.

4.3 Map View

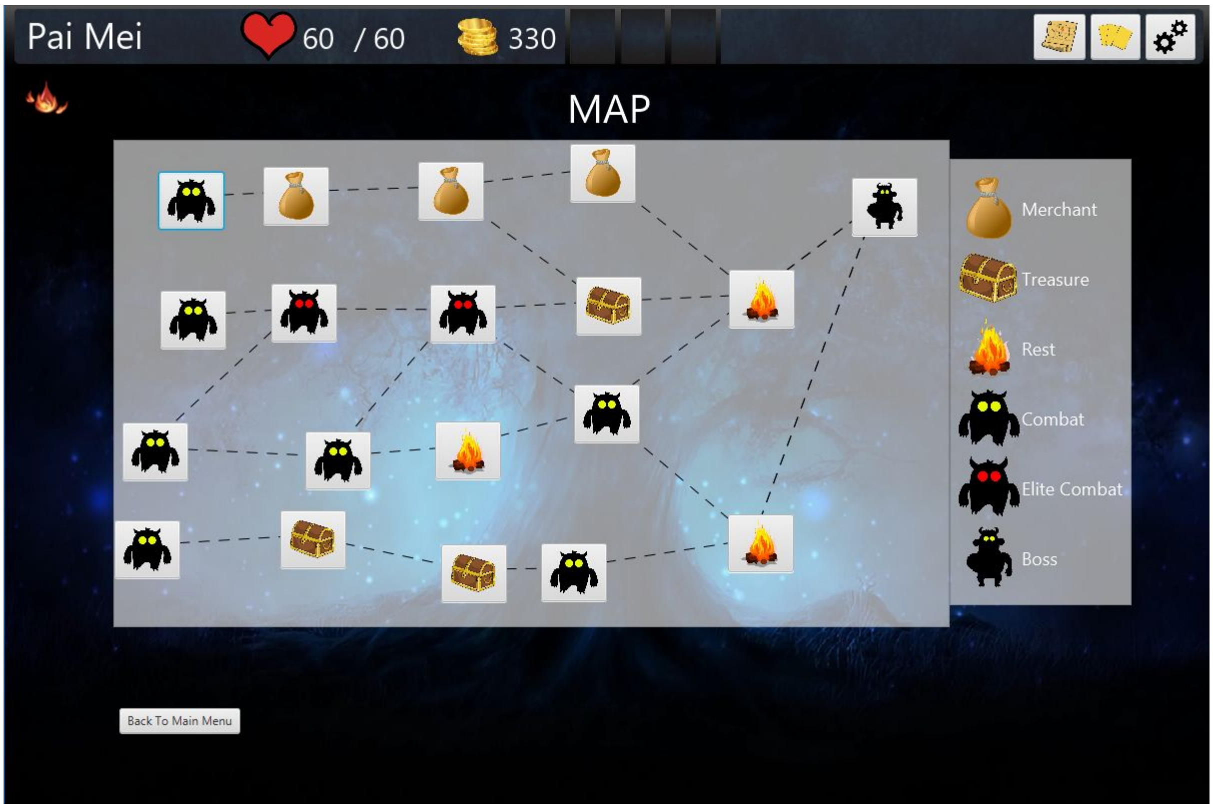


Figure 3: The Map View Scene of Slay the Spire game

At the right top of the screen, the deck of the player and settings can be displayed. When the player chooses one of the starting points, the player is navigated to the vertex chosen from the map.

How to Play

Player can manage his character on the map, but with only open vertices and can only go forward, turning back or forwarding to a locked vertex is impossible, so the player must select his way carefully. There are different vertex types with each having different content. Treasure, Merchant and Rest vertices are safe places where players can improve his/her character. (his deck, potions, relics and HP) However, Combat, Elite Combat and Boss vertices are where the real game takes place. In combat screen, the player's character is placed at left and enemies placed at right side of the screen. The player can play his/her cards by clicking them. If the clicked card has a target requirement player must also select an enemy target to play these types of cards. If the player kills all the enemies in combat, he will be rewarded, and will be returned to the Map to continue his journey. In a different situation which player gets killed, all game ends, player’s all progress and well earned attributes will be lost. Besides these, the player can reach the options from anywhere in the game. At main menu achievements can be seen, they are earned when the player did the corresponding act first time and did not disappear when the game ends.

5. Build Instructions

Make sure you have Java SDK, JRE and javafx library installed.

1) Navigate to the root directory (the directory that contains src, deliverables, this file and the sound files).

2) Run the following command in the terminal:

```
     javac -d out src/run_management/*.java
src/combat_management/*.java src/card_subclasses/*.java
src/enemy_subclasses/*.java src/navigation/*.java
src/potion_subclasses/*.java src/relic_subclasses/*.java
src/status_effect_DecayBehaviour/*.java
src/status_effect_interfaces/*.java
src/status_effect_subclasses/*.java
```

if this command does not work (Depending on your Java version), run the following (replace $PATH-TO-JAVAFX$ with the path to your javafx lib directory.):

```
     javac -d out --module-path $PATH-TO-JAVAFX$ --add-modules
javafx.controls,javafx.fxml,javafx.media src/run_management/*.java
src/combat_management/*.java src/card_subclasses/*.java
src/enemy_subclasses/*.java src/navigation/*.java
src/potion_subclasses/*.java src/relic_subclasses/*.java
src/status_effect_DecayBehaviour/*.java
src/status_effect_interfaces/*.java
src/status_effect_subclasses/*.java
```

3) Copy the contents of "src/res" to the directory "out".

4) Navigate to "out"

5) Run the following command to launch the application:

java Main

Similarly, run the following if the command does not work.(replace $PATH-TO-JAVAFX$ with the path to your javafx lib directory):

```
     java --module-path $PATH-TO-JAVAFX$-- add-modules
     javafx.controls,javafx.fxml,javafx.media Main
```

6. Work Allocation

Can Cebeci:

- Wrote the functional requirements in the Analysis Report with Cem Cebeci.
- Designed and drawn the solution-level object diagram diagram in the Analysis Report with Gökberk Boz.
- Designed and drawn the sequence diagram describing the interaction between subsystems with Can Cebeci.
- Designed the Combat Management Subsystem, drawn its corresponding diagrams and verbally explained the function of each class with Cem Cebeci.
- In the implementation phase, I abstracted myself from the GUI issues as much as possible. Revised and implemented the extensible polymorphic architecture of the Combat Management Subsystem, which then allowed me to add all of the card/relic/potion/enemy/status effect subclasses. Implemented a text-based UI to combat that was used for quite a while in testing.
- Implemented all enemy, potion, relic, DecayBehaviour, status effect subclasses.
- Implemented all status effect interfaces.
- Implemented nearly all Card subclasses, with Gökberk Boz also implementing a few of them. He also helped test all of the cards and added the images for them.
- Implemented the portion of persistent data management which deals with adding the state of an ongoing combat (if any) into the save file and re-loading combat from the save file.
- Designed & implemented the architecture based on registering cards/potions/relics/enemies as system-wide constants and the utilities that generate random behaviour based on those, the latter part together with Cem Cebeci.
- Wrote the extension guidelines in the final report.

Cem Cebeci:

- Wrote the functional requirements in the Analysis Report with Can Cebeci.
- Designed and drawn the state machine diagram in the Analysis Report.
- Designed and drawn the sequence diagram describing the interaction between subsystems with Can Cebeci.
- Helped designers of Run Management Subsystem with their design and the sequence diagram explaining it.
- Designed the Combat Management Subsystem, drawn its corresponding diagrams and verbally explained the function of each class with Can Cebeci.
- Made the necessary changes to the Combat Management Subsystem design to be able to accomodate a pet as an extra feature.
- Implemented a considerable portion of the Combat Management logic, though Can Cebeci is accountable for the bigger part.
- Implemented Combat Management UI with the help of Mehmet Ali Altunsoy on understanding JavaFX.
- Integrated Combat Management with Run Management (both logic and UI) with Mehmet Ali Altunsoy.
- Wrote the Build Instructions.
- Wrote the section about design changes in Combat Management.

Gizem Karal:

- Drawn Run Management Sequence diagram
- Designed and drawn use case diagrams with Sena Sultan Karataş for Run Management Subsystem.
- Contributed to UML diagrams for design report.
- Contributed implementation of Run Management Subsystem.
- Helped for video demo to Mehmet Ali Altunsoy

Gökberk Boz:

- Contributed to reports.
- Contributed to Run Management Subsystem diagrams.
- Implementation of some Run Subsystem Classes.
- Help to Mehmet Ali in some Run Subsystem GUI parts.
- Help to Can Cebeci in testing combat attributes and visual parts of these attributes. Also implemented some simple cards.

Mehmet Ali Altunsoy:

- Contributed to reports.
- Contributed to UML diagrams.
- Designed and drawn the Navigation Subsystem Class and Sequence Diagrams
- Implemented the GUI of Navigation and Run Management Subsystems
- Implemented the vertices functions with help of all other group members
- Design and implement the Map structure of the game
- Edit the Video Demo with help of Gizem Karal and Sena Sultan Karataş
- Help the Cem Cebeci and the Can Cebeci to implement Combat Management Subsystem.
- Help the graphic related issues. (Edit and design, images, videos, sounds, presentations etc.)

Sena Sultan Karataş:

- Contributed to Analysis reports.
- Designed and drawn use case diagrams and class diagram of Run Management Subsystems.
- Designed and drawn Run Management Subsystems with Gizem Karal.
- Contributed to UML diagrams for design report.
- Contributed implementation of Run Management Subsystem.
- Help for video demo to Mehmet Ali Altunsoy.

7. Documentation: Extension Guidelines.

We think that one of the main selling points of Slay the Spire is its variety in different types of cards, relics, potions and enemies. For this reason, extensibility was our primary design goal in this project. We wanted to make sure that adding new cards, relics, potions and enemies is as easy for a programmer as possible, which also required making status effect additions easy. We have accomplished this goal by ensuring genericity in our code and handling all sorts of cards/relics/potions/enemies/status effects polymorphically in the Combat Management Subsystem. This piece of documentation serves to describe how the game can be extended in terms of adding new cards/relics/potions/enemies/status effects.

7.1 Adding A New Card:

A new card can be added to the game by creating a new .java file in the card_subclasses folder which defines a class (from now on referred to as NewCard.java) that extends the abstract class `Card`.
Being a subclass of `Card`, `NewCard` has to implement the following in order to compile:

- A constructor that calls super (the constructor of `Card`) as its first statement, The super constructor has the following signature:
  `Card(String cardName, String cardType, int energy, String description, boolean requiresTarget, int cost)`
    - `cardName` is the name of the card as displayed by various UI elements
    - `cardType` is the type of card. This is commonly “Attack”, “Skill” or “Power”. However, a programmer may choose to pass a different type argument, for example if they wish to define a new card type “newtype” and somehow make cards of this type interact.
    - `energy` is the energy cost of the card
    - `description` is the description of the card as displayed by various UI elements
    - `requiresTarget` determines whether the player will be asked to choose a target enemy when playing the card or not
    - `cost` is the cost at which the card is sold at merchant vertices

Although not enforced by compile-time constraints, all cards in the system must have a default constructor (with no parameters).

- The abstract method `void affect( Enemy target)` in `Card`
  This method is called when the card is played. If the `isTargeted` argument passed to the super constructor call was false, the `affect` is called with target parameter null. Otherwise, it is the enemy chosen by the user while playing the card.

By default, the card can not be upgraded, which is marked by the fact that the method Card `upgradedVersion()` in Card return null. A programmer can add another card to the system (for instance, a class called NewCardPlus) and mark this card as the upgraded version of NewCard by overriding `upgradedVersion()` in NewCard as follows:

```
@Override
public Card upgradedVersion() {
      return new NewCardPlus();
}
```

If NewCard is supposed to have some of the existing additional card attributes (such as Exhaust, Ethereal, Retain etc...), they must be added within the constructor with the `addExtraAttribute(ExtraCardAttribute extraAttribute)` call. The argument is an enum with values EXHAUST and RETAIN. Other card attributes are not implemented at the current version of the project.

7.2 Adding A New Potion:

This is done almost identically to adding a new card. One has to add a new subclass of Potion, complete with a default (no parameter) constructor that calls the super constructor `Potion(String potionName, int potionCost, String potionDescription, boolean targetRequirement)` and implements `void affect( Enemy target).` The rules described regarding the target parameter in `Card` also applies here.

7.3 Adding A New Enemy:

Enemies are added to the system by defining subclasses of Enemy. They also need to have a default (no parameters) constructor that calls
`Enemy(String name, int maxHPLowerLimit, int maxHPUpperLimit)`

- `name` is the name that is displayed under the enemy in a combat view
- The enemy is constructed each time with a random maximum HP uniformly distributed between `maxHPLowerLimit` and `maxHPUpperLimit` inclusive.

One also has to implement the abstract method `declareIntent()` for each Enemy subclass.This method is called at the beginning of each turn. Although not enforced (to provide the freedom of designing enemies that do nothing at some turns), this method should make a number of calls to `addIntent(Intent i)`. Otherwise, the enemy will not do anything after the user presses the "End Turn" button.

Optionally, an Enemy subclass may override the `void restoreExtraState(String[] extraParams)` and `String getExtraState()` methods to have its state saved and loaded with the rest of the combat state. `restoreExtraState(String[] extraParams)` does nothing by default and `getExtraState()` returns an empty string. The status effects on an enemy and their HP/MAXHP are saved automatically, they need not be considered when overriding these methods.

7.4 Adding A New Status Effect:

Status effects are invoked at different times in combat. There is no one place where all status effects are handled. To account for that, the system includes a number of interfaces for status effects, making sure to handle the status effects who implement them properly. The current list of status effect interfaces is described in Section 2.3.3.

To add a new status effect, one has to decide how the status effect will be invoked in combat and add a new subclass of `StatusEffect` that implements the appropriate interfaces. For the sake of concision, how to implement each interface will not be discussed in this guideline.

Status effects decay (lose counters) in various ways. The decay behaviour (using strategy design pattern) is delegated to `DecayBehaviour` objects. By default, the delegated object is one of type `NoDecay`, but one may call `setDecayBehaviour(DecayBehaviour decay)` in the constructor to set a different decay behaviour object. Right now, the DecayBehaviour classes in the system are `DecayOncePerTurn`, `LastOneTurn` and `NoDecay`. However, it is also very easy to extend the system with new `DecayBehaviour` classes (only have to add the new class without any modification to any other class).

The one constraint is that all status effects must have a constructor with a single int argument, which determines the counter that the effect is constructed with, in order to be loaded properly from a save file.

7.5 Adding A New Relic:

The behaviour of relics are delegated to `StatusEffects` with insignificant counters and names. Counters are unused since they do not make sense for relics as an application domain object, moreover we do not ever want relic effects to "decay" and run out. Names are unused since we do not ever display a relic effect. Instead, we display relics themselves. The delegation is done through the intermediate abstract subclass `RelicEffect`, which defaults the name to null, counter to `-1` and the overrides `decay()` method with an empty body, which ensures that there is no decay even if a child class tries to set a `DecayBehaviour` object.

One adds a new relic by adding a `Relic` subclass (with a default constructor). In order not to mess up the namespace, RelicEffect's are declared as inner classes that extend RelicEffect. These effects are added as invisible status effects to the player at the start of each combat. RelicEffect subclasses need not call the super constructor explicitly (since it is empty), and they do not need to declare a constructor at all.

Each Relic subclass has to implement their default constructor as follows in order to be properly associated with its effect:

```
public NewRelic() {
   super(NAME, COST, DESCRIPTION);
   setEffect(new Effect());
}
```

where Effect is the name of the inner RelicEffect subclass.