SPL Final Report

FeatureBattle

Ivo Melse & Madelief Slaats
s1088677 & s1032615

1 Introduction

1.1 Background

Role Playing Games (RPGs) have been a staple of video games for many years. Although the definition of the genre is not completely clear, video games are mostly considered RPGs if they include some kind of battle mechanic where different players and opponents take turns, perform attacks and at the end of the battle, there is a clear winner.
We give a simplified example of an RPG battle between Player A and Player B:

1. Player A starts. Player A attacks Player B, lowering the health of B.
2. The turn goes to Player B. Player B attacks Player A, lowering the health of A.
3. The turn goes to Player A. Player A attacks Player B, lowering the health of B to zero.
4. Player A wins the battle.

1.2 Project Idea

In this project, we focused on the aforementioned RPG battles. We implemented a simple RPG battle simulator as a software product line, called "FeatureBattle". We aimed to implement optional features that affect the core game mechanics and features that affect the way that users perceive and interact with the game.

1.3 Motivation

We were motivated to to make a video game as a software product line because we think that video games can provide an intuitive, visual example to show what software product lines are, in particular for an audience that does not consist exclusively of computing scientists. RPG battle simulators seem particularly suited for this because the base game can be very simple and can be extended to be arbitrarily complex by adding features.

2 Methodology

Feature oriented programming

To realize FeatureBattle, we chose Feature Oriented Programming (FOP) as our methodology. We made this choice because FOP enables the addition of new features with minimal disruption to existing code. This is especially useful for this project, where optional features can be selectively enabled based on user preference. FOP facilitates this modular approach, making it easier to manage feature interactions and dependencies.

Tools and version control

For writing the code, we used FeatureIDE with Java and FeatureHouse. For version control, we used GitHub for collaborative and incremental development.

Quality Assurance

We took some steps in our development process to ensure quality. Firstly, all code is version-controlled. FOP enabled us to work in a very modular way. This means that our code is easily extendable. Furthermore, when a developer added a feature, they were responsible for resolving feature interactions and testing, before the changes were added to the main branch. This worked quite well in practice and effectively prevented a common problem in Software Product Lines where existing functionality breaks in an unexpected way after adding a new feature.

3 Areas of responsibilities

We give an overview of the responsibilities that both developers had. The features will be explained in more detail in the Results section.

Ivo:

- Set up FeatureIDE project & version control system
- Develop SimpleEngine
- Develop optional features:
  - AsciiEngine
  - SoundEngine
    - Music
    - SoundEffects
  - Credits
  - Rules

Madelief:

- Develop base game
- Develop optional features:
    - Extra attack options:
        - StrongAttack
        - PreciseAttack
    - Defense option: Shield
    - Computer opponent
    - Multiple battles
    - Score

Both Ivo and Madelief were responsible for testing the code after adding a new feature before adding the changes to the main branch.

4 Results

We successfully implemented FeatureBattle using the aforementioned methodology.

The base game is fully operational with turn-based attack mechanics and health tracking. The game is played as follows: there are two players that can in turn do an action. Both players start with 40 HP. During their turn, each player can choose one action from a list of options. In the base game there is only one attack option, but by selecting certain features this list can be extended to include other attack options or even a defense option. The user can also choose to play the game against a computer player by selecting the ComputerOpponent feature.

We give an overview of all the features. For more detail, we refer to the implementation. The feature tree is presented in Figure 1

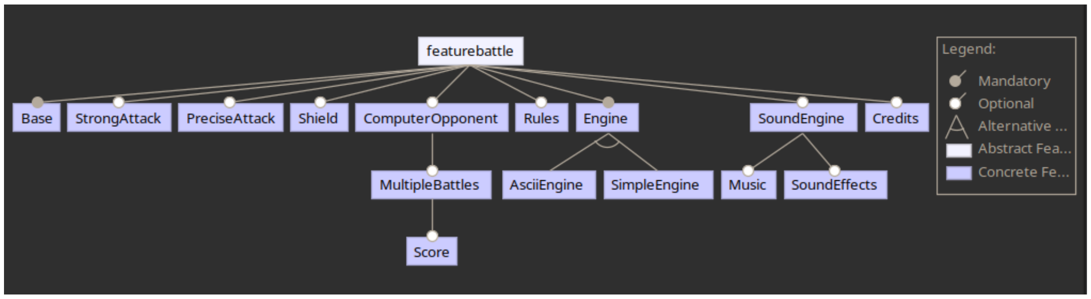


Figure 1: Feature tree of FeatureBattle

Base (mandatory)

In the base game we have two players that are both controlled by the user. Both players have 40 HP and are allowed to do one type of attack: StandardAttack. StandardAttack is an attack option that deals 10 HP damage to the opponent.

StrongAttack (optional)

An extra attack option that deals 20 HP damage to the opponent, but has a 50% chance to fail.

PreciseAttack (optional)

An extra attack option that deals 30 HP damage to the opponent, but also deals 10 HP energy cost damage to the attacker.

Shield (optional)

A defense option that reflects the next attack of the opponent. The attacker takes 5 HP energy cost damage for using a Shield.

ComputerOpponent (optional)

The user plays against a computer player that randomly chooses an action from the provided list of options. The computer player has the same options as the user-controlled player.

MultipleBattles (optional)

The user continues playing when it has defeated an opponent for as long as the user is still alive. The opponent is always a computer opponent and the new opponent will always start with maximal HP. The user does not regain HP during any of the rounds. Requires ComputerOpponent.

Score (optional)

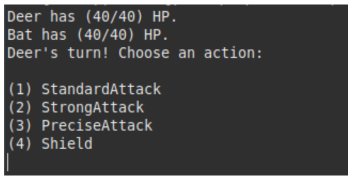
Each time the user defeats an opponent, it is displayed how many opponents have been defeated so far. When the player dies, the total amount of opponent that have been defeated is displayed. Requires MultipleBattles.

Engine (required)

An Engine is responsible for handling all user input and output (except sound).

SimpleEngine (alternative)

A very basic command-line implementation of the Engine. See Figure 2 for an example. Requires Engine.



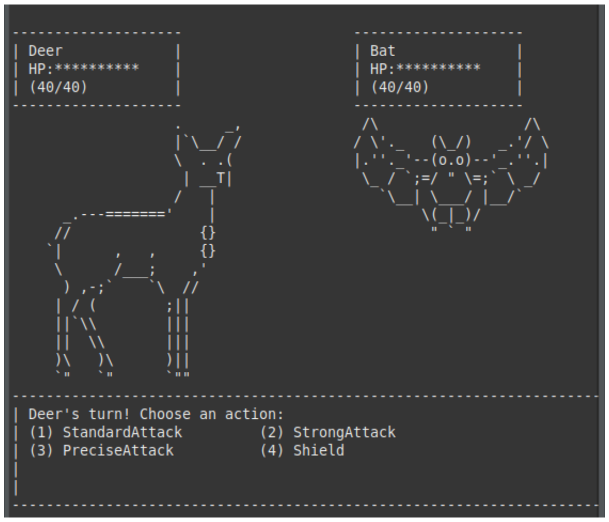
```
Deer has (40/40) HP.
Bat has (40/40) HP.
Deer's turn! Choose an action:

(1) StandardAttack
(2) StrongAttack
(3) PreciseAttack
(4) Shield
|
```

Figure 2: Example of the Simple Engine

AsciiEngine (alternative)

A more sophisticated engine that is technically still a command line interface, but designed to look like a GUI. It uses ASCII art to display the players, see Figure 3 for an example. Requires Engine.



```
--------------------                    --------------------
| Deer              |                   | Bat               |
| HP:**********     |                   | HP:**********     |
| (40/40)           |                   | (40/40)           |
--------------------                    --------------------
                   .     _,              /\                 /\
                   |`\__/ /             / \'._   (\_/)   _.'/ \
                   \  . .(              |.''._'--(o.o)--'_.''.|
                    | __T|               \_ / `;=/ " \=;` \ _/
                   /   |                   `\__| \___/ |__/`
      _.---======='    |                        \(_|_)/
     //               {}                         " ` "
    `|       ,   ,    {}
     \      /___;    ,'
      ) ,-;`    `\  //
     | / (        ;||
     ||`\\        |||
     ||  \\       |||
     )\   )\      )||
     `"   `"      `""
----------------------------------------------------------------------
| Deer's turn! Choose an action:
| (1) StandardAttack          (2) StrongAttack
| (3) PreciseAttack           (4) Shield
|
|
----------------------------------------------------------------------
```

Figure 3: Example of the ASCII Engine

SoundEngine (optional)

Abstraction for playing sounds within the game.

SoundEffects (optional)

Plays a sound effect whenever a player takes damage. Requires SoundEngine.

Music (optional)

Plays music during the game. Requires SoundEngine.

Rules (optional)

Add an explanation of how to play the game that displays when opening it. Only the rules for the actions that are allowed/have been selected are displayed.

Credits (optional)

Displays credits for FeatureBattle. If AsciiEngine is enabled, the creator of the Ascii art images is also credited. If Music is enabled, the composer is also credited.

5 Conclusion

In this project, we demonstrated the principles and advantages of Software Product Lines (SPLs) through the development of FeatureBattle, a modular RPG battle simulator. The project effectively showcases how an SPL methodology can be used to build a flexible and customizable software system, with various features that can be selectively enabled or disabled to meet user preferences.

Our use of Feature-Oriented Programming (FOP) was instrumental in achieving this flexibility. By designing the game as a collection of features, we were able to implement optional components such as advanced attack options, defensive mechanics and enhanced user interfaces in a modular and maintainable way. This modularity helped to make the development process more efficient as new features could be added with minimal disruption to the existing codebase.

Quality assurance practices, such as version control, modular testing and feature interaction management, prevented feature conflicts.

Overall, FeatureBattle achieves its goal of being an intuitive and visual demonstration of SPL methodologies.