

Advanced programming practices
SOEN6441

Section WW

Architectural Design Description

Version 1.2(build2)

By:
Team 7

Present to:
Joey Paquet

March 2016
Concordia

Design Overview

We used the general Game Architecture, which focus on separating concerns, our grouping of package was based on this notion. And also since game programming relies heavily on a loop, we created a GameLoop class which is like a timer that continuously updates the game in frames per seconds.

We will show the four major packages of our design and explain the major concept behind this grouping in the Figure below:

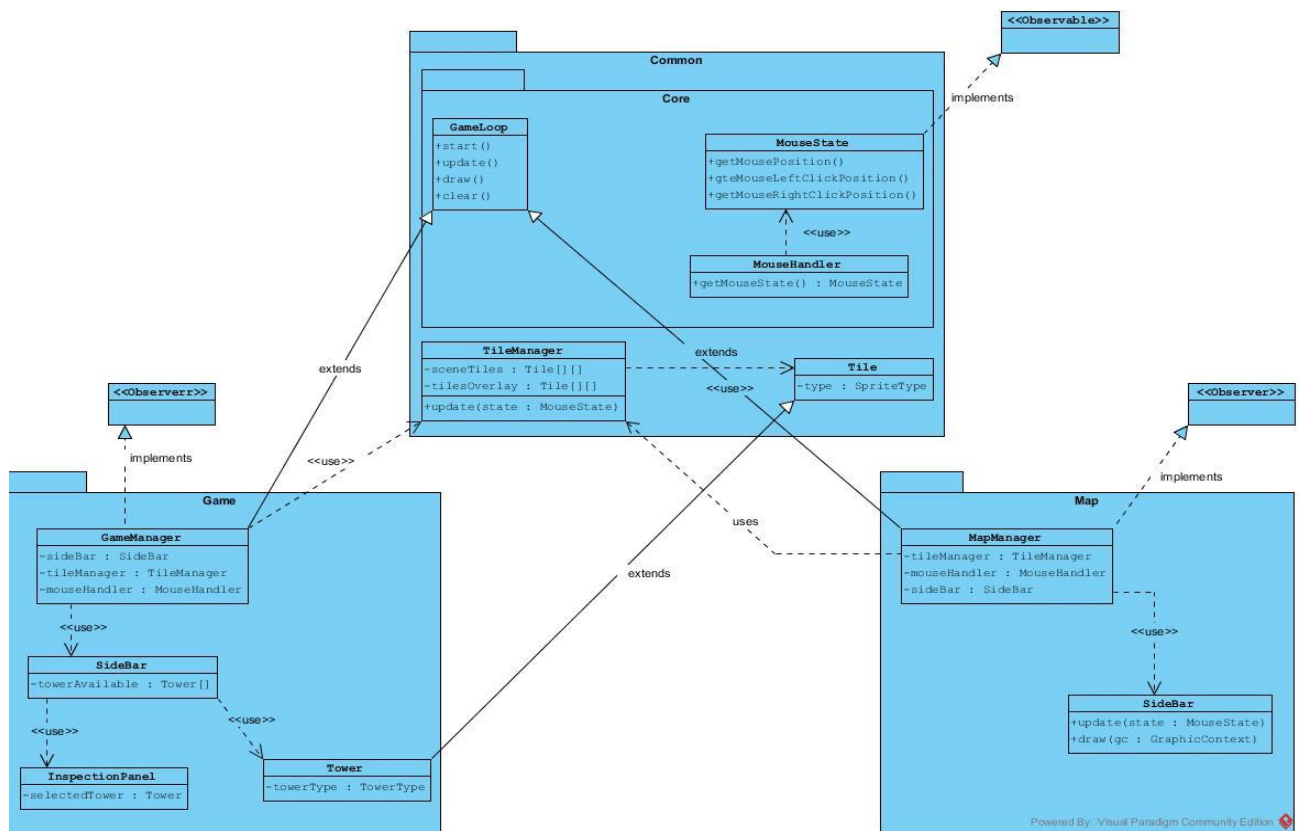


Figure1.0 – General Design

As you can see, we have the common package which represents what resources are commonly used by other packages such as the game and the map package.

The common package contains the core package which hold the GameLoop class, this can be visualized as the game engine responsible for continuous refreshing of the game environment.

The game package represents every functionality that is pertaining to the game play while the map is also responsible for all functionality belonging to map creation and map saving.

Class Description

Common classes

TileManager	Holds a tile type of 2-dimension array, used in map creation
ScoreManager	Responsible for managing level, health and money at game play
Settings	Class responsible for variable initialization
Tile	Class responsible for drawing tiles
Helper	Helps abstract commonly used method throughout game application

Game classes

GameManager	Contains all method related to game play
SideBar	Responsible for holding towers and inspection panel
InspectionPanel	Class responsible for holding tower information during game play
Button	Class responsible for holding a sprite of type button
Critter	Class that constructs a critter in the game
CrtterManager	Manages the spawn critters

Map classes

MapManger	Is the main class for handling operation on the map
SideBar	Contains method for managing sidebar items

Design Decisions

In this section we will explain the design decision we made by adopting some of the most commonly used design patterns (with little variations) based on our context, and the problem we intend to solve.

Strategy and Factory Pattern for Tower Attack

Figure1.1 show the representation of the Strategy Pattern within our context and the participating classes.

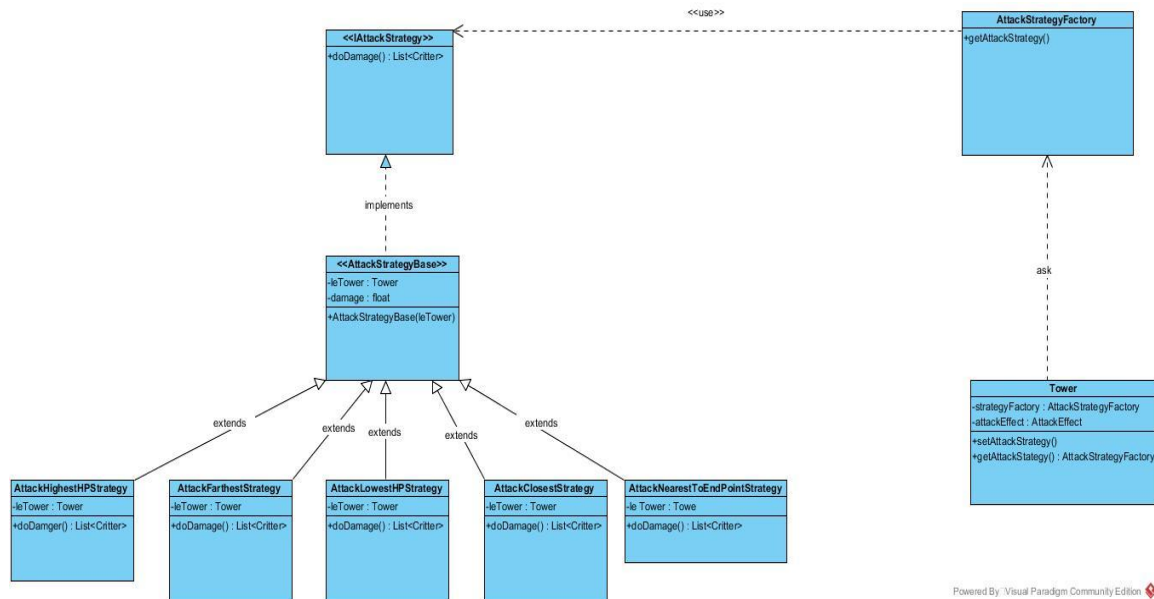


Figure1.1 – Strategy Pattern

The Problem: To allow a tower use different strategy of attacks and also allow user dynamically switch between strategies at run time.

We created three strategies, `AttackFarthestStrategy`, `AttackLowestHPStrategy`, `AttackHighestHPStrategy`, `AttackClosestStrategy` and `AttackNearestToEndStrategy`. The Tower needs to use either one of these strategies without knowing the actual implementation. Therefore, we programmed the implementation of each of the strategies to the interface `IAttackStrategy`. So based on the `attackStrategy` of the type `IAttackStrategy` selected, the `doDamage()` method gets overridden by the selected concrete strategy .

Although, there is a little variation in our implementation of this pattern. We introduced a factory class `AttackStrategyFactory`, in which gets the instance of the selected strategy using the method `getAttackStrategy` when the Tower request it.

Singleton Pattern for GameManager Class

Figure 1.3 show the class represented as a singleton and we further describe how it is been used.



Figure1.3 – Singleton Pattern

We decided to make the GameManager class a singleton because we thought that it is ideal to only have a single instance of its object through our entire game application.

The MainMenu uses the static method getInstance of the GameManager to get an object of the GameManager class provided it has not been already instantiated.

Observer Pattern for Updating MouseState

Figure 1.4 shows the representation of the Observer Design Pattern in our context and the classes involved.

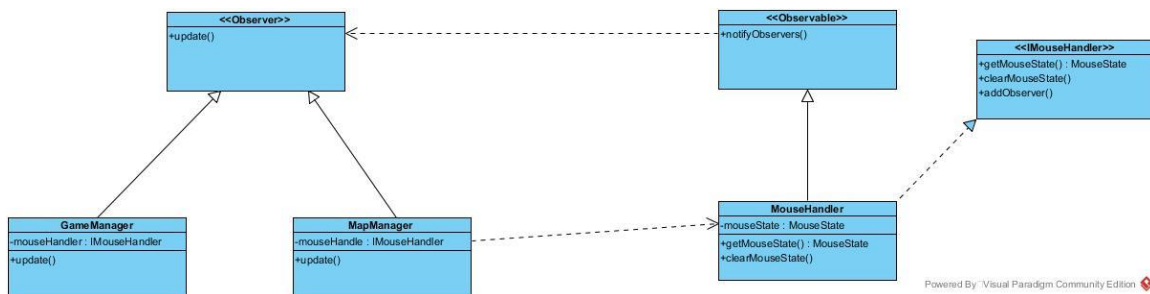


Figure1.4 – Observer Pattern

The GameManager and MapManager are the concrete observer which are updated according to the MouseState which is handled by the MouseHandler. The MouseHandler in this case is the concrete Observable and is responsible for notifying the observers when the state of the mouse changes.