# TOWER DEFENSE PROJECT: SOFTWARE DESIGN SPECIFICATIONS

Andrei Chubarau 260581375

YordanNeshev260587938

Dang Khoa Do 260584925

Steven Voyer260531264

ECSE 321 Introduction to Software Engineering
McGill University Winter 2015

## TABLE OF CONTENTS

# 1 INTRODUCTION

Tower Defence is a type of real-time strategy video game that has been around since the early 1990's. In the case of this project, the objective of the game is for the player to prevent computer controlled enemies from reaching the end of a certain path. To achieve this, the player is given a variety of gameplay options (these are detailed in the Requirements Document for this project, as well as in the System Overview section of current document).

## 1.1 PURPOSE OF THE DOCUMENT

This document summarizes and describes software architecture design of the developed Tower Defense game. An overview of the system as well as detailed insight into the overall design plans are provided, including but not limited to architectural diagrams, descriptions of the different layers of the design, and sequence diagrams to analyze the dynamic behaviour of the game.

## 1.2 AUDIENCE

The intended audience of this document is outlined below:

- Original developers of the game, with regards to structuring and managing the development process and the implementation of the game;
- Testers of the game, in their planning and definition of test cases;
- Maintenance/extension developers of the game;
- Graders of the project, namely the instructor and Teaching Assistants of ECSE 321 at McGill University.

## 1.3 SCOPE

The scope of this document includes all relevant architectural models and design specifications to the Tower Defence game. However, since this is a university project, a limited amount of information is provided as defined in the project specifications documentation (See section 1.4 Related Documents).

## 1.4 RELATED DOCUMENTS

Please see the Software Requirements Document for more information of the requirements of the development of this game.

## 2 SYSTEM OVERVIEW

The system will provide the user with options to do the following:

1. Login to or manage existing profiles
2. Create new or manage existing maps using Map Editor
3. Start new or manage existing saved games
4. Play Single Player mode. Gameplay features include:
    - buy/place/upgrade structures
    - earn score
    - inspect game element information
    - save played game

The user tasks listed above will be facilitated through a convenient user interface.

# 3 DESIGN CONSIDERATIONS

## 3.1 ASSUMPTIONS AND DEPENDENCIES

It is important to first state all assumption made during the development of this project along with any dependencies. Firstly, the project instructions do not state which type of user interface should be implemented. Therefore, a graphical interface (as opposed to a text-based console) will be made which will enable the user to observe the game state, namely the map the game is played on, enemy positions, tower positions, scenery, and tower-enemy interactions. This will be done by manipulating graphical sprites. In addition, the login, main and game time menus will all be graphically implemented as well.

Secondly, while the main theme of the game will remain Tower Defense, the developers are free to produce their own variations of the game, that is choose the overall theme of the game as well as make custom graphics, since these parameters have not been constrained by the requirements of the client.

## 3.2 CONSTRAINTS

In terms of constraints, the first and most important restriction to note is that this game must be completed by April 6th, 2015. It is essential that this time constraint be respected. Programming wise, the game must be programmed in JAVA and all in-game graphics be implemented using the Swing library and public-use or custom made images. Additionally, no monetary resources are to be expended on the game development. Gameplay wise, while little intricate features of the game are open to programmer imagination, the integrity of the original Tower Defence game must be maintained.

The following features will be present in the final product:

- Main menu and Leaderboards complete with a user login feature.
- A functional map editor which can differentiate between valid and non-valid maps. The path and the scenery must be separate (path must have one entry and one exit point)
- Currency (player score) system to buy different types of towers with different attributes (Status ailment, range, strength, projectile type, projectile speed, etc.)
- Enemies (wave-based) that will attack the player. There will be several types of enemies each with their own unique attributes (hit points, speed, kill score worth etc…)
- Towers can only be put on the scenery and enemies can only travel on the path.

All of the above features will be implemented in the final game, however extra features that are currently unspecified in this document may be implemented for gameplay extension and originality purposes at developers’ discretion.

## 3.3 GOALS AND GUIDELINES

The most important goal is that the game be playable, relatively bug-free and amusing to the player. Secondly, the game must be complete by the deadline specified by the client. In terms of programming, an important goal is to make game code as adaptable to changes as possible. This implies that future extendibility and maintenance are key to the game design. Additionally, it is a priority to implement a user-interface that is easy to understand and interact with by the end user.

## 3.4 DEVELOPMENT METHODS

The project is to be directed following the principles of the iterative Agile development method. Stages of the development process can be intertwined and carried out simultaneously. Stages can also be reviewed or reiterated upon need.

The development methods that will be used in this project include object oriented programming, test driven iterative methods and the GRASP programming principles.

# 4 SYSTEM ARCHITECTURE

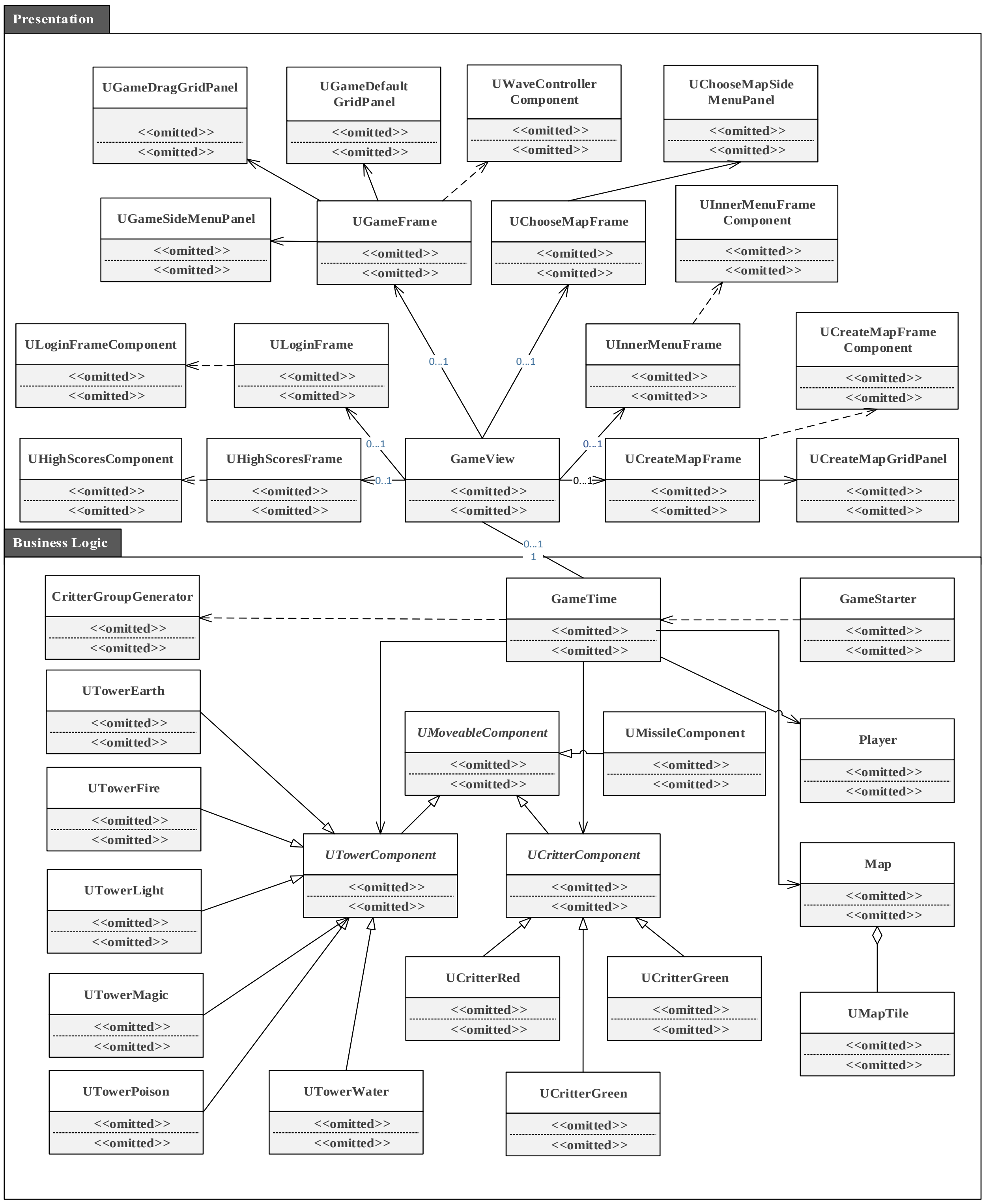
## 4.1 ARCHITECTURAL STRATEGIES

The design of the current system is based on the layered architectural style, where the main system is divided into two main subcomponents, namely Presentation and Business Logic layers. Such an approach limits interaction between the two layers and separates responsibilities (where applicable) to achieve higher system cohesion.

The previously mentioned duality of the system ensures separation of concerns. Business Logic layer handles most system parameters, whereas the Presentation layer has the responsibility of displaying the resulting system behaviour as well as handling user input based on the displayed User Interface. This contributes to managing system complexity by separating system domain logic and system view functionality.

Moreover, Presentation layer depends on Business Logic layer as the latest provides services and information to the first. The Model-View separation principle is applied as Business Logic has no dependency on the Presentation layer. This design style contributes to high cohesion and low coupling of the system. Additionally, this allows the system to display the same domain logic component in different presentation styles, thus enhancing reuse and extendibility.

## 4.2 Architectural Diagram



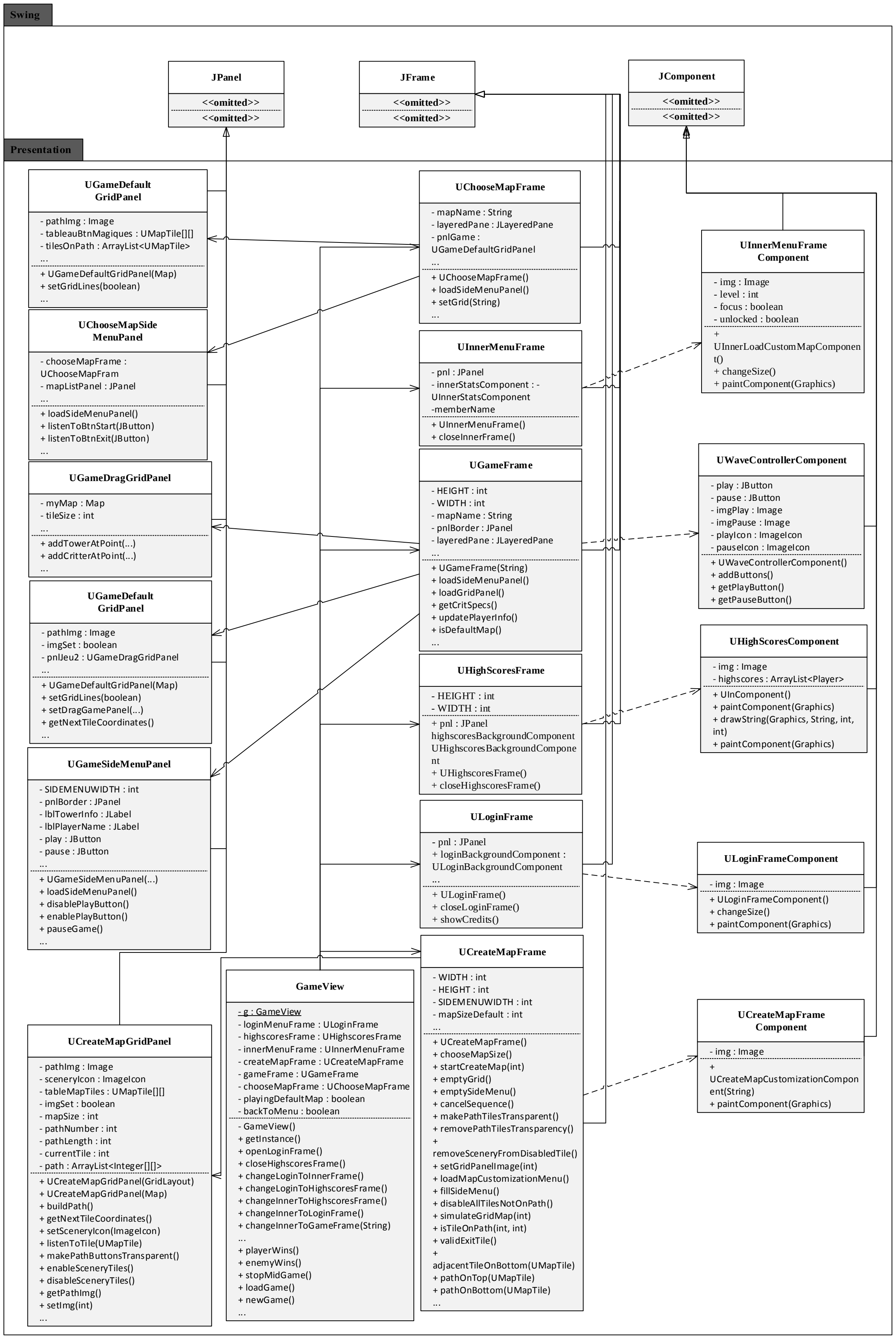
# 5 DETAILED SYSTEM DESIGN

## 5.1 COMPONENT LEVEL DESIGN

System subcomponents are explained in detail in the following sections.

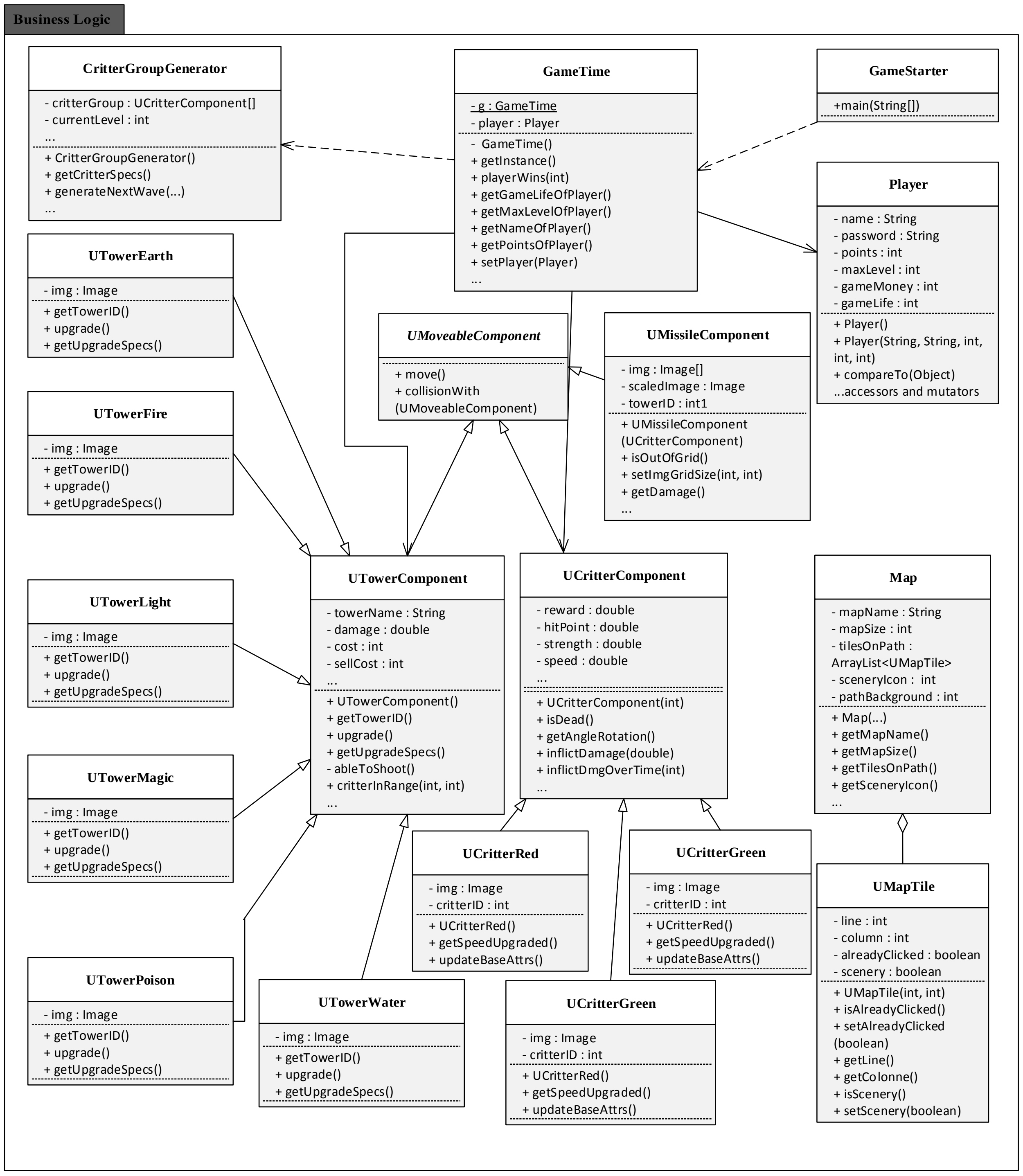
### 5.1.1 Presentation Layer

GameView class is at the base of the Presentation layer. It is responsible for displaying the system parameters in addition to handling and communicating user input to the Business Logic layer. GameView communicates with a single GameTime instance that maintains all game parameters of the system. A variation of subpanels is developed to provide user interface including game menus, game view components and game object representations. A system of switches is implemented in order to display a variety of game views according to current game state.


Swing
JPanel
<<omitted>>
<<omitted>>
JFrame
<<omitted>>
<<omitted>>
JComponent
<<omitted>>
<<omitted>>
Presentation
UGameDefault GridPanel
- pathImg : Image
- tableauBtnMagiques : UMapTile[][]
- tilesOnPath : ArrayList<UMapTile>
...
+ UGameDefaultGridPanel(Map)
+ setGridLines(boolean)
...
UChooseMapSide MenuPanel
- chooseMapFrame : UChooseMapFram
- mapListPanel : JPanel
...
+ loadSideMenuPanel()
+ listenToBtnStart(JButton)
+ listenToBtnExit(JButton)
...
UGameDragGridPanel
- myMap : Map
- tileSize : int
...
+ addTowerAtPoint(...)
+ addCritterAtPoint(...)
...
UGameDefault GridPanel
- pathImg : Image
- imgSet : boolean
- pnlJeu2 : UGameDragGridPanel
...
+ UGameDefaultGridPanel(Map)
+ setGridLines(boolean)
+ setDragGamePanel(...)
+ getNextTileCoordinates()
...
UGameSideMenuPanel
- SIDEMENUWIDTH : int
- pnlBorder : JPanel
- lblTowerInfo : JLabel
- lblPlayerName : JLabel
- play : JButton
- pause : JButton
...
+ UGameSideMenuPanel(...)
+ loadSideMenuPanel()
+ disablePlayButton()
+ enablePlayButton()
+ pauseGame()
...
UCreateMapGridPanel
- pathImg : Image
- sceneryIcon : ImageIcon
- tableMapTiles : UMapTile[][]
- imgSet : boolean
- mapSize : int
- pathNumber : int
- pathLength : int
- currentTile : int
- path : ArrayList<Integer[][]>
+ UCreateMapGridPanel(GridLayout)
+ UCreateMapGridPanel(Map)
+ buildPath()
+ getNextTileCoordinates()
+ setSceneryIcon(ImageIcon)
+ listenToTile(UMapTile)
+ makePathButtonsTransparent()
+ enableSceneryTiles()
+ disableSceneryTiles()
+ getPathImg()
+ setImg(int)
...
UChooseMapFrame
- mapName : String
- layeredPane : JLayeredPane
- pnlGame : UGameDefaultGridPanel
...
+ UChooseMapFrame()
+ loadSideMenuPanel()
+ setGrid(String)
...
UInnerMenuFrame
- pnl : JPanel
- innerStatsComponent : -UInnerStatsComponent
-memberName
+ UInnerMenuFrame()
+ closeInnerFrame()
UGameFrame
- HEIGHT : int
- WIDTH : int
- mapName : String
- pnlBorder : JPanel
- layeredPane : JLayeredPane
...
+ UGameFrame(String)
+ loadSideMenuPanel()
+ loadGridPanel()
+ getCritSpecs()
+ updatePlayerInfo()
+ isDefaultMap()
...
UHighScoresFrame
- HEIGHT : int
- WIDTH : int
+ pnl : JPanel
highscoresBackgroundComponent UHighscoresBackgroundComponent
+ UHighscoresFrame()
+ closeHighscoresFrame()
ULoginFrame
- pnl : JPanel
+ loginBackgroundComponent : ULoginBackgroundComponent
...
+ ULoginFrame()
+ closeLoginFrame()
+ showCredits()
UCreateMapFrame
- WIDTH : int
- HEIGHT : int
- SIDEMENUWIDTH : int
- mapSizeDefault : int
...
+ UCreateMapFrame()
+ chooseMapSize()
+ startCreateMap(int)
+ emptyGrid()
+ emptySideMenu()
+ cancelSequence()
+ makePathTilesTransparent()
+ removePathTilesTransparency()
+ removeSceneryFromDisabledTile()
+ setGridPanelImage(int)
+ loadMapCustomizationMenu()
+ fillSideMenu()
+ disableAllTilesNotOnPath()
+ simulateGridMap(int)
+ isTileOnPath(int, int)
+ validExitTile()
+ adjacentTileOnBottom(UMapTile)
+ pathOnTop(UMapTile)
+ pathOnBottom(UMapTile)
...
GameView
- g : GameView
- loginMenuFrame : ULoginFrame
- highscoresFrame : UHighscoresFrame
- innerMenuFrame : UInnerMenuFrame
- createMapFrame : UCreateMapFrame
- gameFrame : UGameFrame
- chooseMapFrame : UChooseMapFrame
- playingDefaultMap : boolean
- backToMenu : boolean
- GameView()
+ getInstance()
+ openLoginFrame()
+ closeHighscoresFrame()
+ changeLoginToInnerFrame()
+ changeLoginToHighscoresFrame()
+ changeInnerToHighscoresFrame()
+ changeInnerToLoginFrame()
+ changeInnerToGameFrame(String)
...
+ playerWins()
+ enemyWins()
+ stopMidGame()
+ loadGame()
+ newGame()
...
UInnerMenuFrame Component
- img : Image
- level : int
- focus : boolean
- unlocked : boolean
+ UInnerLoadCustomMapComponent()
+ changeSize()
+ paintComponent(Graphics)
UWaveControllerComponent
- play : JButton
- pause : JButton
- imgPlay : Image
- imgPause : Image
- playIcon : ImageIcon
- pauseIcon : ImageIcon
+ UWaveControllerComponent()
+ addButtons()
+ getPlayButton()
+ getPauseButton()
UHighScoresComponent
- img : Image
- highscores : ArrayList<Player>
+ UInComponent()
+ paintComponent(Graphics)
+ drawString(Graphics, String, int, int)
+ paintComponent(Graphics)
ULoginFrameComponent
- img : Image
+ ULoginFrameComponent()
+ changeSize()
+ paintComponent(Graphics)
UCreateMapFrame Component
- img : Image
+ UCreateMapCustomizationComponent(String)
+ paintComponent(Graphics)

### 5.1.2 Domain (Business) Logic Layer

Business Logic layer is mainly characterized by the GameTime class. An instance of GameTime contains all runtime parameters of the system such as Map, Structures, Critters, and others. Presentation layer, namely GameView, accesses GameTime to acquire system state and display it, but GameTime does not initialize communication with the presentation layer. GameView listens for and handles user input communicated to it through the presentation layer components and passes on the information to the GameTime instance that applies required changes to individual business layer components.


Business Logic
CritterGroupGenerator
- critterGroup : UCritterComponent[]
- currentLevel : int
...
+ CritterGroupGenerator()
+ getCritterSpecs()
+ generateNextWave(...)
...
GameTime
- g : GameTime
- player : Player
- GameTime()
+ getInstance()
+ playerWins(int)
+ getGameLifeOfPlayer()
+ getMaxLevelOfPlayer()
+ getNameOfPlayer()
+ getPointsOfPlayer()
+ setPlayer(Player)
...
GameStarter
+main(String[])
Player
- name : String
- password : String
- points : int
- maxLevel : int
- gameMoney : int
- gameLife : int
+ Player()
+ Player(String, String, int, int, int)
+ compareTo(Object)
...accessors and mutators
UTowerEarth
- img : Image
+ getTowerID()
+ upgrade()
+ getUpgradeSpecs()
UMoveableComponent
+ move()
+ collisionWith (UMoveableComponent)
UMissileComponent
- img : Image[]
- scaledImage : Image
- towerID : int1
+ UMissileComponent (UCritterComponent)
+ isOutOfGrid()
+ setImgGridSize(int, int)
+ getDamage()
...
UTowerFire
- img : Image
+ getTowerID()
+ upgrade()
+ getUpgradeSpecs()
UTowerLight
- img : Image
+ getTowerID()
+ upgrade()
+ getUpgradeSpecs()
UTowerComponent
- towerName : String
- damage : double
- cost : int
- sellCost : int
...
+ UTowerComponent()
+ getTowerID()
+ upgrade()
+ getUpgradeSpecs()
- ableToShoot()
+ critterInRange(int, int)
...
UCritterComponent
- reward : double
- hitPoint : double
- strength : double
- speed : double
...
+ UCritterComponent(int)
+ isDead()
+ getAngleRotation()
+ inflictDamage(double)
+ inflictDmgOverTime(int)
...
Map
- mapName : String
- mapSize : int
- tilesOnPath : ArrayList&lt;UMapTile&gt;
- sceneryIcon : int
- pathBackground : int
+ Map(...)
+ getMapName()
+ getMapSize()
+ getTilesOnPath()
+ getSceneryIcon()
...
UTowerMagic
- img : Image
+ getTowerID()
+ upgrade()
+ getUpgradeSpecs()
UCritterRed
- img : Image
- critterID : int
+ UCritterRed()
+ getSpeedUpgraded()
+ updateBaseAttrs()
UCritterGreen
- img : Image
- critterID : int
+ UCritterRed()
+ getSpeedUpgraded()
+ updateBaseAttrs()
UMapTile
- line : int
- column : int
- alreadyClicked : boolean
- scenery : boolean
+ UMapTile(int, int)
+ isAlreadyClicked()
+ setAlreadyClicked (boolean)
+ getLine()
+ getColonne()
+ isScenery()
+ setScenery(boolean)
UTowerPoison
- img : Image
+ getTowerID()
+ upgrade()
+ getUpgradeSpecs()
UTowerWater
- img : Image
+ getTowerID()
+ upgrade()
+ getUpgradeSpecs()
UCritterGreen
- img : Image
- critterID : int
+ UCritterRed()
+ getSpeedUpgraded()
+ updateBaseAttrs()

## 5.2 Object Oriented Principles and Patterns

This section explains and concretises on design decisions carried out during the development process of the current system. As these decisions were made based on common object-oriented principles and patterns (GRASP and GoF), the following text describes and demonstrates decision applicability and significance in terms of these notions. Additionally, relevant partial UML class diagrams or Domain Models are provided for visual inspection of considered system components, where applicable.

### 5.2.1 Principles Used

1. Principle: Information Expert

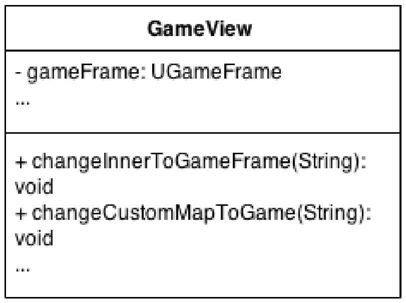
   Problem: What system component should be assigned the responsibility of generating and maintaining the path for critters to move on?

   Solution: In the current system, Map class manages and constructs all map related components of a game instance. This includes storing all tile information and links between these. The process of building a path depends on and is limited by map parameters which must be accessed during construction of path. Therefore, the Map class should be responsible for building and maintaining path and is the information expert in this scenario.

2. Principle: Creator

   Problem: What system component should be assigned the responsibility of creating an instance of UGameFrame?

   Solution: In the current system, GameView class is responsible for handling window interaction and view changes associated to the runtime state of the game, notably the map components and game menu panels. Therefore, the GameView should initialize an instance of UGameFrame when changing from a menu to the game frame.


GameView
- gameFrame: UGameFrame
...
+ changeInnerToGameFrame(String): void
+ changeCustomMapToGame(String): void
...

3. Principle: Controller

   Problem: What system component should be in charge of handling user input events?

   Solution: In the current application, each view component has a separate controller (where needed or applicable).This is done so to allow for relevant user input processing and runtime parameters manipulation as defined by the panel. Separating all of this functionality into a controller class would have resulted in significant complexity and importance due to high responsibility of a single class and was judged an ineffective solution.

4. Principle: Polymorphism

   Problem: How should the application handle the behaviour of different runtime type of similar elements?

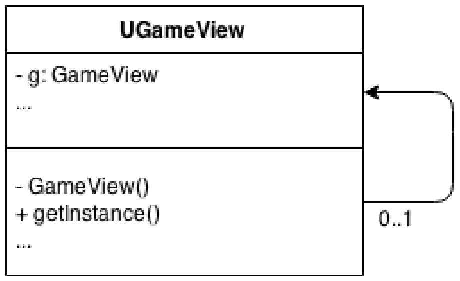
   Solution: In this application, Polymorphism is applied in the definition of frames, panels, panel components, structures, and critters. Each individual previously mentioned type is designed to have runtime variations, or subtypes. For example, subtypes extending the Structure class provide custom definitions of the generic methods found in their parent Structure class. This implies that the behaviour of these subtypes will be determined during runtime of the program. These methods are defined differently for each subtype based on given relevant criteria, although they result in similar final behaviour. Such approach allows for low coupling between subsystems and overall high system cohesion.

### 5.2.2 Pattern Used

1. Pattern: Singleton

   Problem: How to ensure that only one instance of UGameView is ever created?

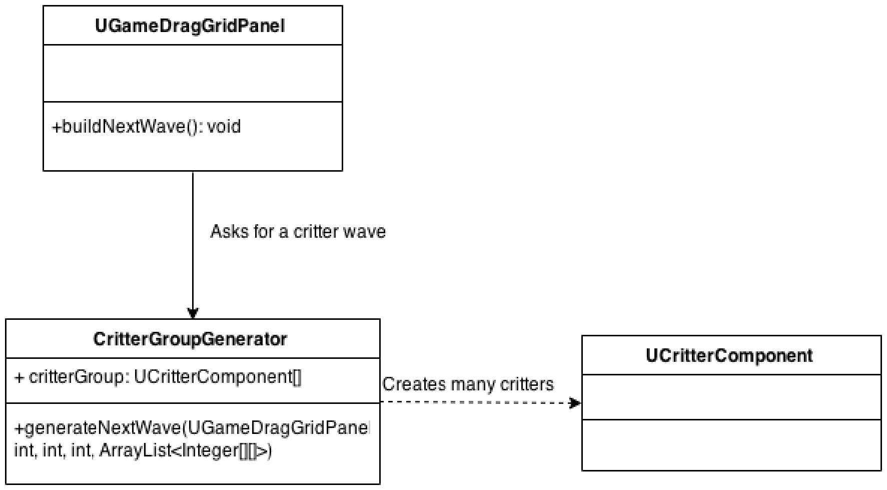
   Solution: Apply the Singleton pattern on UGameView. Using this pattern, UGameView is implemented in such a way that all attempts to create a new instance of it either return a new instance if none has been previously initialized or return the very same existing instance. In such a way, a maximum of only one element of UGameView is ever used.



2. Pattern: Factory

   Problem: How to generate a group of related objects of different types?

   Solution: CritterWaveGenerator class is responsible for generating a list of instances of Critter subclasses for use by the game. The class generates a list of Critter waves based on level, difficulty and map parameters defined by GameTime class.



3. Pattern: Observer

   Problem: How to handle runtime changes to game parameters with respect to updating Graphical User Interface (GUI)?

   Solution: A 'basic' system-level event-based observer pattern (in which GUI would be updated only if some game parameter was changed) was judged to be ineffective for application in gaming environment of this scale and thus not implemented in the current system on all components. Instead, a slightly different approach was used to ensure that GUI displays recent information to the end user. In the current design, the system regularly updates GUI display with a predetermined frequency in order to provide the User with an updated visual representation of the game. This implies that most graphical elements are redrawn upon any given GUI update. This ensures that all game elements are updated and thus represent the most recently available game state. UGameView is the main handler and container of all components of the GUI and thus handles its own updates. However, game menus (UI panels, but not the game map UI) are updated only when a visual change requires their update.

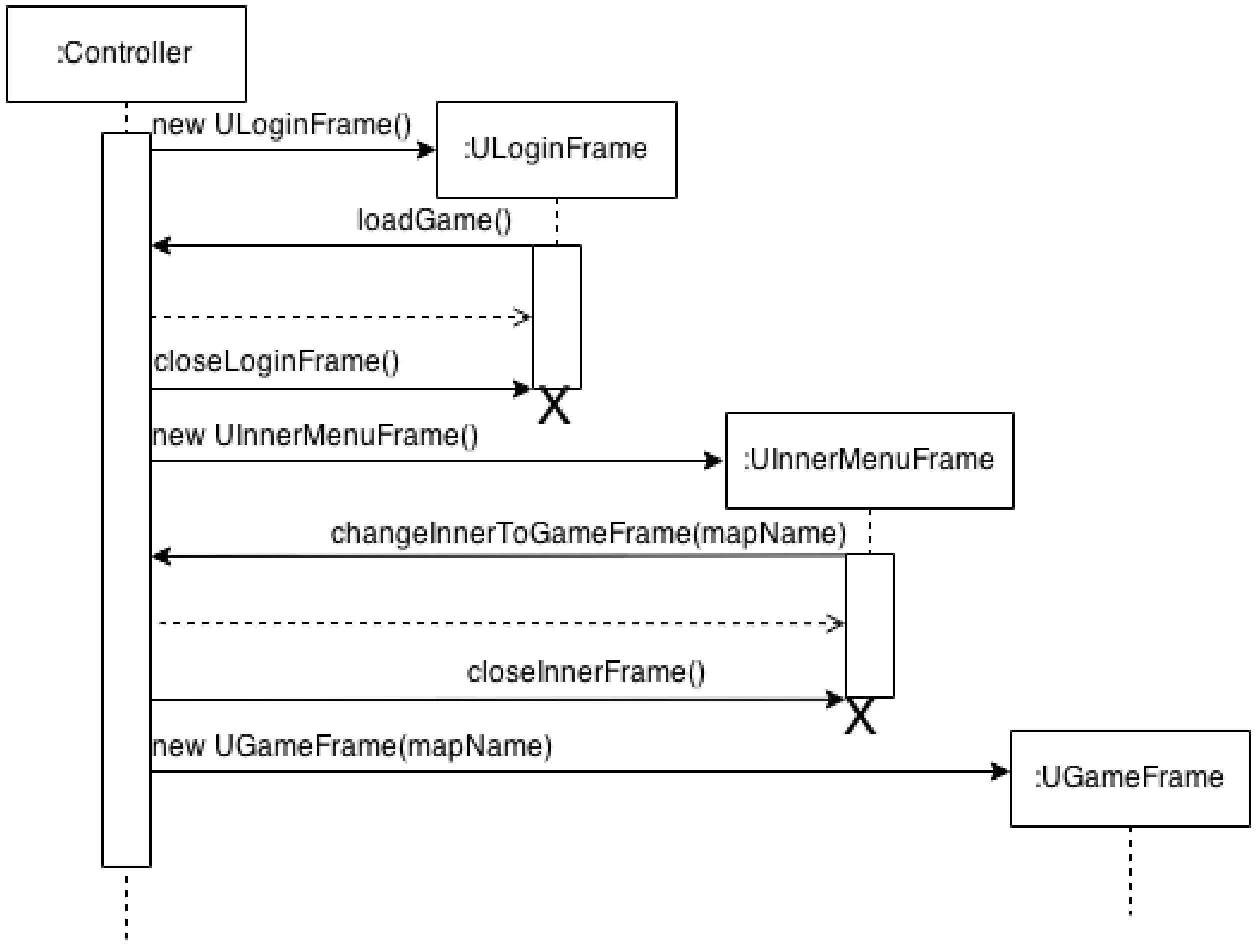
## 6 DYNAMIC BEHAVIOUR: SEQUENCE DIAGRAMS

A few Use Cases, that were created while making the requirements specifications, were picked to display how the classes interact with each other in the system:

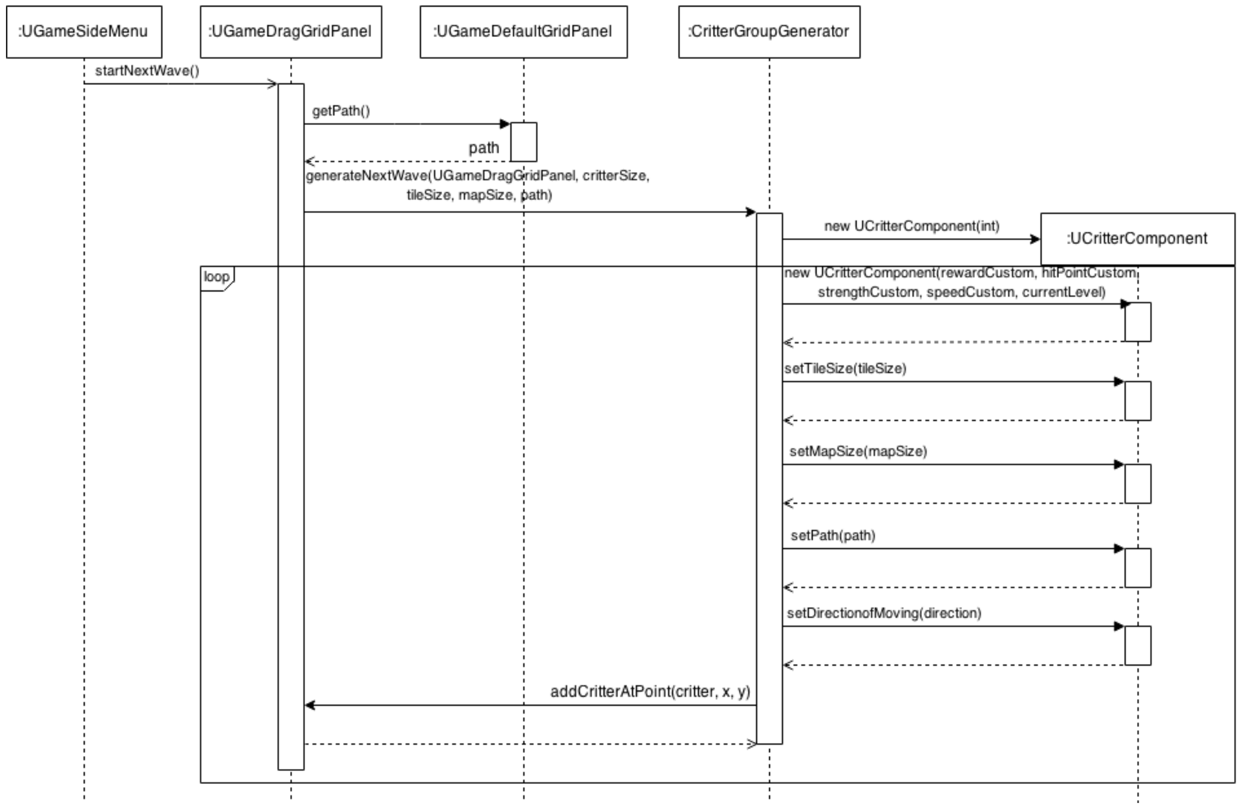
- Use Case 13: Load game
- Use Case 15: Start enemy attack
- Use Case 17: Buy Structure

The use cases above are the main operations that are implemented in the game. Many different objects work together, such as UGameDragGridPanel, UGameDefaultGridPanel, and UGameSideMenu which use the UTowerComponent and CritterGenerator classes to create towers and to start a new critter wave. The sequence diagram for Use Case 17: Buy Structure only shows how the operation is performed with UTowerFireComponent, but it is the same interactions if any other type of tower is used.

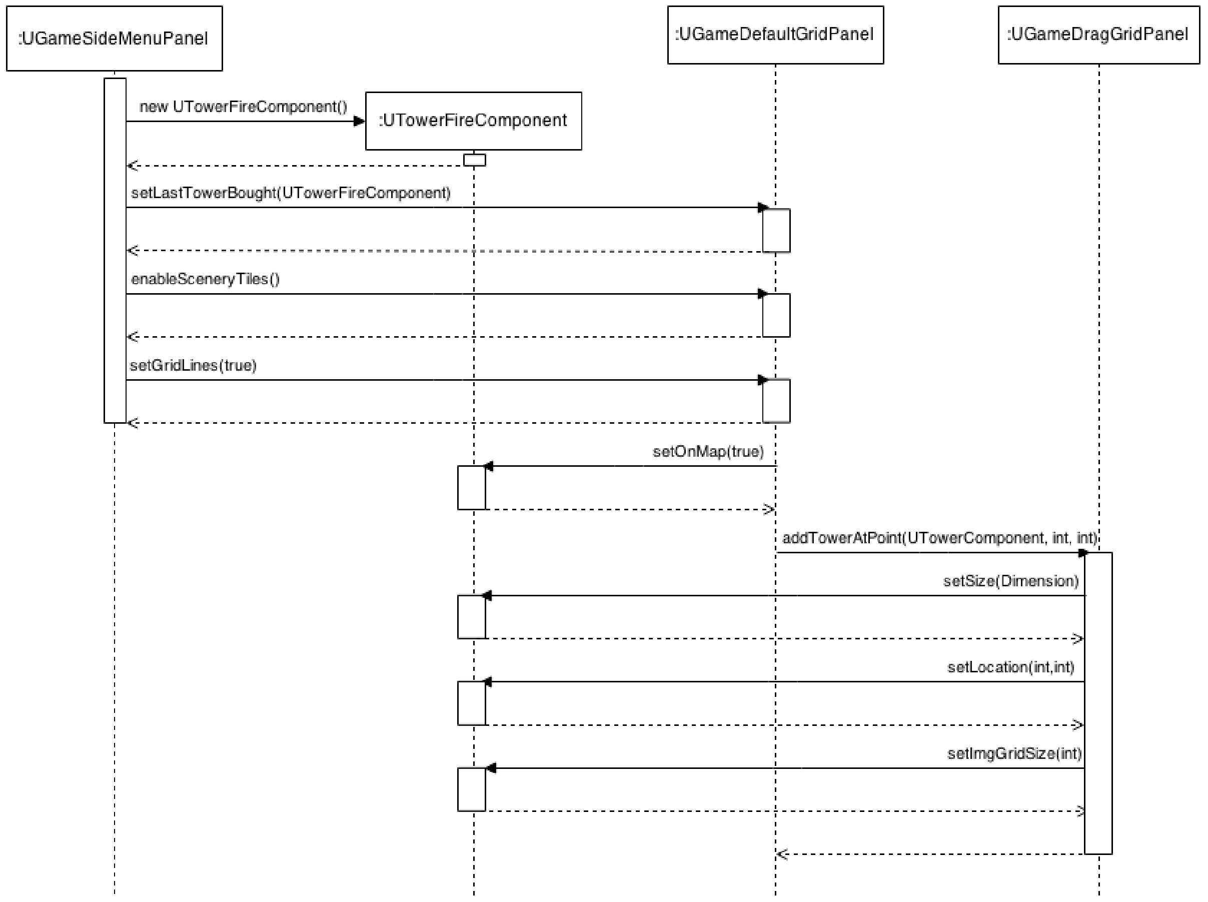
**Use Case 13: Load game**



## Use Case 15: Start enemy attack


:UGameSideMenu
:UGameDragGridPanel
:UGameDefaultGridPanel
:CritterGroupGenerator
:UCritterComponent
startNextWave()
getPath()
path
generateNextWave(UGameDragGridPanel, critterSize, tileSize, mapSize, path)
new UCritterComponent(int)
loop
new UCritterComponent(rewardCustom, hitPointCustom, strengthCustom, speedCustom, currentLevel)
setTileSize(tileSize)
setMapSize(mapSize)
setPath(path)
setDirectionofMoving(direction)
addCritterAtPoint(critter, x, y)

## Use Case 17: Buy structure



# 7 USER INTERFACE DESIGN

## 7.1 DESCRIPTION OF THE USER INTERFACE

In order to implement User Interface of the current application, Java SWING library was adopted. This allows the system to produce powerful visuals with high hardware performance. SWING contains convenient and easy to use implementations UI components such as of buttons, panels, fields, frames, etc. In the case of our application, JFrame, JPanel, JButton and JComponent were largely used.

## 7.2 VISUAL PRESENTATION OF GRAPHICAL USER INTERFACE

An example of Graphical User Interface (GUI) can be observed on the figures below.

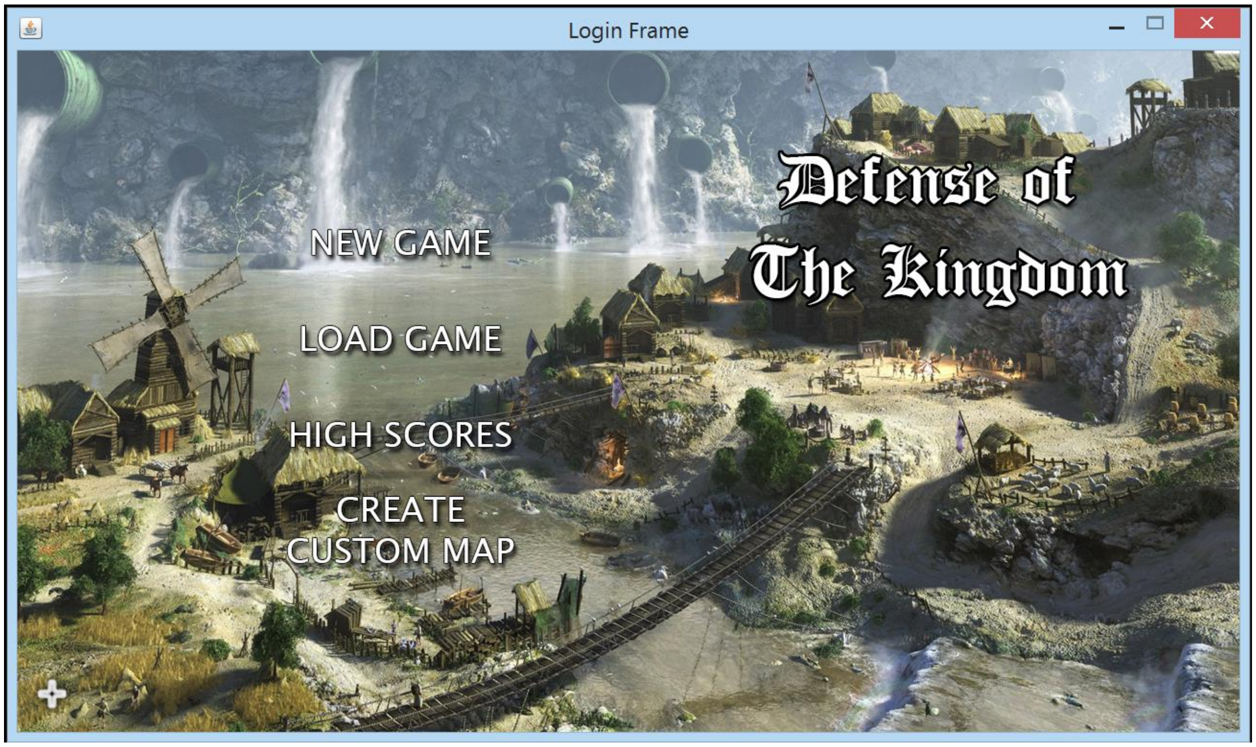
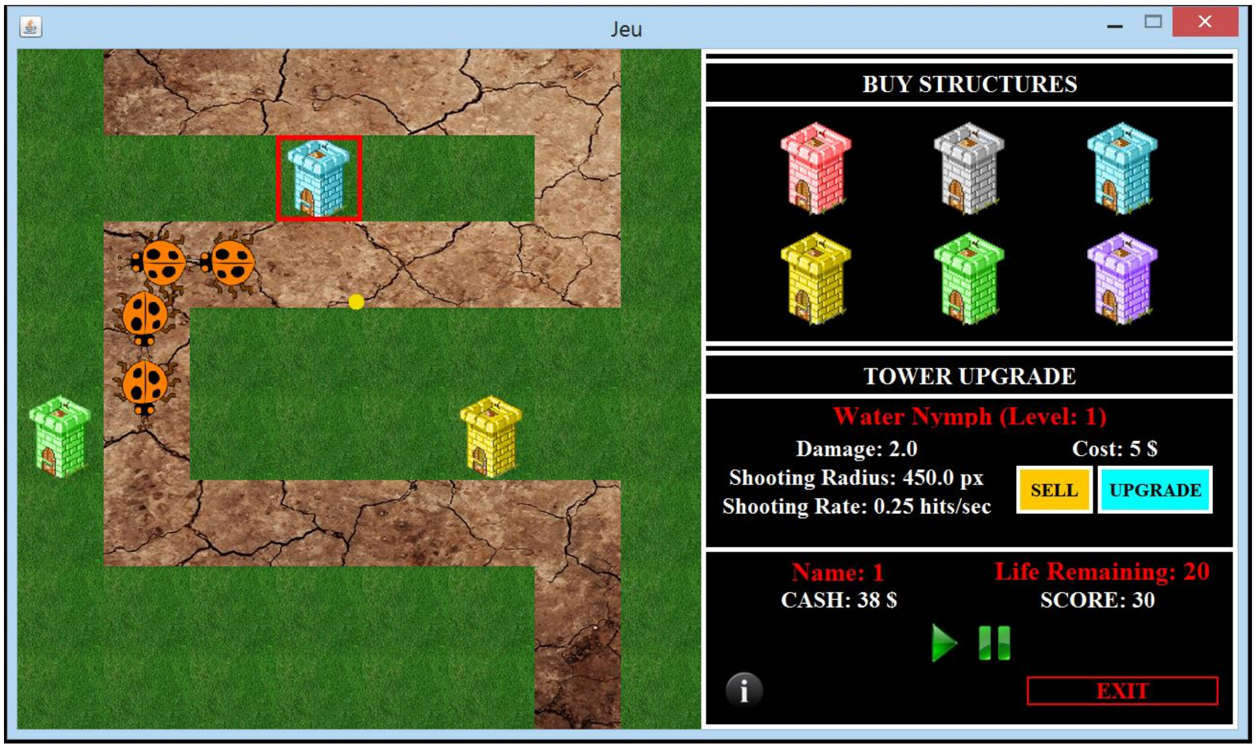


Figure 1. Main menu of the game



Water Nymph (Level: 1)
Damage: 2.0
Cost: 5 $
Shooting Radius: 450.0 px
Shooting Rate: 0.25 hits/sec

Name: 1
Life Remaining: 20
CASH: 38 $
SCORE: 30

Figure 2. Map User Interface of the game

## 8 REFERENCES

**Daniel Sinnig PhD** Lecture Slides, ECSE-321. McGill University Winter 2015

**Martin Fowler** UML Distilled: A Brief Guide to the Standard Object Modelling Language