ECSE 321 Introduction to Software Engineering

McGill University Winter 2015

Tower defense project:  
Software Design Specifications

Andrei Chubarau 260581375

Yordan Neshev 260587938

Dang Khoa Do 260584925

Steven Voyer 260531264

Table of Contents

[1 Introduction 2](#_Toc416102601)

[1.1 Purpose of the document 2](#_Toc416102602)

[1.2 Audience 2](#_Toc416102603)

[1.3 Scope 2](#_Toc416102604)

[1.4 Related documents 2](#_Toc416102605)

[2 System Overview 3](#_Toc416102606)

[3 Design Considerations 4](#_Toc416102607)

[3.1 Assumptions and Dependencies 4](#_Toc416102608)

[3.2 Constraints 4](#_Toc416102609)

[3.3 Goals and Guidelines 5](#_Toc416102610)

[3.4 Development Methods 5](#_Toc416102611)

[4 System Architecture 6](#_Toc416102612)

[4.1 Architectural Strategies 6](#_Toc416102613)

[4.2 Architectural Diagram 7](#_Toc416102614)

[5 Detailed System Design 8](#_Toc416102615)

[5.1 Component level design 8](#_Toc416102616)

[5.1.1 Presentation Layer 8](#_Toc416102617)

[5.1.2 Domain Logic Layer 9](#_Toc416102618)

[5.2 Object Oriented Principles and Patterns 10](#_Toc416102619)

[5.2.1 Principles Used 10](#_Toc416102620)

[5.2.2 Pattern Used 11](#_Toc416102621)

[6 Dynamic Behaviour: Sequence Diagrams 13](#_Toc416102622)

[7 User Interface Design 14](#_Toc416102623)

[7.1 Description of the User Interface 14](#_Toc416102624)

[7.2 Visual Presentation of Graphical User Interface 14](#_Toc416102625)

[8 References 15](#_Toc416102626)

# 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).

## 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.

## 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.

## 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).

## Related documents

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

# 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 SinglePlayer 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.

# Design Considerations

## 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.

## 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.

## 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.

## 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.

# System Architecture

## 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 to achieve higher system cohesion.

The previously mentioned duality of the system ensures separation of concerns. Business Logic layer handles all system parameters and user inputs, whereas Presentation layer has the responsibility of displaying the resulting system behaviour. 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.

## Architectural Diagram

[INSERT DIAGRAM HERE]

# Detailed System Design

## Component level design

System subcomponents are explained in detail in the following sections.

### Presentation Layer

UGameView class is at the base of the Presentation layer. It is responsible for displaying the system parameters and communicating user input to the Business Logic layer. UGameView 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.

[INSERT DIAGRAM HERE]

### Domain Logic Layer

Business Logic layer is mainly characterized by GameTime and Game Controller classes. An instance of GameTime contains all runtime parameters of the system such as Map, Structures, and others. Presentation layer, namely UGameFrame, accesses GameTime to acquire system state and display it, but GameTime does initialize communication with the presentation layer. GameController listens for and handles user input communicated to it through the presentation layer and passes on the information to the GameTime instance that transmits required changes to individual system components. GameController is implemented as a Façade Controller as it is singular and represents the overall system.

[INSERT DIAGRAM HERE]

## 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.

### 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.

1. Principle: Creator

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

Solution: In the current system, GameTime class is responsible for holding all game time parameters of a played game, notably the map it is played on. GameTime also is responsible for delegating runtime system parameters to the Presentation Layer for display. Therefore, GameTime class should initialize an instance of Map upon runtime of the application.

1. 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.

1. 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 structures, critters and tiles. Each individual previously mentioned type is designed to have runtime variations, or subtypes. For example, subtypes implementing the Structure interface, must provide custom overridden definitions of upgrade(), damage() and inspect() methods. 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.

### 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.

[INSERT DIAGRAM]

1. Pattern: Decorator

Problem: How to add behavior to an individual instance of Structure during runtime of the application, without affecting the behavior of other objects of the same type?

Solution: Create a DecoratorStructure class which is a subtype of Structure and in which a ‘decoratee’, an instance of Structure to be decorated, is maintained. When initializing an instance of DecoratorStructure (later referred to as DS), its ‘decoratee’ is defined as an instance of any given Structure or a subtype of Structure. When used during runtime, a DS will have exactly the same behaviour as the decoratee instance, except that it will also perform some extra ‘decorated’ behaviour. This is achieved by following the logical steps described below:

[DS is called to perform a specific method ‘m’ which is common to DS and its decoratee]

1. DS calls its decoratee to perform ‘m’
2. DS performs an extra action as defined by its own implementation of the ‘m’ method

In such a way, functionality is added to a decoratee instance without affecting its basic behaviour.

[INSERT DIAGRAM]

1. Pattern: Observer

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

Solution: (This pattern was not implemented and what follows is an argument against its use)  
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 current system. 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 all 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.

# Dynamic Behaviour: Sequence Diagrams

# User Interface Design

## 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.

## Visual Presentation of Graphical User Interface

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

[INSERT PICTURE]

# 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