**NATIONAL RESEARCH UNIVERSITY HIGHER SCHOOL OF ECONOMICS**

Faculty of Informatics, Mathematics, and Computer Science

PROJECT PROPOSAL

Development of a 2-Dimensional Real-Time Strategy Game

Student: Andrey A. Talanin

Group 18SE

Scientific Advisor: Maxim V. Leykin

NIZHNY NOVGOROD

2022

**Table of Contents**

[Abstract 3](#_Toc96808589)

[Introduction 4](#_Toc96808590)

[Project Overview 6](#_Toc96808591)

[Objectives 6](#_Toc96808592)

[Analysis of Existing Solutions 7](#_Toc96808593)

[Functional and Non-Functional Requirements 8](#_Toc96808594)

[Tools, Frameworks and Libraries 10](#_Toc96808595)

[Contribution of the Project 11](#_Toc96808596)

[Conclusion 12](#_Toc96808597)

# Abstract

This project demonstrates the capabilities of application of enterprise-level development technics in a real-time strategy game. While there are countless books, articles, and online courses on code quality and application design guidelines, many people still have the opinion that these best practices are only applicable to complex commercial software and that following these guidelines in personal projects causes significant additional time costs while providing questionable advantages. Therefore, the main objective of this project is to determine the most common architecture flaws in public open-source solutions and to develop a 2-dimensional real-time strategy game that would highlight the benefits of following the enterprise approach in a personal project, providing an example of how it increases overall project maintainability and extensibility, at the same time simplifying the process of introducing new developers to the existing codebase. The results may prove useful for developers having insufficient experience in object-oriented programming as understanding of various technics, their applications, and use cases is essential for gaining expertise in professional software development.

# Introduction

In the world of software development, it is commonly seen that commercial products tend to have a well-designed architecture that makes it easy to maintain the solution as its complexity grows over time. The majority of such products are also likely to have documentation on user and program interfaces, implementation details, database and workflow diagrams, deployment instructions, etc., as having everything documented significantly simplifies onboarding of new developers and users and ensures knowledge preservation in case someone leaves the team. In most cases, this documentation is arranged and hosted on some Wikipedia-alike internal website providing convenient access to all team members. In contrast, personal projects are nearly the opposite of what was described above: they rarely have any documentation, the source code often has design flaws and poor formatting, making it hard to read and understand. This fact can be partially explained by the amount of time and effort required to keep a non-commercial solution clean and well-documented, which is not always available, especially if it is developed by an individual and not a team, however, there are many features of enterprise development that can be easily implemented in personal projects, drastically increasing the maintainability of a solution, and improving its code quality.

For a professional developer, it is a crucial skill to be able to identify software parts that require extra attention: it may be a piece of program code that is likely to change rapidly according to business requirements, a procedure that requires customization by the client, a module interacting with other software, which is likely to have compatibility issues, etc., and many of these cases are well-known and have standard solutions. Dealing with them properly allows the developer to save time and other resources and also avoid possible issues in the future (which are likely to become more expensive to fix as the development process goes on, with code being written above them, dramatically increasing the number of components related to each particular issue and multiplying places in the code that need to be fixed).

Therefore, this project addresses the problem of applying enterprise-level technics to the needs of personal non-commercial solutions, highlighting points in program code and architecture that are prone to design-level errors, and suggesting solutions that are extensible and easy to maintain. To achieve this, a 2-dimensional real-time strategy game will be developed, as games of this genre contain a lot of entities and subsystems, providing a wide range of applicable architecture technics and design patterns. The developed solution will be hosted in a public repository on GitHub, thereby it will be useful for beginners and developers not familiar with software architecture guidelines, allowing them to implement demonstrated ideas in their personal projects, increasing their extensibility and maintainability.

# Project Overview

## Objectives

Since games of the real-time strategy genre have a quite complex structure and introduce a large number of entities and subsystems, as well as game mechanics, it is required to define the objectives of the project so it will be possible to track progress and not left something out by a mistake:

* Introduce .NET Core hosting extensions to the MonoGame framework

MonoGame is a useful framework when it comes to game development but as one of its goals was to preserve full backward compatibility on binary level with its predecessor, Microsoft XNA, its API remains unchanged since 2012, which makes development much less convenient than it would be if nowadays language features were allowed. The idea is to retarget the MonoGame framework to the latest SDK (Software Development Kit) version and add support for .NET Core generic host and its extensions.

* Implement engine-level systems for handling user input, render graphics user interface, change game states, etc.

Systems of this level should be developed with no binding to any gameplay-related logic, as it will make them more reusable, so they can be utilized in other projects, not necessarily related to games.

* Add gameplay-level entities, implement systems updating the game state and rendering it
* Prepare design documentation giving a top-level view of the project’s architecture
* Prepare a report on points where special development technics were used providing reasoning for the final solution

Once these goals are achieved, it will be possible to count the project as completed.

## Analysis of Existing Solutions

There are a few open-source game projects hosted on GitHub that can be used to understand the approximate scope when developing an application. A brief overview is provided for two of them: Stratagus (part of the Wargus project, see https://github.com/Wargus) and Flare (https://github.com/flareteam/flare‑engine).

Stratagus is a real-time strategy engine written in C++ and published under the GPL 2.0 license, tracking its history almost 12 years back. It was originally developed to reproduce the gameplay of the game of the same genre, Warcraft II: Tides of Darkness (released by Blizzard Entertainment in 1995), on Linux, as the original game was using Windows-only graphics libraries. At some point, the developers took out the engine code to a standalone project, removing bindings to the specific game and making it reusable and customizable. However, while the project demonstrates many interesting gameplay mechanics, its source code contains a lot of anti-patterns and bad design decisions (global variables, non-static classes that have no state, etc.)

Flare is also a game engine, but it represents another genre: fantasy action role-playing games (Diablo series, Path of Exile). Just like the previous one, it is written in C++ and published under GPL 3.0 license, but also permits later versions. This project does not have any interesting story behind it; however, it still implements a lot of mechanics like character equipment, inventory, spells, and in-game events. Unfortunately, it has all the same issues: its codebase varies widely in both age and quality, introducing different code styles and questionable solutions.

It is worth noticing that there are also a lot of other game engines, but these two are the closest ones to the game being developed as a part of the current project from the perspective of game mechanics implemented in them. Moreover, design flaws that can be found in them are widely found so they can be used to illustrate decisions to avoid when developing an application.

## Functional and Non-Functional Requirements

In order to represent a complete product, the game to be developed should meet the functional and non-functional requirements that will be listed in this section. The list of entities existing in the system must include the following:

* Game map

A game map represents a rectangular area divided into multiple cells having different landscape types affecting gameplay (for instance, units can walk on plains, but not on water).

* Units

A unit represents an atomic entity that can be controlled by the player. It can move, fight, gather and deliver resources, construct and repair buildings, depending on its type.

* Buildings

Buildings are entities that have economical value: they can produce units, research upgrades, provide food, which defines the number of units a player can have at the same time. Buildings can be constructed by units, repaired if damaged, and serve as a resource delivery point.

* Players

A player is an entity containing information describing a human or computer-controlled faction. It includes name, flag color, researched upgrades, available resources, and game statistics.

There is also a list of systems responsible for processing the game’s state and changing it according to the game mechanics that should be implemented so the game is playable:

* Movement and path generation system

The system will be responsible for generating a path between two points on the game map, taking into account various obstacles on the way including terrain, units, and buildings.

* Resource gathering system

The resource gathering system will manage extracting resources from map decorations (trees) and buildings (gold mines), delivering them to stockpiles, and storing them as a part of the player’s data.

* Building and repairing system

The system will handle orders related to repairing existing buildings and placing new construction sites, track progress on construction, and replace construction sites with actual buildings.

* Production queue system

As buildings of some types have available upgrades and units, there will be a system processing production queues and managing orders related to them.

* Combat system

The combat system will process fights between units and damage done to buildings. Each unit has limited heath (buildings have limited durability), so when its value reaches 0, a unit is killed, and a building is destroyed. When out of combat, the unit’s health regenerates over time, and the durability of a building can be restored by repairing it.

* Victory and defeat conditions

This system will check for multiple conditions indicating the end of the game. For instance, losing all buildings counts as a defeat condition.

* Options to save a game or to load an existing one

There will be options to save the current game or to resume a previously saved one, as game sessions may take several hours, and players should not lose their progress if they have to exit the game.

Finally, there is a list of implementation-level requirements:

* Event-based system for handling user input
* Graphics user interface system for displaying windows and controls
* Game state management system
* Viewport-based world rendering and 2D camera
* Controllers for propagating user commands to game-level services
* Order management system supporting order queues
* In-memory repositories for storing game state

As noticed before, a real-time strategy game is a complex system having various entities and subsystems. In terms of source code, it leads to a large number of types and relations between them, requiring careful architecture planning. Therefore, it will be possible to use implemented solutions as examples of solving different design-level problems that other developers are likely to face in their personal projects.

## Tools, Frameworks and Libraries

The following tools and frameworks will be used in the development:

* The C# programming language

C# is an object-oriented programming language with a rich base class library, supporting modern mechanisms like reflection and runtime module loading, which make it a perfect tool for developing a complex system with a large number of entities.

* .NET Core hosting extensions

As C# is an enterprise-level programming language, it is widely used in commercial development. Over the past 20 years of its history, it accumulated a lot of extensions providing convenient API-s for logging, configuration management, dependency injection, etc. Using these extensions greatly saves time resources on writing repeated code.

* MonoGame

MonoGame is an open-source framework that was developed as a successor to the Microsoft XNA framework (discontinued in 2012). It provides cross-platform graphics, sound, and input API-s, serving as a managed wrapper for native platform-specific libraries like Microsoft DirectX and OpenGL.

* Microsoft Visual Studio, an integrated development environment
* Git, a version control system

## Contribution of the Project

While the project does not have any commercial value, with it being published in a public GitHub repository, it may be found useful by other developers, as it will provide examples of multiple developing technics and solutions for typical design problems. Moreover, as Git tracks all changes in the project from the moment of its creation, the current project’s history may help to acquire skills of planning the development process, setting up priorities, dividing requirements into critical and optional ones, etc.

# Conclusion

To sum up, there are many features in the world of enterprise software development that can be implemented by anyone in their personal projects, increasing code quality, solution’s extensibility, and maintainability. While using such technics may slightly add to the amount of time that needs to be spent on development, it will make application components more reusable and will help to avoid long-running problems that are costly to fix.

The results of this project will be published on GitHub under a permissive license so other developers will be able to browse and reuse the source code. Besides the code itself, it will also contain design documentation and component and class diagrams allowing to get a top-level view of the project and easily find implementation details of the required component or subsystem. Therefore, this project will be useful for people interested in game development or using .NET Core hosting extensions for everyday needs, as it will provide examples for many use cases of the most commonly used framework features.

Furthermore, the development of the game application will not stop once the goals of this academic project are completed: new features and game mechanics will be added, so it will eventually become possible to use the achievements of the current project as a base for a commercial product.