Assignment of bachelor's thesis

Title:	FitLife – game about studying at FIT
Student:	Duc Minh Pham
Supervisor:	Ing. Radek Richtr, Ph.D.
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2022/2023

Instructions

Fitlife is a game project about studying at the faculty. The goal of the bachelor's thesis is to create a software engineering design, GDD and refactoring of the previous state of FitLife. On the graphical side, work with graphics designer David Mikulka.

1) Summarize the game plan in a game design document.
2) Analyze games of similar range and theme.
3) Create a complete SE design
4) Based on the prototype of the game, and its user testing perform refactoring and UX changes
5) Sufficiently test the game.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Bachelor's thesis

FitLife - a game about studying at FIT

Pham Minh Duc

Department of Software Engineering
Supervisor: Ing. Radek Richtr, Ph.D.

May 11, 2022

Acknowledgements

I would like to thank Ing. Radek Richtr, Ph.D. for his guidance during the creation of this thesis, my thanks also go to my family, who supported me during my studies.

Declaration

Citation of this thesis

Pham, Minh Duc. *FitLife - a game about studying at FIT.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Cílem této práce je navrhnout a dokončit hru FitLife. Na procesu finalizace hry se podílejí dva lidé, a proto bude proces rozdělen do dvou bakalářských prací. Tato práce se zaměří na softwarově inženýrský aspekt vývoje hry. Při návrhu a implementaci klíčových back-endových komponent hry budou použity návrhové vzory.

Klíčová slova Unity, 2D-RPG Hra, UML, Návrhové vzory

Abstract

The goal of this thesis is to design and finalize the game of FitLife. The game is being finalized by two people, and thus the process be split into two theses. This thesis will focus on the software engineering aspect of game development. Design patterns will be used in the design and implementation of key back-end components of the game.

Keywords Unity, 2D-RPG Game, UML, Design patterns

Contents

Introduction **1**

The main goal **3**

I Analysis **5**

1 Video game analysis **7**
1.1 World representation 7
1.2 Camera types 8
1.3 Genre 8
1.3.1 Simulation 8
1.3.2 Adventure 9
1.4 Analysis of similar games 9
1.4.1 Stardew Valley 10
1.4.2 Pokémon Diamond and Pearl 12
1.4.3 Undertale 14
1.4.4 Octopath Traveler 15
1.5 Game analysis conclusion 17

2 Design patterns **19**
2.1 What is a design pattern 19
2.2 Design pattern use in Video games 20
2.3 Observer pattern 20
2.4 Decorator pattern 21
2.5 Singleton pattern 21
2.6 Service locator pattern 22
2.7 State machine pattern 23

3 FitLife analysis - prequel **25**
3.1 User testing analysis 26
3.2 User interface 27
3.3 Game timer 27
3.4 Quest system 28

3.5 Player status 28
3.6 Audio manager 28
3.7 Interactions 29

II Design 31

4 Design of FitLife 33
4.1 General game design 33
4.2 Game managers 35
4.2.1 Status controller 35
4.2.2 Quest system 36
4.2.3 Game timer 38
4.2.4 Player status 40
4.2.5 Interaction tracker 42
4.2.6 Interactive 42

5 Game design document 45
5.1 Introduction 45
5.2 Target System 45
5.3 Development System 45
5.4 Target Audience 46
5.5 Specification 46
5.6 Gameplay 48
5.6.1 Story 48
5.6.2 Stats 49
5.6.3 Grading system 49
5.6.4 Quests 50
5.6.5 Characters 51
5.6.6 Level design 53
5.6.7 Music and effects 53
5.6.8 In-game controls 54
5.6.9 User interface 54
5.6.10 In-game pause menu 55
5.7 First launch 56

III Realisation 57

6 Development 59
6.1 Used unity features 59
6.1.1 Canvas components 60
6.2 UI scaling 61
6.3 Camera scaling 61

6.4 Game managers 63
6.5 Implementation curiosities 68

7 User testing 71
7.1 Testing description 71
7.1.1 The questionnaire questions 71
7.2 Testing conclusions 73

Conclusion 75

Bibliography 77

A Acronyms 79

B Contents of enclosed SD-Card 81

List of Figures

1.1 Types of third person cameras 9
1.2 Stardew Valley 10
1.3 Pokémon Diamond 12
1.4 Undertale 14
1.5 Octopath Traveler 16

2.1 Observer pattern 20
2.2 Decorator pattern 21
2.3 Singleton pattern 22
2.4 Service locator pattern 22
2.5 State machine pattern 23

4.1 Functional and Non-functional requirements of FitLife 33
4.2 Functional and Non-functional requirements of game managers . . 34
4.3 Status controller class diagram 35
4.4 Quest tracker class diagram 36
4.5 Quest tracker activity diagram 37
4.6 Game timer class diagram 38
4.7 Game timer activity diagram 39
4.8 Player status class diagram 40
4.9 Player status activity diagram 41
4.10 Interaction tracker class diagram 43
4.11 Interaction tracker activity diagram 43

5.1 Game activity diagram 47
5.2 The player, during one of the lectures 48
5.3 ZMA exam minigame 49
5.4 Main character 51
5.5 Level design 52
5.6 UI layout - wireframe 55
5.7 Day and Night cycle 56

6.1 Text window 61
6.2 Expanded text window 61
6.3 Phone 64

6.4 Hint window 65
6.5 Status change event 66
6.6 Quest menus 68

List of Tables

3.1 The people who participated on the project 25

List of Code listings

6.1 Camera resizing calculation 62
6.2 Camera follow calculation 62
6.3 Status controller - Service Locator 63
6.4 Game Timer - Observer pattern 64
6.5 Interaction tracker - dialogue history 65
6.6 Dialogue hashing 65
6.7 Player stats management 66
6.8 Quest tracker 67
6.9 School quest 67
6.10 Scratching and typing animation effect 69
6.11 Coroutine queue 70

Introduction

Games have been a prevalent part of many people's childhoods. We play games to pass the time, make friends or learn something new. We will primarily focus on computer games, which are played on a Personal computer (PC). These digital programs are platforms with unbelievable potential for story-telling and entertainment. The player experiences a world crafted to perfection by teams of developers of various sizes and experiences, each adding a subtle touch to the final product.

I was a part of such a team during SP1 and SP2 subjects, where we developed a fun simulator game called FitLife. The game did not end up as we expected, so me and my colleague David Mikulka [1] took it upon ourselves to finish the game as part of our Bacherlor's thesis.

The motivation for this work is to create a game set in the scenery of Czech Technical University (CTU) campus. Simulating how a student feels in his first semester of school at Faculty of Information Technology (FIT) CTU.

The main goal

This thesis focuses on the design and finalization of the game FitLife. As part of this goal, we will demonstrate how design patterns allow us to develop more stable and extensible code. The game is developed in Unity and will target the WebGL platform.

Design patterns will be used in the designing and implementation of critical components that drive the inner workings of the game.

The game has been in creation for one year and has been in the hands of several developers. It has been user-tested after this period, and the feedback will be used to improve the game.

There are many obstacles along the way, as many game components need to be reworked.

Part I

Analysis

CHAPTER 1

Video game analysis

This section is dedicated to discovering various aspects of computer games, about what makes the games fun for the end-user and therefore successful in the vast selection of games. First, we will categorize video games by their genre-defining features, and then we will take inspiration from some specific games and apply them to FitLife.

1.1 World representation

There are two main types of world representation: 2D and 3D. A 2D world consists of objects which are flat and drawn using flat images[*], 3D world objects are defined using materials and 3D modeled objects onto which the materials are rendered. There is a vast difference in scope and system requirements between 2D and 3D projects.

- *2D* - a simple representation of the game world, defined using the x and y-axis. Typically used in platform games and top-down games
- *3D* - with the addition of the z-axis, object representation and physics become more complicated. It becomes very demanding on the end user's machine but allows for a more immersive experience.
- *2D with perspective* - using 2D with depth projection, a sort of 2.5D is achieved, combining graphic simplicity and depth perception.

[*]In Unity these objects are called Sprites

1.2 Camera types

Cameras are objects through which the player sees the game world. There are many variations in the implementation of cameras. We will focus on the most relevant for 2D development.

- *First-person* - The camera is in the place of the character's eyes/chest. This type of camera is impossible for 2D games
- *Third-person* - Detached from the body of the main character, the camera usually views the character itself and its close proximity
  - *Side view* (Figure 1.1a) - This type of camera is typically used for platforming games or fighting games. All objects are viewed from the side, and there is little to no depth perceived.
  - *Isometric projection* (Figure 1.1c) - This atypical view allows for 3D representation of objects. All axes are the same length and angle from each other (120° degree angle).
  - *3/4 view* (Figure 1.1b) - A mix of the above two camera types, where both the top and side textures are shown. This type was used for many games in the 90s

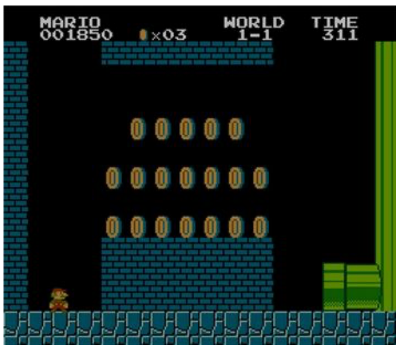
1.3 Genre

Game genres group games into categories with similar mechanics and core gameplay. There are a considerable number of genres circulating the industry like shooters, puzzlers, platformers, multiplayer battle arena games, sports games, and fighters. We will be focusing on Simulation games and Adventure games as they are most relevant to FitLife.

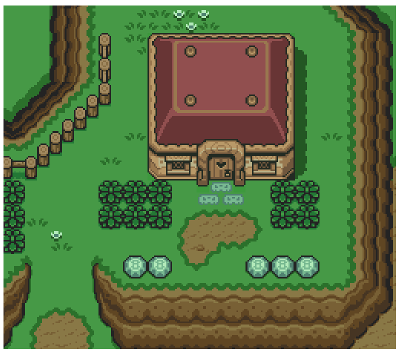
Some games are so revolutionary in their gameplay that they introduce a whole new subgenre. There are examples like Soulslike games, which are inspired by the game Dark Souls, or Roguelike, inspired by the game Rogue.

1.3.1 Simulation

Simulation games are unique that they cannot be represented by a single way the game should be played. The developers set out to simulate a real or fictional world in which the player is free to roam around, explore and complete objectives. Simulation games often have difficulties keeping the player interested in exploring all aspects of the game. Simplification and retracting of complicated concepts are crucial to keeping the game accessible to more audiences.



(a) Side view [2]


(b) 3/4 view [3]

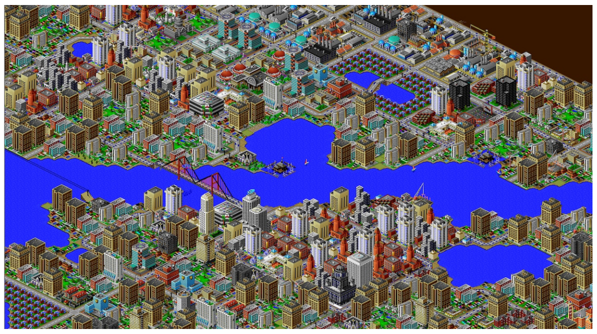

(c) Isometric projection [4]

Figure 1.1: Types of third person cameras

1.3.2 Adventure

The adventure games are story-oriented. The player is part of a grand adventure with an exciting story completed by the end of the game. The player is often portrayed as the main protagonist battling evil and destroying it by the end. This genre is prevalent, and for a good reason. There are parallels drawn to literature and movies with enormous appeal and massive audiences. After reading a book or watching a movie, experiencing a game in the same world is often sought after.

Non player characters are also important as they are a vital part of the adventure game genre. NPCs are characters whose actions cannot be controlled by the player. They are in the game to interact with players: participating in banter, fighting, or giving the player quests to complete.

1.4 Analysis of similar games

FitLife will be taking inspiration from games in the similar genre of 2D 3/4 view Role playing game (RPG) simulator/adventure games.

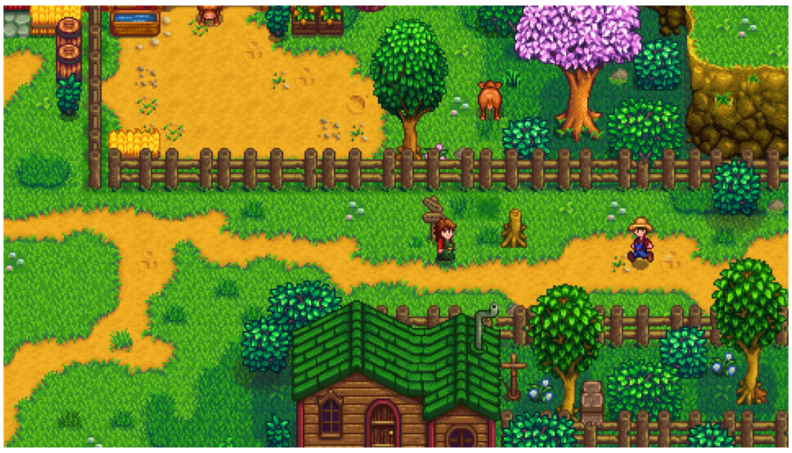


Figure 1.2: Stardew Valley [5]

1.4.1 Stardew Valley

Stardew Valley is a third-person 2D simulator role-playing game, it is an homage to “Harvest moon,” a casual farming simulator. The player begins his adventure as a new citizen of a small town named Stardew Valley, who has inherited a vast farm from his grandfather. The player is then free to explore his newly acquired land and the nearby village. In doing so, he discovers that the village is lively and active. There are many hidden interactions for the discover.

There are different environments the player can be found in. The farm is where the player can interact with the surroundings the most; the other areas consist of the cave and village, which are more limited. The farming aspect is deep and well developed. The player is pushed to expand and upgrade/fix his gear and buildings. This doubles as a sort of organic achievement system, giving the player a sense of accomplishment.

NPCs

The game features around 46 NPCs, each with its own story. In the town center, we find a calendar that shows each NPC’s birthdays. Exploring the town and talking to them builds relationships between the player and the characters. They offer a range of quests for which the player is rewarded with items that help him grow his farm. Further interactions allow the player to continue relationship building, allowing marriage to available bachelors/bachelorettes.

Defining characteristics

- *Engaging NPC characters*
- *Pixel art* - This type of art fits the game well, it is expressive and colorful. The colors change with the season, and it is portrayed well in the game.
- *Farming* - This is the game's defining feature and main mechanic. Taking care of the farm is intricate and time-consuming but somehow addicting and relaxing.
- *Time* - The game portrays time in an interesting way. Keeping time is quite important for the gameplay, the town's people have a schedule that dictates if they are available to the player or not. The people are sleeping early in the morning and are also not available at night time.

Figure 1.3: Pokémon Diamond [6]

1.4.2 Pokémon Diamond and Pearl

A third-person RPG game set in a fictional universe shared with an anime TV Series with the same name. Pokémon Diamond and its Pearl version are iconic games released for the Nintendo DS. Pokémon games are often released in two versions, each having exclusive Pokémon that cannot be found in the other version. Diamond and Pearl are located in the fictional region of Sinnoh.

The games are iconic and (except for graphical improvement) do not differ much from previous generations. A turn-based battle system with collection of a huge amount of Pokémon as the core of the gameplay. This template has been successful in every generation bringing new and old nostalgia-driven* players to buy, making them wildly popular with both younger and older audiences.

The fighting system between trainers is engaging and involved. Pokémon have various classes and have synergies with each other. Often the rarer the pokémon are, the stronger they are. Leveling the pokémon also increases their strength. After a threshold, they evolve into stronger versions, establishing natural player progression without being too repetitive.

*Many returning players have played the game before at a younger age, returning to the same type of game brings nostalgia.

NPCs

There are several types of characters, varying from friendly to villainous. Players are offered trading, battles, or gifts from various characters living in Sinnoh. The vast majority of NPC interactions lead to pokémon battles. Pokémon battles are turn-based, using the collected pokémons to defeat the opponent's pokémon.

Hidden themes often occur when traveling the world, e.g., the trainer in a rocky mountain has rocky pokémon, and a timid trainer owns a timid pokémon. This leads to the concept of regionalism [7] and the game's connection to Japan.

Defining characteristics

What is important is the gameplay and interactions with wild Pokémon and Pokémon trainers. The player learns about the world and becomes closer to the character through conversations with other NPCs. The NPC interactions are brief and text-based. The soothing music and sound effects bring this game to life.

The game story is quite simple, yet because the Pokémon games are so iconic, they spawned an anime television series (that is extremely popular), which compensates in this aspect.

- *Simple story*
- *Interesting mechanics* - Wild pokémon hunting encourages exploration of the whole world multiple times. The battling system brings strategic and engaging gameplay.
- *Huge world* - The world is massive yet does not feel empty due to random encounters with Pokémon and trainers.

Figure 1.4: Undertale [8]

1.4.3 Undertale

A simple-looking game with a surprising amount of plot and story. Undertale is a 2D adventure role-playing game with pixel graphics. The player controls a child who finds herself in the Underground. The player is tasked with escaping this place, but many monsters stand in the way. During fights, the player controls a small heart moving inside a square playing area. When encountering these monsters, there are choices to be made, either defeating them via dealing damage resulting in killing them or finding ways to avoid killing them and sparing them. These actions affect the ending of the game in a significant way.

There are several endings based on the player's actions or the lack of them. The game has many easter eggs hidden all over the game world. Some even require editing game files to find. Finding these will not reward the player with much more than a good feeling, but they serve to further expand the complexity of the world.

NPCs

The NPCs have personalities and emotions, which are portrayed during the combat and conversation. Ultimately they [9] evoke thoughts about responsibility and raise the player's awareness of violence and its consequences. In the end, the player is judged for his actions toward other characters. The encounters are turn-based and are similar to Pokémon.

Defining characteristics

Undertale's graphics are deceivingly simple, but it makes up for it in the story and a relatively large world. Undertale includes many easter eggs, while exploring the world, the player is rewarded with the satisfaction of finding these objects. These Easter eggs are often hard to find but entertaining nonetheless.

- *Simple graphics* - Not a downside, but rather a characteristic feature of the game.
- *Emotion-provoking story* - The best part of the game, the player chooses to act a certain way throughout the game and is judged at the end.
- *Memorable characters* - Each character has a unique style. The game perfectly portrays the emotions of the characters with visual effects, fighting style and sound effects.

1.4.4 Octopath Traveler

Octopath Traveler is an open-world RPG game released by Square Enix. Inspired by retro 2D graphics, Octopath traveler brings a unique take on the 2D adventure genre, sporting retro pixel-art sprites mixed with HD special effects. The player is put into the shoes of one of the eight main characters, each with their own story. The player is given complete freedom to roam the vast world of Orsterra, to discover the stories of its inhabitants. The personalities of each of the travelers are important. They develop the character as the player learns more about their backstory. There are many hidden artifacts throughout the game, encouraging the player to push through challenging environments and fight through the enemies to perhaps discover a secret, boosting their favorite characters' powers.

The gameplay revolves around its turn-based battle system. There are complex synergies and interactions between party members, chosen by the player.

NPCs

The main interactions happen between the travelers and NPCs from their storylines. Each of the characters has a special "Path Action" The travelers do not participate in banter during the long enemy-ridden trips between zones.

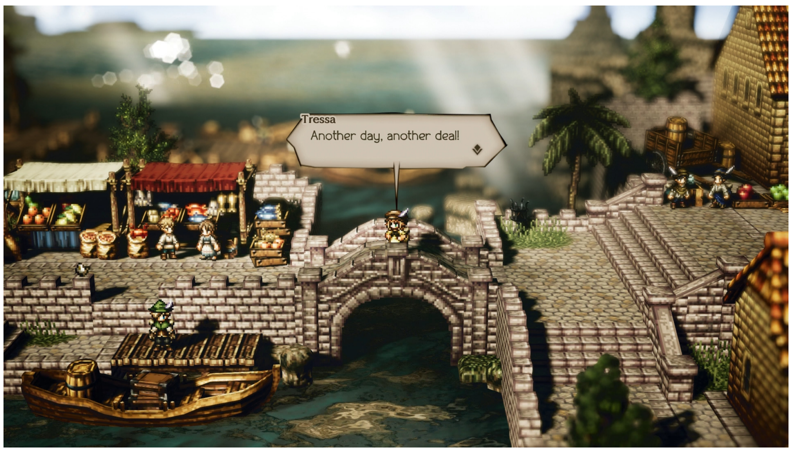


Figure 1.5: Octopath Traveler [10]

Defining characteristics

Beautiful graphics and animations, extensive world, and length of story. Octopath Traveler explores the retro 2D with new innovative style. Sadly the combat system becomes repetitive a while into the game.

- *Travel freedom* - The massive world is open and free to explore
- *Character uniqueness* - The main eight characters have very different unique personalities, and it is easy to connect with one of the travelers
- *Beautiful graphics* - The game is beautiful, it is the perfect synergy of retro gameplay and current graphics

1.5 Game analysis conclusion

FitLife will carry inspiration from each of the mentioned games. Specifically:

- *Importance of time of day (Stardew Valley)* (Section 1.4.1)

  The days are dynamic and the lighting changes throughout the day/night cycle. Events driven by the time of day, make sure that the player pays attention to what time it is. The player is punished for staying outside too long.

- *World exploration (Pokémon)* (Section 1.4.2)

  Pokémon manages to organically motivate the player to explore the world by rewarding them with special Pokémon that live there. Not only that but the world is detailed and fun to explore.

- *Easter eggs and interesting dialogue (Undertale)* (Section 1.4.3)

  There are many easter eggs in Undertale, most do not add much in terms of gameplay advantage. They are like hidden achievements which the player can find in obscure places. It gives the player more to learn from the world.

- *Adding animations for a lively game (Octopath Traveler)* (Section 1.4.4)

  Octopath Traveler is a beautiful game, it has the retro pixel art style, but combines it with 3D effects, and the 2.5D world ties it all together.

CHAPTER 2

Design patterns

This chapter is dedicated to the research of Design Patterns, describing their use cases including their benefits and pitfalls of using such design patterns.

> “A well designed game programming that offer great flexibility, code reusability, extensibility, and low maintenance costs is highly desired.“
>
> (J. Qu, Y. Song and Y. Wei — Design patterns applied for game design patterns [11])

2.1 What is a design pattern

> “Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice. “
>
> (Christopher Alexander — Design patterns elements of Reusable Object Oriented Software [12])

Design patterns are by nature, universal. They are not bound by a language or specific technology, rather they describe a concept that is widely adopted and defines a ”best practice”* way of approaching it. Using design patterns is not only recommended but in some cases absolutely vital for correct implementation.

The patterns are divided into three categories: Creational, Structural and Behavioral. Mentioned will be some, but not all design patterns that exist.

*A term used in the industry, describing something that is best done in that one for various reasons, be it readability or functionality.

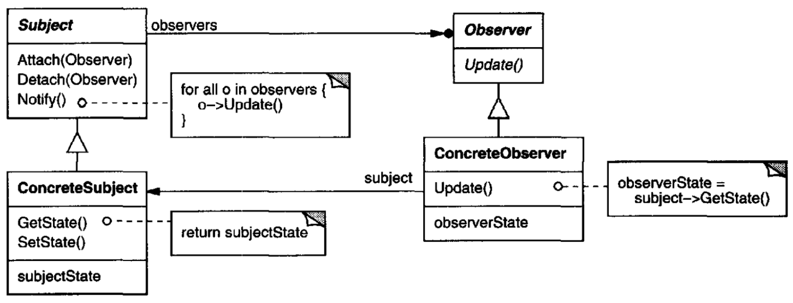


Figure 2.1: Observer pattern

2.2 Design pattern use in Video games

Based on this article [13] design patterns find successful use in video game development. The article showcases three different games of different genres developed in Unity. The developers found different design patterns useful for different types of games. The one used most was the singleton pattern.

Another article focusing on FPS games [14] uses a wide selection of design patterns like Flyweight, State machine, Singleton, Decorator, and Observer patterns to solve various aspects of the game.

2.3 Observer pattern

Observer patterns (Figure 2.1) are used in one-to-many scenarios. When one object changes state, all of its subscribers are notified and updated accordingly [12].

Use the pattern when changes in one object require changes in the observer. This pattern allows for any number of observers without coupling with the subject. There might even not be any observers.

The concept of loose coupling and minimizing state dependencies is brilliantly explained in article [15]. The observer pattern is a powerful tool in software design, often many components are tightly coupled, but in fact, do not need to be. Where User interface (UI) elements periodically poll for any changes in the managing class, they can often be replaced with the observer pattern. As a subscriber, the UI element waits for the subscribed event to be invoked. After invocation the subscriber polls for changes made in the managing object. This saves countless CPU cycles that would be wasted on checking whether the state had changed.

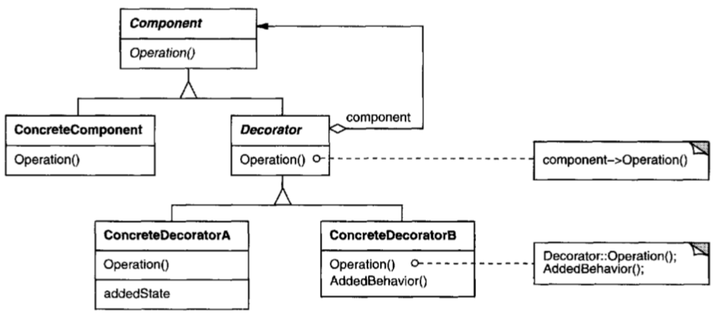


Figure 2.2: Decorator pattern [12]

2.4 Decorator pattern

A decorator pattern (Figure 2.2) is used to extend subclassing, allowing dynamically attaching new responsibilities, also known as a wrapper. It allows for defined objects to have additional functionality based on the decorator.

As an example, a player object is wrapped with a weapon decorator, then wrapped with a shield decorator. The decorators do not interact with each other, yet they modify the behavior of the underlying class. Decorators allow further functionality for the object. The decorators can be stacked indefinitely.

Used when subclassing needs to be reduced as a subclass needs to be created for all combinations of decorated objects. Allows for abstract extensions to objects of a concrete type that can be removed.

2.5 Singleton pattern

A Singleton is an object that restricts its creation to a single instance. This is both useful and restrictive in some way. They are used for the game controller and persistent aspects between scenes, like time, quests, and music. The class itself maintains the single instance while providing an access point for it and disrupts the creation of another.

In applied game development, it is essential to distinguish between correct uses of this pattern, a player can be set as a singleton, but this would limit the game to only one player. This pattern must then be modified as a player-manager – the singleton instance, managing the player instances.

Singleton is used when only one instance of the object must exist. Global access to the object is required.

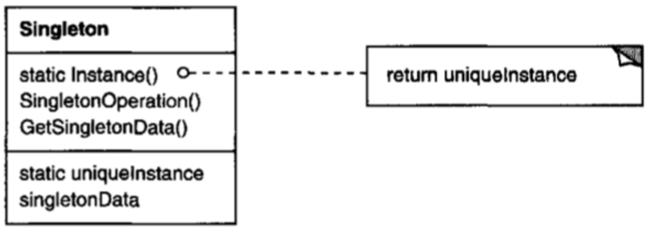


Figure 2.3: Singleton pattern [12]

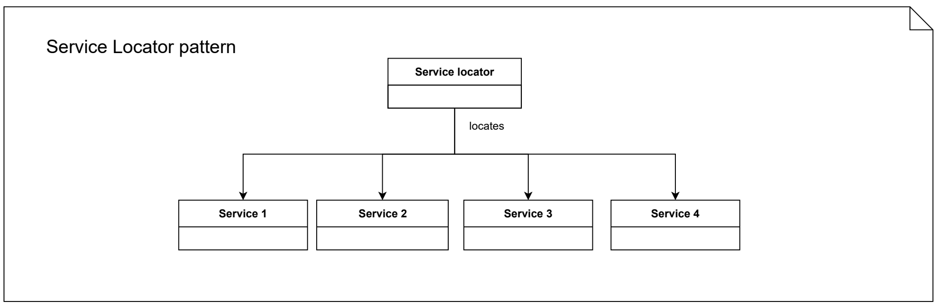


Figure 2.4: Service locator pattern [16]

2.6 Service locator pattern

The service locator pattern can be used as an extension [16] of the singleton pattern (Section 2.5). This pattern is especially useful in game development as there are many managing objects that need to be accessed by many different scripts, based on this article [16] a service locator can be used as a gateway singleton to manage access to multiple other game systems. This allows for simpler object management and further extensibility.

The service locator can act as the sole singleton object needed in the game. A kind of registry keeping track of other ”singleton” objects, without needing the singleton code to be repeated. It allows for lazy instantiation of objects required later in the game, reducing the load on the CPU during startup.

Used when many singleton objects are required and their access maintained, like sound managers, player managers, scene managers, etc. Which are then accessible through the registry.

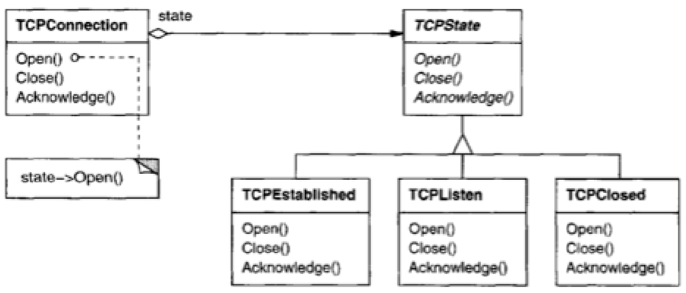


Figure 2.5: State machine pattern [12]

2.7 State machine pattern

State machine pattern (Figure 2.5) is used on objects that act differently based on its current state.

This is often done using polymorphism and defining different behavior for each state. Use this pattern when an object's behavior depends on its state, or when large conditional statements are required to check for the current state of the object [12].

Unity uses developer-defined state machines for animation handling. Based on which state the state machine hierarchy is in, that animation is played or transitioned.

CHAPTER 3

FitLife analysis - prequel

The game has been in the making for one year. It was a result of subjects SP1 and SP2. Eight students have worked on the project, every person, who participated in developing the game is listed in table3.1.

Here we find ourselves at the beginning of the scope of this thesis. We will be looking into User experience (UX) issues that people encountered during their time with the game. We will be discussing some of the shortcomings the game had at this stage.

BI-SP1	BI-SP2
Ing. Skotnica Marek (Teacher)	Ing. Skotnica Marek (Teacher)
Pham Minh Duc (myself)	Pham Minh Duc (myself)
David Mikulka	David Mikulka
Lukáš Jílek	Lukáš Jílek
Pham Lan Phuong	Josef Havelka
Nguyen Xuan Thang	
Dominik Hulina	
Nguyen Quynh Chi	

Table 3.1: The people who participated on the project

Most of us did not even know each other before the project inception. The project started in the third semester of our studies and was led by Ing. Skotnica Marek. We wanted a 2D game with pixel art, like Pokémon. A couple of weeks later, our goal was set: develop a game that reflects how a FIT student feels in the first semester of their studies. Most of the visual part of the game is covered by my colleague David Mikulka's Bachelor's thesis [1].

FitLife was our first big project, and it showed in many lousy development practices. The code was unstable, unsustainable, and a pain to change. However, this was to be expected as we were inexperienced. I will not be blaming anyone in the following critique and subsequent reworking of the workflow and refactoring of the code.

3.1 User testing analysis

After SP1 and SP2 we held a limited user testing period in which we sent the game to our friends. We got some feedback on things that do not quite work correctly or need to be changed.

Around the beginning of 2022, we held another period of user testing which happened in an online group of game developers of CTU FIT. The user testing involved a questionnaire which we will provide a summary of:

- *Can the game be run in a browser?*

  Answer summary: No problems.

- *Are the controls intuitive and understandable?*

  Answer summary: Yes, after reading the tutorial, there were no problems.

- *Is the UI designed well?*

  Answer summary: Mostly yes, a person with a 4k monitor complained that the game did not scale

- *Was the goal of the game communicated well?*

  Answer summary: The goal was not clear to anyone. There was little to no way to know where to be and what to do to finish the game successfully.

- *Were you able to reach the minigames (NTK, ZMA & CAO exams)?*

  Answer summary: There were problems with quest completion, quests did not match with time. The players had trouble with a quest that was not fully implemented.

- *How would you improve the game? (Pluses/minuses)*

  Pluses summary: graphics, idea, controls, soundtrack and dialogues

  Minuses summary: there is a problem with waiting on lectures and exams, big problem with time, broken quest system which needs individual quest activation, highlighting of some quest objects which are available is needed

3.2 User interface

The User Interface has several issues. First, the interface was not scalable and would break if used with a different resolution or screen ratio. This would sometimes lead to unplayable sequences where a text box or hint window covered the game. At the latter stages of the development, this was circumvented by forcing a 16:9 ratio, disabling resizing of the game window, and disabling fullscreen mode.

Secondly, it is fragmented. There were different instances of the UI elements in different scenes. Changes made to UI elements in one scene would not propagate to others, even though they act the same in the background. There is a dependency between the presentation layer (front-end) and the business layer (back-end). Changes in objects in the UI elements would lead to errors in the scripts keeping track of the game's status. This would discourage team members from changing visual aspects of the game.

3.3 Game timer

The game timer is unstable. Here UI and Business layer (back end) are entirely disconnected. The player sees that the time is at 9:00, but in reality, the game is led by its internal clock, which is different from the one viewed. The problem would occur when one timer is stopped, but the other is not. Keeping track of the game time is an essential part of the game, yet it is very difficult working with this component. Many events of the game are driven by the time of day[*]. The timer is also tightly coupled with the Player Status component even though they should work fine without one another.

The time was handled with four integers: Days, Hours, Minutes, and Seconds. This is unwanted in many ways as the component is cluttered with unnecessary time computation.

[*]Like quests mentioned in section 5.6.4, day/time scene changes, and sleep management

3.4 Quest system

The Quest system requires the player to manually activate each quest, which are described in section 5.6.4, with a button before completing it, or the completion would not be logged. The quests do not have any feedback when activated, completed, or failed. The quests are kept track of at the NPC/objects which give them out, making it very difficult to add or change quests as the change needs to be handled in many places. It should observe the internal clock and update itself based on the quest's time windows and requirements.

The quest objectives are vague and do not give feedback on when the quest could be completed. This is especially important for school quests which require to be present in a tight window of time to be completed successfully.

The quest window is also tightly coupled with the quest manager class. This would, in some cases, throw null exceptions when changing scenes with the window open.

3.5 Player status

This is an important aspect of the game. Reaching one of the stats to zero results in an end game condition. But there is no indication what action changes the values of these stats or when the stat is added/subtracted. Because the stats, described in section 5.6.2, are sort of hidden[*] There is also no warning when the stats become low, so the player needs to make an effort to replenish them. There are three stats: energy and hunger decay with time, but the social status does not change whatsoever. Neither does it increases or decrease throughout the game.

3.6 Audio manager

This is handled during the first visit to the main menu. An object is created that persists through the whole game. Sadly, it is not used for other sounds that happen in the game. After ending the game and returning to the main menu, a new object is created, creating two instances of the same object, potentially playing the music twice. The music control is also lost after returning to the main menu.

The main theme is quite long, yet after leaving scenes with different music, the main theme starts from the beginning.

[*]Requiring an action (opening the phone), makes the stats more obscure than they should be.

3.7 Interactions

There is no interaction history implemented, meaning dialogues can be repeated if the player leaves the scene and comes back. This breaks the immersion and does not make much sense.

When an interaction or dialogue is available, a window with a hint is rendered upon the screen, showing when an interaction is available to be accessed.

Part II

Design

CHAPTER 4

Design of FitLife

We will be processing feedback based on the previous version of the game, designing improved components which are more stable while fixing the UX issues.

Based on the research surrounding design patterns, we shall design the game's troublesome components with their help. We will be using an activity diagram, class diagram and functional/non-functional requirements to present them. In the realization phase, we will be implementing them into the game.

4.1 General game design

The general game design is not what this thesis focuses on, rather this aspect is explored in Mikulka's thesis [1]. But there are some functional general UX issues that belong in this category and are in the scope of this thesis, particularly listed in figure 4.6.

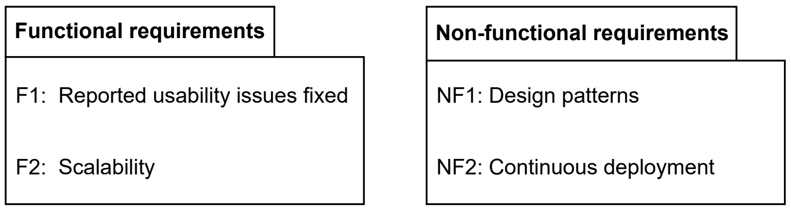

Functional requirements
F1: Reported usability issues fixed
F2: Scalability
Non-functional requirements
NF1: Design patterns
NF2: Continuous deployment

Figure 4.1: Functional and Non-functional requirements of FitLife

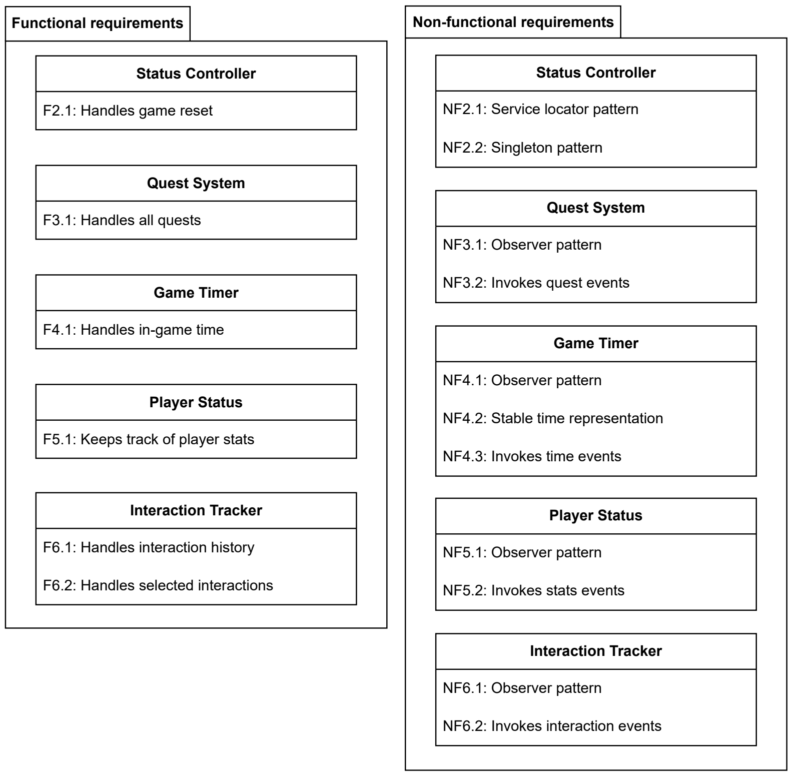

Functional requirements
Status Controller
F2.1: Handles game reset
Quest System
F3.1: Handles all quests
Game Timer
F4.1: Handles in-game time
Player Status
F5.1: Keeps track of player stats
Interaction Tracker
F6.1: Handles interaction history
F6.2: Handles selected interactions
Non-functional requirements
Status Controller
NF2.1: Service locator pattern
NF2.2: Singleton pattern
Quest System
NF3.1: Observer pattern
NF3.2: Invokes quest events
Game Timer
NF4.1: Observer pattern
NF4.2: Stable time representation
NF4.3: Invokes time events
Player Status
NF5.1: Observer pattern
NF5.2: Invokes stats events
Interaction Tracker
NF6.1: Observer pattern
NF6.2: Invokes interaction events

Figure 4.2: Functional and Non-functional requirements of game managers

F1: Reported usability issues fixed

Gameplay problems from the received feedback will be fixed by implementing a more stable back-end, that is flexible and modular.

F2: User interface scalability

The user interface will be reworked to allow dynamic scalability based on the screen resolution and ratio.

NF1: Design patterns

Design patterns will be used in the development of key components.

NF2: Continuous deployment

A continuous deployment cycle is developed and used for the deployment platform

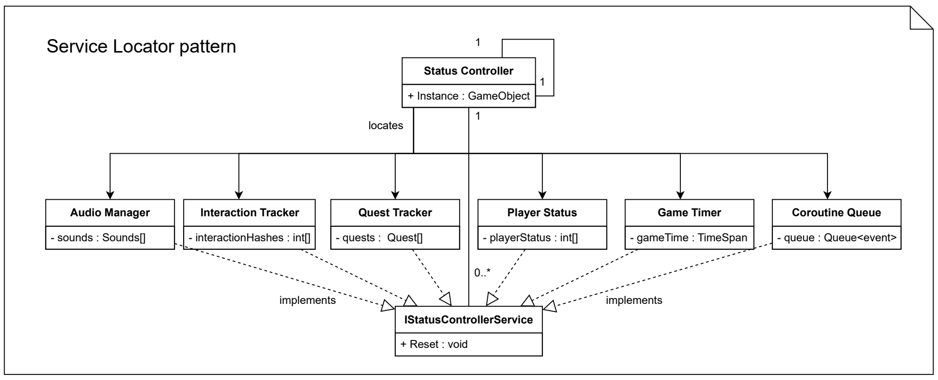


Figure 4.3: Status controller class diagram

4.2 Game managers

A component often used throughout the whole game. A game manager is often used to keep track of key values that drive many other components. Here a singleton pattern, which is mentioned in section 2.5, is a perfect candidate. This has the added benefit of being globally accessible for other components to use. As there are so many they can be grouped together with a service locator pattern, mentioned in section 2.6, therefore only one master singleton exists, which further increases accessibility. This class will be called Status Controller. Functional and non-functional requirements for all game managers are listed in figure 4.2.

4.2.1 Status controller

This object will act as the service locator of other game manager components. This object will also be the carrying singleton object (master singleton) allowing programmers easy access to this component from any script in the scene.

F2.1: Handles game reset

Status controller carries out reset for contained services.

NF2.1: Service locator pattern

Status controller implements the service locator pattern.

NF2.2: Singleton pattern

Status controller implements the singleton pattern.

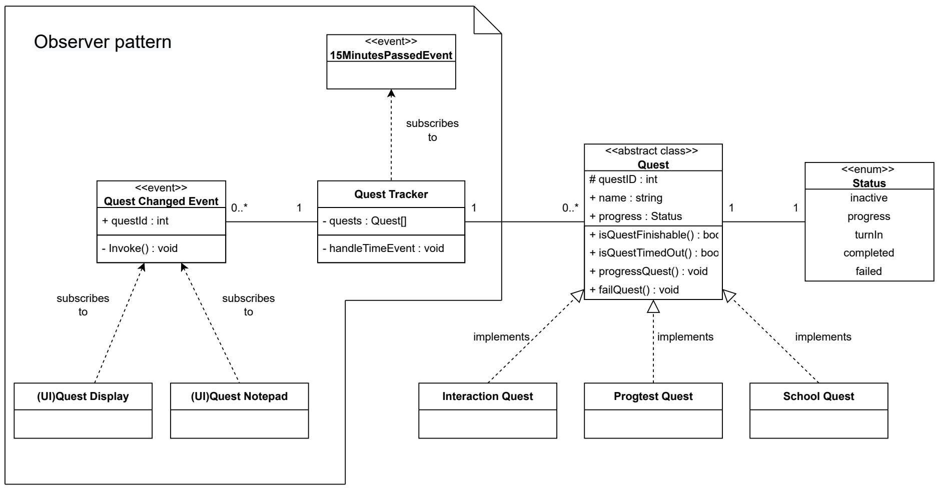


Figure 4.4: Quest tracker class diagram

4.2.2 Quest system

It subscribes to the Game Timer for elapsed time events, based on which it will check for quest requirements and allow progression or cancel them due to time-out. It itself will invoke events of its own like quest changed events, and these will be subscribed to by quest handling UI elements, decoupling them from this component. The Quest system knows nothing about its subscribers, allowing many or no subscribers to the event without a problem.

F3.1: Handles all quests

This object will handle the business layer of quests. It will act as a database for quest fetching and also allow for progression, completion, or cancellation of quests.

NF3.1: Observer pattern

Quest system implements the observer pattern.

NF3.2: Invokes quest events

Quest system invokes events when the quest log changes in any way.

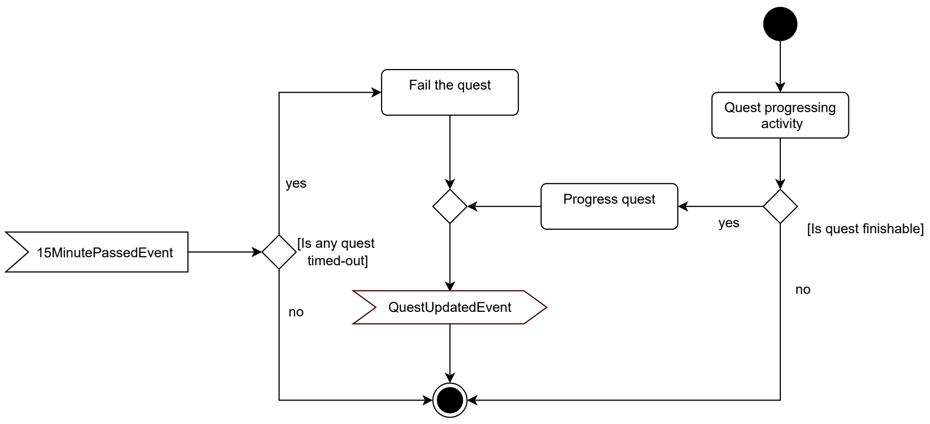


Figure 4.5: Quest tracker activity diagram

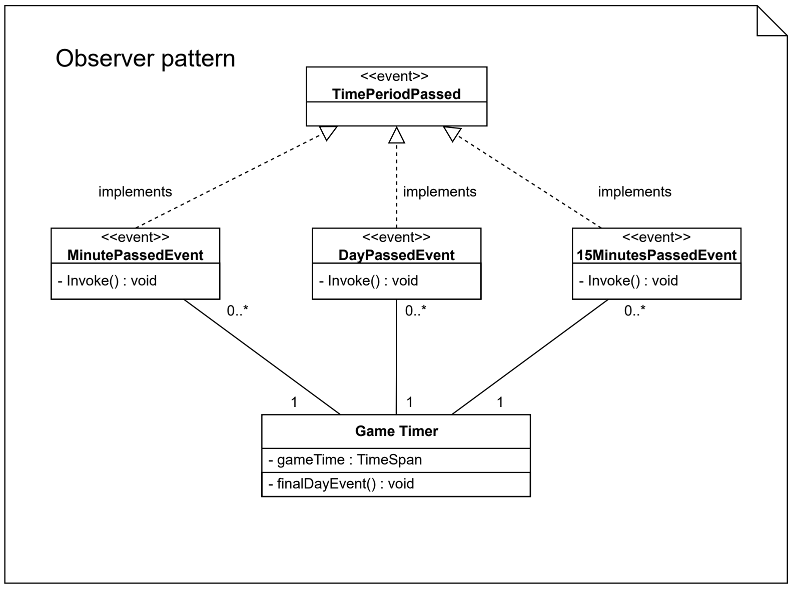


Figure 4.6: Game timer class diagram

4.2.3 Game timer

There are multiple mechanics tied to the game time ie. nighttime scene changes, quest acquisition/expiration, and several presentation components. The game timer is also a good candidate for the observer pattern, it updates its time internally and sends events based on the time elapsed.

The 15-minute interval events are enough because the School Quests adhere to a potential school timesheet which is segmented into 15-minute chunks.

F4.1: Handles in-game time

Game timer manages the in-game time.

NF4.1: Observer pattern

Game timer implements the observer pattern.

NF4.2: Stable time representation

Game timer will use a system class for time representation, making it more stable than own time representation.

NF4.3: Invokes time events

Game timer will invoke events based on elapsed time chunks.

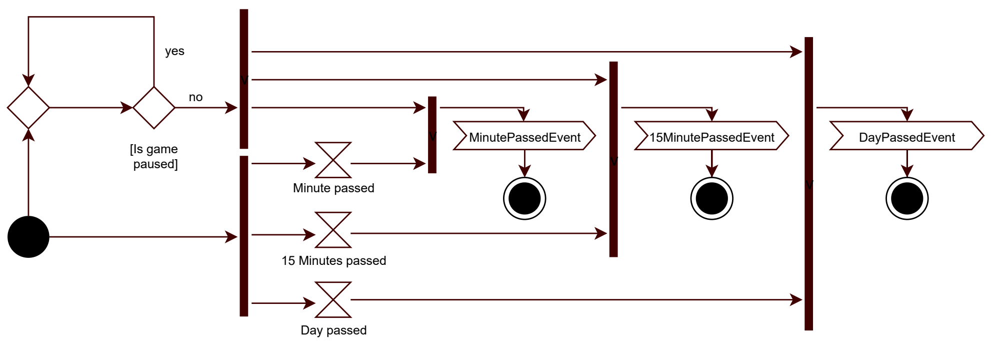


Figure 4.7: Game timer activity diagram

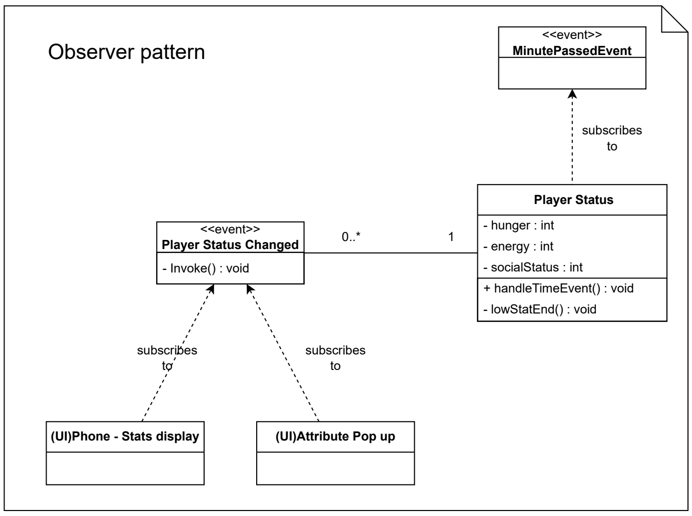


Figure 4.8: Player status class diagram

4.2.4 Player status

This component subscribes to the Game Timer, listening for a sufficient amount of time passed for changing the values of stats. Sends out attribute changed event together with information on what attribute and value. The subscribers are typically in the presentation layer.

F5.1: Handles player stats

Player status keeps track of status attributes, which are described in section 5.6.2, ie. Social status, hunger, energy.

NF5.1: Observer pattern

Player status implements the observer pattern.

NF5.2: Invokes stats events

Player status invokes events when stats change in any way.

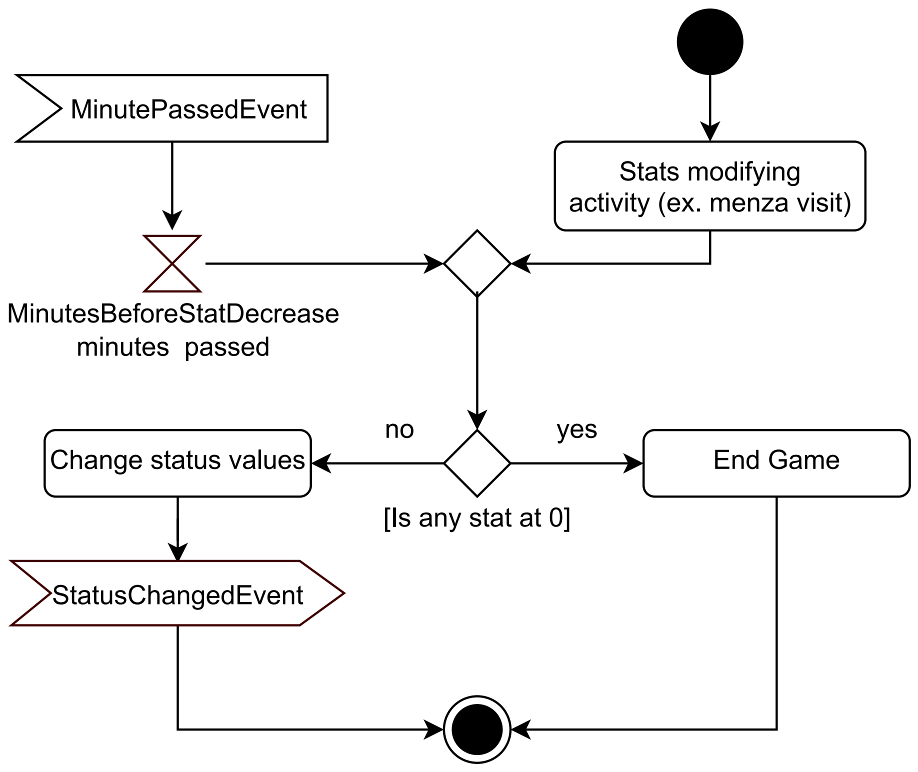


Figure 4.9: Player status activity diagram

4.2.5 Interaction tracker

It keeps track of dialogues that already happened, but keeping whole dialogues in some kind of array only to compare them to check if they happened is a waste of space. Another approach would be to assign a unique id to each dialogue and keep track of those ids in a dictionary. But the best approach would be to define what a unique dialogue is. A unique dialogue is one that differs content-wise to another. We will be hashing the contents of these dialogues and keeping track of those hashes, a sort of signature.

The dialogue history is then cleared once per day. When he approaches an NPC again next day, the quest is still available, and the NPC is waiting for the quest to be completed. A subscription to the game timer's day passed event will take care of that.

F6.1: Handles interaction history

Interaction tracker will be handling the availability of selected interactions with the player. If they happened the same day, the interactions will not happen the second time.

F6.2: Handles selected interactions

It will make sure that the player cannot interact with two Interactive (described in section 4.2.6) objects at once.

NF6.1: Observer pattern

The interaction tracker implements the observer pattern.

NF6.2: Invokes interaction events

It will trigger an event when an interaction is available.

4.2.6 Interactive

When added onto an object, this component makes it interactive to the main character. It controls everything that the player can interact with, i.e. NPCs, easter eggs, etc... It will need to execute an action after the interaction ex. start a dialogue, transport the player to a different scene or start a quest. This component needs to be easy to edit in the editor, allowing simple text changes or interaction changes to take effect without looking at the code. This is one of the components that a game designer can interact with, with ease.

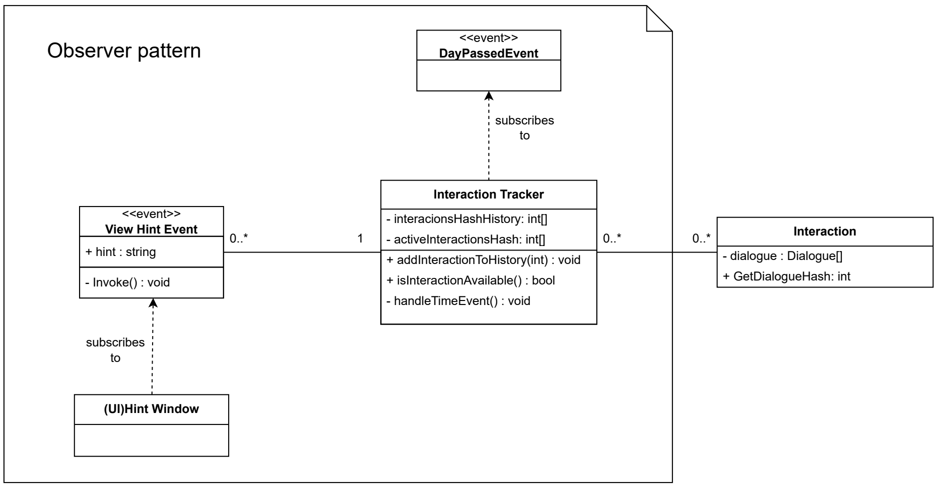
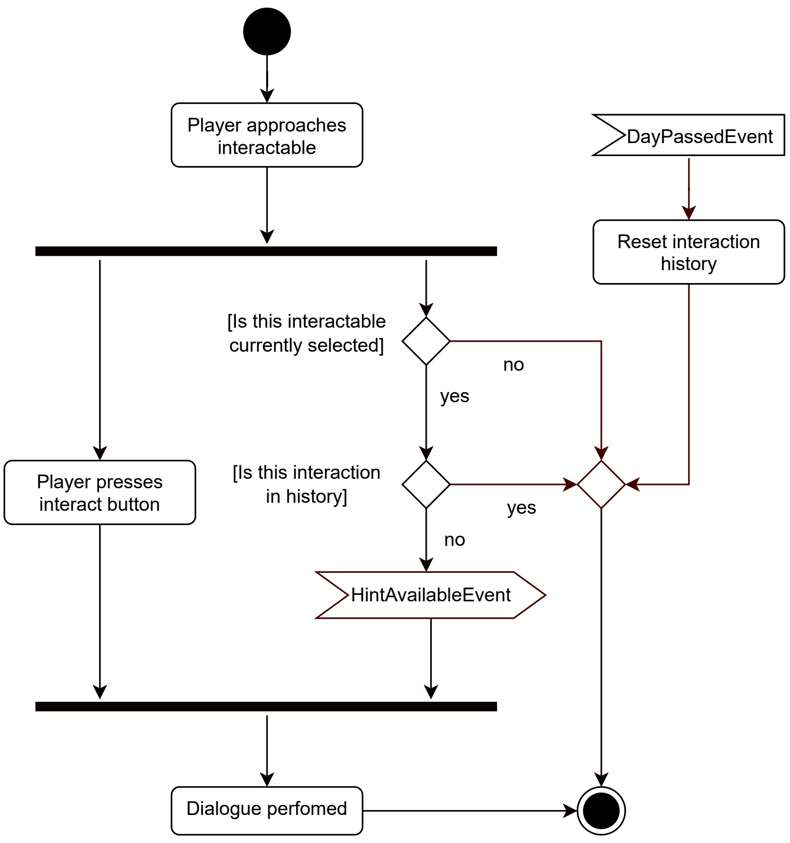


Figure 4.10: Interaction tracker class diagram



Player approaches interactable
DayPassedEvent
Reset interaction history
[Is this interactable currently selected]
no
yes
Player presses interact button
[Is this interaction in history]
yes
no
HintAvailableEvent
Dialogue perfomed

Figure 4.11: Interaction tracker activity diagram

CHAPTER 5

Game design document

5.1 Introduction

The game design document describes the gameplay design of FitLife. It shows a vision of the final game and specifics about its inner workings and goals.

Scope

This is used by all parties involved during the creation and programming of the game.

5.2 Target System

The game is played in an internet browser. The target platform is PC (WebGL).

WebGL

WebGL is a platform that allows Unity content to be played inside a web browser [17]. It is supported in most recent web browsers, including Chrome, Firefox, and Safari. This platform allows the game to be played without downloading files and running any executable files. Its drawbacks come from the lesser performance provided by the browser sandbox.

5.3 Development System

FitLife is developed in Unity, allowing further expansion to other platforms in the future.

5.4 Target Audience

The target audience are current or future Czech-speaking CTU FIT students.

5.5 Specification

Concept

FitLife aims to simulate what a student in his first semester of CTU FIT feels. The gameplay is hectic, chaotic, and entertaining.

Setting

The game is set in present-day CTU campus.

Game Structure

There are five environments that the player can freely explore. There are various NPCs scattered around the game, which are interactable and provide exciting dialogue. Certain NPCs provide side quests that the player chooses to complete or not.

Players

The game is played by a single player on a PC.

Graphics

The game has pixel graphics. Most of the graphics, animations, and sprites shown in the game are created by David Mikulka [1], please refer to his bachelor's thesis to learn about this aspect of the game.

Objective

The objective of the game is to survive the three days at FIT. Try to pass the semester with good grades at the end. Go to school on time of the lecture, explore the world and enjoy the environment with many objects to interact with. The general gameplay loop is shown in an activity diagram 5.1.

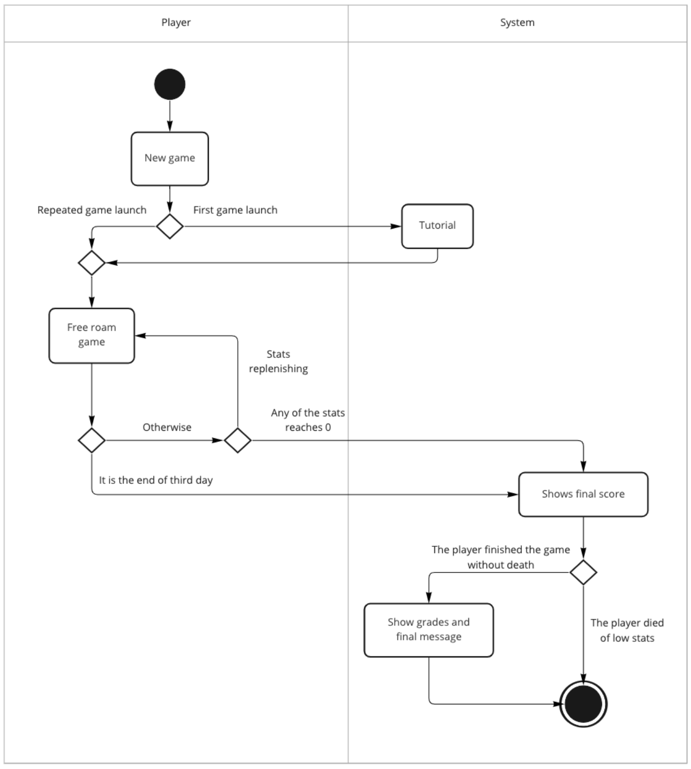

Player
System
New game
Repeated game launch
First game launch
Tutorial
Free roam game
Stats replenishing
Any of the stats reaches 0
Otherwise
It is the end of third day
Shows final score
The player finished the game without death
Show grades and final message
The player died of low stats

Figure 5.1: Game activity diagram, created in cooperation with Mikulka [1]

Failing conditions

The game is failed upon any of the main stats (Section 5.6.2) reaching zero.

Winning conditions

The game is won if day three midnight (beginning of day 4) is reached. The player is presented with grades at the end of the semester based on how he has done, this is further explained in section 5.6.3.

Landscape

The landscape is 3/4 view, 2D with perspective. There are multiple environments, e.g., Strahov dormitory, CTU campus, bar, etc. Some NPCs are animated, including the main character. The screen scrolls with the character moving left/right/up/down keeping the player in the center. The perspective is purposefully kept as such to not reveal the whole map to the user. As the player wanders around the map, he figures out where he needs to go.

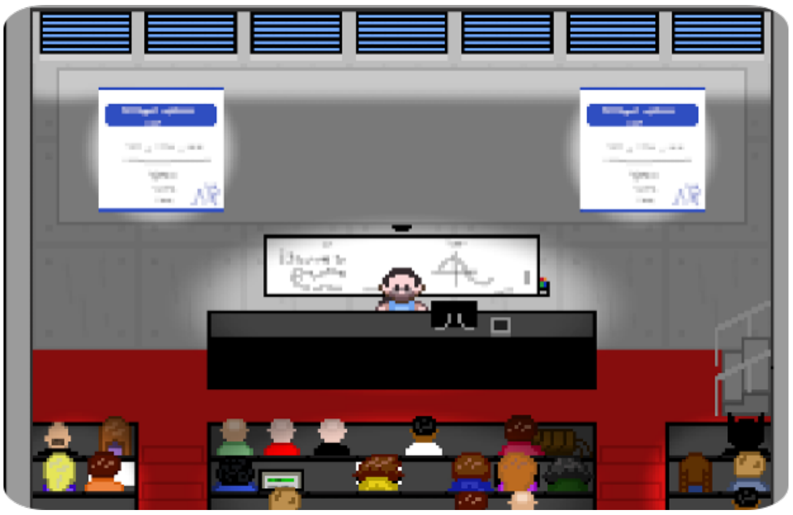


Figure 5.2: The player, during one of the lectures

5.6 Gameplay

5.6.1 Story

The player wakes up in his dormitory in Strahov. After acknowledging the tutorial, he heads out to explore the world. The quest tracker shows him that he needs to attend school activities. During his exploration, he discovers many people and their interesting dialogue. After visiting his lectures, he visits the bar for different quests and interactions and heads off to sleep back in his dormitory. The next day he receives new school activities in his quest tracker along with side quests he picked up on his first day. There are new people around him, and the player continues discovering the world by attending lectures and finishing his side quests from the first day. He discovers that when working on his homework at the library, he falls asleep and needs to complete a timed minigame[*] or he will be locked inside the library. After leaving NTK, he heads off to sleep for his exam day.

On the third day, his school activities consist of exam minigames. Being successful during these minigames dictates his final score on that subject. At the end of the day, the player receives a final exam summary and is presented with the ending screen.

[*]A small game within a game, typically with a very simple objective

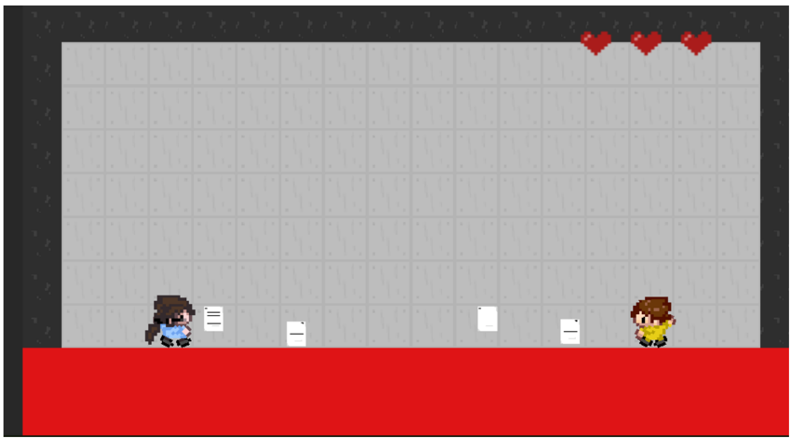


Figure 5.3: ZMA exam minigame

5.6.2 Stats

Stats include:

- *Energy*
- *Hunger*
- *Social status*

These stats simulate the main character's (Figure 5.4) well-being. Energy and Hunger are constantly subtracted during active gameplay (there are events where this is paused, e.g., pause menu, reading text, etc.) Social status is granted when talking to NPC characters and subtracted when performing school activities (Section 5.6.4).

The player is granted an initial amount of stats at the beginning of the game. If any of these stats reaches 0, the game ends and any progression is lost.

5.6.3 Grading system

The player is graded by each subject, which are listed in section 5.6.3, at the end of the game, if he survives. The player for each subject receives a grade (A-F) based on attendance at school. Going to lectures grants him 1 point, and attending exams grants him 4 points. Failing an exam in any subject results in an F. In the end, the scores are tallied and a grade is given according to the points received (0-5, F-A).

Subjects

Subjects that our protagonist is attending this year. These are categories upon which the player is graded at the end of the game, specifics are explained in section 5.6.3.

- PA1
- ZMA
- PS1
- PAI
- CAO
- MLO

5.6.4 Quests

During the gameplay, the player is guided by quests. Finishing these improves the final ending of the game. The quests are either main quests, which happen primarily at school, or side quests which are received by NPCs around the world.

Main quests

Main quests are assigned automatically at the start of a new day. Completing them will subtract a set of predefined stats, including social status (the player is studying and does not have time to socialize). They are as follows:

- *Day 1: 9:15* - BI-CAO lecture
- *Day 1: 11:00* - BI-ZMA lecture
- *Day 1: 16:15* - BI-PS1 test
- *Day 2: 7:30* - BI-MLO lecture
- *Day 2: 9:15* - BI-PA1 lecture
- *Day 2: 16:15* - BI-PAI test
- *Day 2: 23:59* - Finish Progtest homework (BI-PA1)
- *Day 3: 9:15* - BI-ZMA exam
- *Day 3: 11:00* - BI-CAO exam
- *Day 3: 16:15* - BI-MLO exam

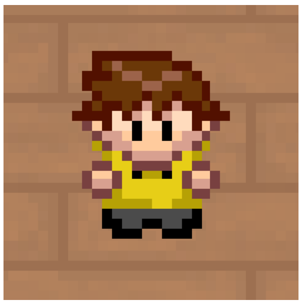


Figure 5.4: Main character

Side quests

Side quests are picked up by talking to certain NPCs marked by an exclamation mark. Completing these quests will grant the player some social status.

- *Club house FIT--* - You will need to get past the guards in front of the club
- *NTK ticket* - You again do not have your card, get yourself a temporary ticket
- *Stray dog* - Dog comes home with you if you find him a treat
- *Lost plant* - Go fetch the friend in FIT-- his plant
- *Lost scripts* - A panicking student left his AAG scripts somewhere
- *NTK closing minigame* - You overslept while doing progtest, get out of the library before they lock you in
- *Lost ISIC card* - Attend one lecture while this quest is active so you do not forget to get the lost card on your way out
- *Beer pong* - Help fellow FIT-- clubmates find their beer pong ball

5.6.5 Characters

Main character

A new student at CTU FIT, clueless as the player controlling him (Figure 5.4).

Kosta and Filip

Welcoming the player into the bar, they are guarding the entrance to a super-secret school club FIT--. The player is tasked with getting past them, granting access to the club on the second floor.

Dog

The dog is sitting on the second floor of the bar. After giving him a treat, he comes with you and can be seen in your dormitory.

The medicine man

Standing in the FIT-- club, he tells the player that he is missing his favorite medicine plant. And the player is tasked with retrieving this plant during his exploration.

Panicking student

A student located on the CTU campus is struggling to find his lost school book that he lost after a night of partying. The player is tasked with fetching the book if he finds it.

Ping pong mishap

A pair of students lost their ping-pong ball, the player is tasked with finding it. It is cleverly hidden in the terrain.

Pedestrians and students

These NPCs are interactable, and have interesting dialogue but do not lead to any quests. There are there to liven the environment.

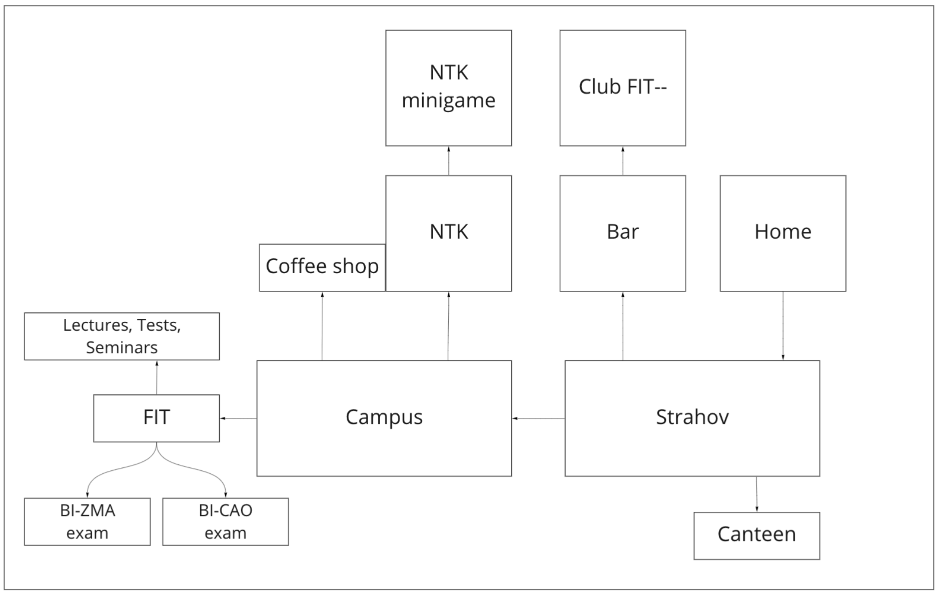


Figure 5.5: Level design, cooperation effort with Mikulka [1]

5.6.6 Level design

There are seven scenes that the player can be in. Traversing different scenes are done through points of transportation (e.g., Bus station to Campus, Door to the bar, Door to school, etc.) The camera is zoomed in close to the player. The whole scene is not in his view (except his dormitory), encouraging exploration.

- *Strahov dormitory* - the introductory scene where the player is introduced to the game, he returns here to sleep every night.
- *Strahov* - just outside the dormitory, the player is introduced to the environment with unique people and items.
- *Bar first floor* - a lively place to meet interesting characters, the entrance to the second floor is guarded by Kosta and Filip.
- *Bar second floor* (FIT--) - after finishing the quest, Kosta and Filip let the player through a place where students from the same school are gathered to chat and do homework.
- *CTU campus* - arriving with a bus from Strahov, the player finds himself at the school campus.
- *NTK library* - for NTK access, a quest needs to be finished, giving the player a place to study.
- *FIT lecture hall* - accessed at CTU campus, main lecture and exam quests are finished here.

5.6.7 Music and effects

The main background music is generated using AI (Google Magenta) and seeded with own creation. It is fun and upbeat, giving the game a unique tone and atmosphere. The rest of the music is composed by David Mikulka [1] and is expanded upon in his Bachelor's thesis.

Google Magenta

An open-source research project exploring the role of machine learning as a tool in the creative process [18]. Using this library and its trained neural network, I was able to create unique music using various settings and edited the output tracks into something pleasant.

5.6.8 In-game controls

- *ESC* - opens the Pause menu
- *W, Up arrow* - moves the character up
- *A, Left arrow* - moves the character left
- *S, Down arrow* - moves the character down
- *D, Right arrow* - moves the character right
- *Q* - opens the quest tracker
- *Tab* - opens side quest menu
- *M* - opens the phone
- *E* - interactions with objects and environments
- *Space* - dismisses the dialogue window

5.6.9 User interface

These components are for informing the player on the current state of the game. Designed as a presentation layer for the user, an Observer pattern, described in section 2.3, is a good choice. The presentation layer does not need to be updated with each frame. Implementing events responding to the application layer are a vital part in making a well-performing software without creating dependency spaghetti[*].

Main menu

The main menu is controlled with the mouse and consists of three buttons:

- *Start game* - begins the game
- *Settings* - gives the option to turn off the music
- *About authors* - gives the list of all participants on the project of FitLife, listed in table 3.1
- *End game* - terminates the game

[*]This describes the state of complex coupling and unmanageable dependencies throughout the solution

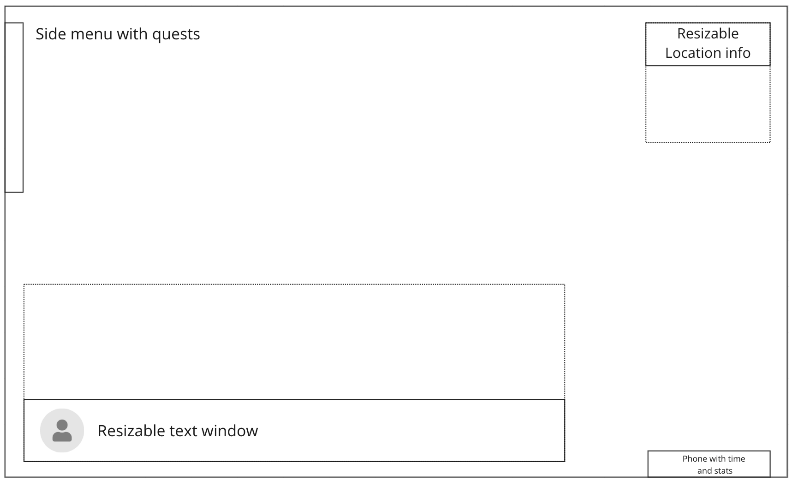


Figure 5.6: UI layout - wireframe

5.6.10 In-game pause menu

- *Sound toggle* - toggles the game music
- *Continue* - unpauses the game
- *End game* - terminates the game

This menu, when triggered, pauses the game pausing any player movement and time progression.

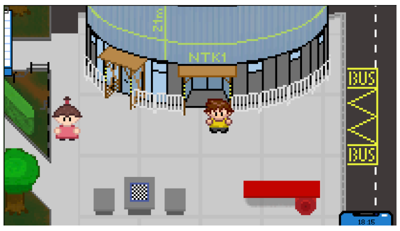
Phone

The phone is visible in the bottom right of the screen (Figure 5.6) showing the current time. It can be expanded using M key with accompanying sound effects and animation. In the expanded view, the values of the main three stats, described in section 5.6.2, are visible. The values of the phone are updated using the constructs of the Observer pattern, described in section 2.3.

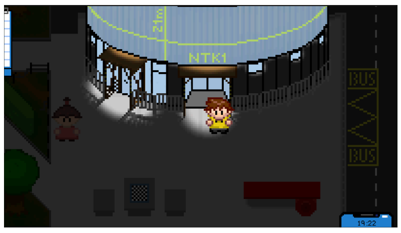
Pop up text window

This window at the bottom left (Figure 5.6) is used in many places during the game. It is one of the main interfaces between the player and the game. Any text-based information is conveyed using this component, shown during the tutorial, npc conversation or quest interactions. It needs to be versatile and universal, expanding based on the size of the text.

An icon portraying the person talking is also included, portraying the person talking. The icon is set to the main character's sprite when talking to himself, it is changed to a question mark when the game is informing the player about certain events.



(a) Day lighting



(b) Night lighting

Figure 5.7: Day and Night cycle

The window is dismissed by pressing space. A queue needs to be implemented, allowing asynchronous tasks to inform the player after he is done reading the current text. During the active period of this window, the player is unable to move, and time progression is paused.

Quest display

Quest display (Figure 5.6) shows active quests ready to be completed. The viewing of quests adheres to the Observer pattern, described in section 2.3. The quest display updates in reaction to quests changing state, with accompanying animation and sound effects.

Location Info

This window in the top right of Figure 5.6 shows the current location the player is located at, the window will be resizable to fit any reasonably long text.

Day and Night cycle

After 19:00, the environment transitions into night lighting, and at 6:00 transitions back into day lighting. If the player is not present at home during the shifting of days (23:59), the player appears at home with a headache and is slashed -20 points from each of his stats, encouraging spending the night in bed instead of falling asleep on the street.

5.7 First launch

The game starts with the unity logo, continuing onto a disclaimer. Warning the player that the similarities of the interpreted characters are completely random. The player is prompted with the main menu. Choosing play will take the player to his dormitory. A tutorial is shown, and after finishing it, the player is free to roam the world.

Part III

Realisation

CHAPTER 6

Development

In this chapter, I will be focusing on the programming of the final game. The designed components will be incorporated into the rest of the game. There is also a lot of refactoring to do.

The code will be written in C#. The project has been developed in Unity version 2019.3.10f1, which will be promptly upgraded to 2019.4.33f1 to allow the use of the new Continuous integration (CI) solution provided by Unity. The game is deployed at the web address: https://antik98.github.io/FitLife/*

Please note that not all programming work is showcased here, countless hours were spent rewriting, integrating, and debugging all components in the game. Nearly all of the code after SP1 and SP2 has been written by me.

6.1 Used unity features

TextMeshPro

TextMeshPro* is an external text package that improves upon Unity's own text solution. This is the only external package used in the project.

Coroutines

Coroutines allow segments of code to run asynchronously. This code execution is used widely throughout the implementation, from basic initialization to visual effects.

*Accessed on 10.5.2022, this is not a purchased domain, but rather a platform (Github pages) provided by Github, and might not be accessible in the future, if that is the case please use included source code to compile the game

*Available at: https://docs.unity3d.com/Manual/com.unity.textmeshpro.html

CI/Continuous delivery (CD)

Unity offered Collaborate* solution, which worked for the project since the beginning of the project was discontinued and replaced by Plastic SCM*. Previously the game was built by a developer each time and uploaded by hand to GitHub pages.

The project needs a CD system, the game will be deployed using Github Actions [19]. This has been done and the game is distributed using a link pointing to Github pages. A guide was followed on the Unity-Actions GitHub repo [20].

Events

Events allow the observer pattern to be implemented. It makes the loose coupling of the manager class and UI component possible.

Prefab

A Prefab is a group of preset and generalized components. The developer puts together prefabs of regularly occurring objects from NPC or simple intractables to UI components and Status controllers, giving freedom to the designer to place them and modify them without touching the underlying code.

6.1.1 Canvas components

These components were vital in the creation of a scalable User interface, they manipulate objects in reaction to the screen changing shape and size.

Canvas Scaler

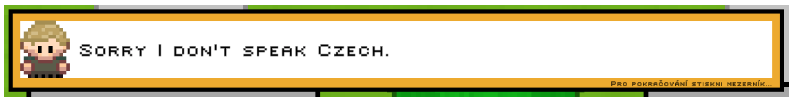
This component is the bread and butter of user interface scaling, yet it is quite complicated. It has issues with overlay canvases resolution management. This component allows the game to be run on a 4k monitor, or a 144p monitor if need be.

Horizontal/Vertical layout group

These two components are what make the UI elements auto-sizing. They are wildly unintuitive to use and extremely hard to get to work correctly, but the result is worth every broken keyboard. The UI elements are regardless of content always perfectly aligned and its container stretched to match the bounding box of the contents.

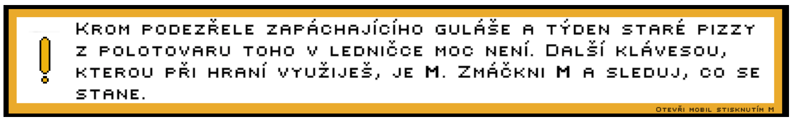
*https://docs.unity3d.com/Manual/UnityCollaborate.html

*Available at: https://unity.com/products/plastic-scm


Sorry I don't speak Czech.

Pro pokračování stiskni mezerník...

Figure 6.1: Text window


Krom podezřele zapáchajícího guláše a týden staré pizzy z polotovaru toho v ledničce moc není. Další klávesou, kterou při hraní využiješ, je M. Zmáčkni M a sleduj, co se stane.

Otevři mobil stisknutím M

Figure 6.2: Expanded text window

6.2 UI scaling

When refactoring the UI, I encountered bad practices with UI object creation, the objects did not match what was seen in the played game. Now, this is normally not a problem when the objects are created and used by people with a monitor of same screen size, ratio and resolution. But when the game is supposed to be played by more people the size of the screen needs to be considered.

The UI has been recreated with proper techniques and scales to the camera using various canvas components, listed in section 6.1.1. The automatic expansion of the layout based on text content and size, like the text window (Figure 6.2) and (Figure 5.6.10) are handled with Horizontal and Vertical layout groups, described in section 6.1.1. The next thing to tackle is the camera itself.

6.3 Camera scaling

This is often solved with Unity's external package Cinemachine [21]. But this is a simple game and should not require a complex package like that.

A script allows for the dynamic resizing of the camera based on the screen size (Code listing 6.3). Because of this camera scaling, we need to also scale our camera boundaries that are used when following the player. That is done using our calculated camera size dimensions (Code listing 6.3), the constant is the size of the camera used when setting the boundaries. By subtracting this constant we are getting the center point of a camera that is touching the boundaries.

Code listing 6.1: Camera resizing calculation

```
// Update is called once per frame
void Update()
{
    sizeY = cam.orthographicSize * 2;
    ratio = (float)Screen.width / (float)Screen.height;
    sizeX = sizeY * ratio;

    heightScale = ratio / targetAspect;
    widthScale = 1.0f / heightScale;

    Rect rect = cam.rect;
    rect.width = heightScale < 1    ? 1 : widthScale;
    rect.height = heightScale > 1   ? 1 : heightScale;
    rect.x = heightScale < 1        ? 0 : (1.0f - widthScale) / 2.0f;
    rect.y = heightScale > 1        ? 0 : (1.0f - heightScale) / 2.0f;

    cam.rect = rect;
}
```

Code listing 6.2: Camera follow calculation

```
//Framerate independent Update()
private void FixedUpdate()
{
    Vector3 cameraFollowPosition = GetCameraFollowPositionFunc();
    cameraFollowPosition.z = transform.position.z;

    Vector3 boundPosition = new Vector3(
        Mathf.Clamp(cameraFollowPosition.x,
        (minValues.x + 3.200652f - (sizeX / 2) * widthScale),
        (maxValues.x - 3.200652f + (sizeX / 2) * widthScale)),

        Mathf.Clamp(cameraFollowPosition.y, minValues.y, maxValues.y),
        Mathf.Clamp(cameraFollowPosition.z, minValues.z, maxValues.z)
        );

    Vector3 smoothPosition = Vector3.Lerp(transform.position, boundPosition,
         smoothFactor * Time.fixedDeltaTime);
    transform.position = smoothPosition;
}
```

6.4 Game managers

For implementaion of the observer pattern I used events and delegates [22] offered by the C# language. The service locator pattern and Singleton pattern were extremely useful in the programming of other parts of the game.

Status controller

The script makes sure that only one instance exists, it simply destroys it's own game object, taking the rest of the game managers with it. This elegant solution makes sure that only one instance exists. It has the added benefit that the other game managers do not need the object management code to be present.

Code listing 6.3: Status controller - Service Locator

```
public QuestTracker questTracker;
public GameTimer gameTimer;
public PlayerStatus PlayerStatus;
public InteractionTracker interactionTracker;
public CoroutineQueue coroutineQueue;
public AudioManager audioManager;

public static bool initialized => _instance != null;

private static StatusController _instance;

public static StatusController Instance { get { return _instance; } }

void Awake()
{
    if ( _instance != null)
    {
        Destroy(gameObject);
        return;
    }
    _instance = this;
}
```

Game timer

The game timer uses System.TimeSpan, instead of own time representation. I went with System.TimeSpan instead of System.DateTime because there was no need for such a wide representation.

Code listing 6.4: Game Timer - Observer pattern

```
public delegate void TimePeriodPassed();
public event TimePeriodPassed BroadcastDayPassed;
public event TimePeriodPassed Broadcast15MinutesPassed;
public event TimePeriodPassed BroadcastMinutePassed;
void InvokeTimePassedEvents()
{
    TimeSpan nextTick = gameTime + TimeSpan.FromSeconds(Time.deltaTime *
         timeSpeedConstant);

    //Called when 15 minutes passed
    if(gameTime.Minutes % 15 == 0 && nextTick.Minutes % 15 != 0 )
    {
        Broadcast15MinutesPassed?.Invoke();
    }

    //Called when 1 minute passed
    if(gameTime.Minutes != nextTick.Minutes)
    {
        BroadcastMinutePassed?.Invoke();
    }

    // Called when Days change
    if (gameTime.Hours >= 0 && gameTime.Hours < 6)
    {
        BroadcastDayPassed?.Invoke();
    }
}
```

UI element

The phone window UI element (Figure 6.3) subscribes to HandleMinutePassed event from Game timer to update it's time, it also subscribes to the Player status component for its PlayerStatusChanged event for updating the stats display.

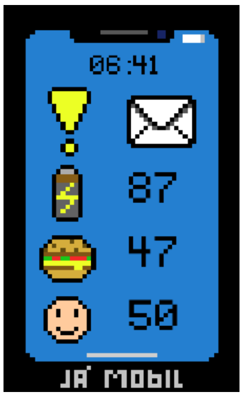


Figure 6.3: Phone

Interaction tracker

The implementation of this component is straightforward. Because we are hashing the dialogue we will be overriding the default Object.GetHashCode() for the Dialogue class and receiving the correct unique id we are looking for. We can use System.Linq and some cool functional programming for this job.

Code listing 6.5: Interaction tracker - dialogue history

```
public delegate void EventTriggeredViewHint(object sender, bool view, string hint);
public event EventTriggeredViewHint HandleEventViewHint;

//This method is triggered when entering/leaving an Interactive hitbox
public void TriggerHint(object sender, bool view, string hint = "")
{
    if (view)
    {
        activeInteractions.Add((sender.GetHashCode(), hint));
    }
    else
    {
        activeInteractions.RemoveAll( s => s.Item1 == sender.GetHashCode() );
    }
    HandleEventViewHint?.Invoke(sender, activeInteractions.Any(), activeInteractions.
        FirstOrDefault().Item2) ;
}
```

Code listing 6.6: Dialogue hashing

```
public Dialogue(string[] sentences)
{
    this.sentences = sentences;
}
public override int GetHashCode()
{
    return sentences.Aggregate(1, (prod, next) => prod ^ next.GetHashCode());
}
```

UI element

The hint window UI element (Figure 6.4) subscribes to the HandleEventViewHint event.

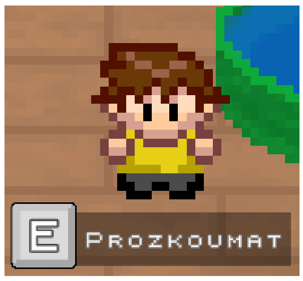


Figure 6.4: Hint window

Player status

Player status handles the stats values for the game. Stats behavior is described in section 5.6.2.

Code listing 6.7: Player stats management

```
public static readonly int minutesUntilStatDecrease = 11;

public int energy { get; private set; }
public int social { get; private set; }
public int hunger { get; private set; }
private IEnumerator OnEnableCoroutine()
{
    yield return new WaitUntil(() => StatusController.initialized);
    StatusController.Instance.gameTimer.BroadcastMinutePassed += HandleMinuteChanged;
}
private void OnDisable()
{
    StatusController.Instance.gameTimer.BroadcastMinutePassed -= HandleMinuteChanged;
}

private void HandleMinuteChanged()
{
    if(++minuteCounter == minutesUntilStatDecrease)
    {
        minuteCounter = 0;

        if (energy <= 0 || hunger <= 0 || social <= 0)
        {
            gameTimer.StopTimer();
            lowStatEvent();
            return;
        }

        hunger -= 1;
        energy -= 1;
        HandleAttributesChanged?.Invoke();
    }
}
```

UI element

The phone stat display (Figure 6.3) subscribes to the HandleAttributesChanged event. Another UI element subscribing to this component is the Attribute pop up.

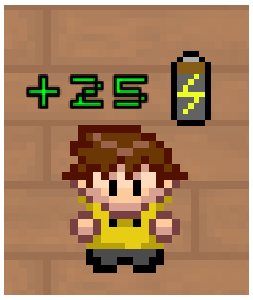


Figure 6.5: Status change event

Quest tracker

This component handles the quests assigning, completing or canceling of individual quests. While also acting as a sort of database for fetching of these quest objects. It subscribes to the Game timer's (Code listing 6.4) Broadcast15MinutesPassed event for the time-out check, relieving many potential CPU cycles. Upon changes done to any quest, an event (HandleQuestChanged) is sent signaling which quest has changed.

Below is how the school quest overrides the abstract implementation of IsQuestTimedOut and IsQuestFinishable. An interaction quest can be finished anytime, so the method returns true every time.

Code listing 6.8: Quest tracker

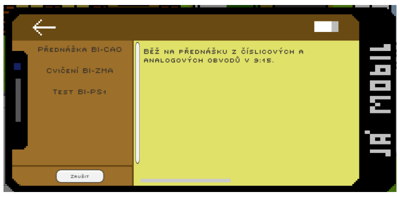
```
public delegate void EventTriggeredQuestChanged(int questId);
public event EventTriggeredQuestChanged HandleQuestChanged;

IEnumerator OnEnableCoroutine()
{
    yield return new WaitUntil(() => StatusController.initialized);
    StatusController.Instance.gameTimer.BroadcastDayPassed += HandleDayPassed;
    StatusController.Instance.gameTimer.Broadcast15MinutesPassed +=
        Handle15MinuteIntervalPassed;
}
public void OnDisable()
{
    StatusController.Instance.gameTimer.BroadcastDayPassed -= HandleDayPassed;
    StatusController.Instance.gameTimer.Broadcast15MinutesPassed -=
        Handle15MinuteIntervalPassed;
}

//Quest is abstract, concrete classes override IsQuestTimedOut based on time given
foreach(Quest q in quests)
{
    if (q.status == Quest.Status.progress && q.IsQuestTimedOut(gameTimer.gameTime))
    {
        FailQuest(q.questID);
    }
}
```

Code listing 6.9: School quest

```
// Is it past deadline?
public override bool IsQuestTimedOut(TimeSpan gameTime)
{
    return gameTime > deadline + TimeSpan.FromMinutes(15f);
}
public override bool IsQuestFinishable(TimeSpan gameTime)
{
    return gameTime >= deadline - TimeSpan.FromMinutes(15f) && gameTime <= deadline +
        TimeSpan.FromMinutes(15f);
}
```

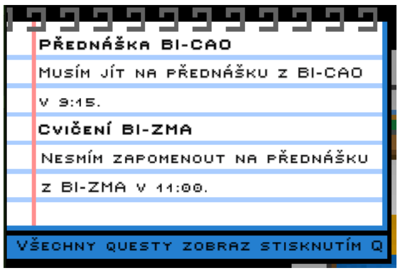
UI element

The Quest window subscribes to the HandleQuestChanged event, so if any quest changes when the window is open, it updates itself. Another subscriber is the quest tracker side menu, which also updates itself on quest change.



Běž na přednášku z číslicových a analogových obvodů v 9:15.

(a) Main quest menu



Přednáška BI-CAO

Musím jít na přednášku z BI-CAO v 9:15.

Cvičení BI-ZMA

Nesmím zapomenout na přednášku z BI-ZMA v 11:00.

Všechny questy zobraz stisknutím Q

(b) Side quest menu

Figure 6.6: Quest menus

6.5 Implementation curiosities

Quest side menu animation

I did not work on many animations in the game, because I mostly focused on making all the features work. One of the exceptions was the scratching and subsequent retyping animation of the quest side menu. This was done entirely through code. TextMeshPro supports HTML font-modifiers like $< b >$, or $< s >$. So using these font modifiers and moving the terminating modifier one letter at a time, this scratching effect is achieved.

Note that this is possible because of the observer pattern (Quest tracker) notifying its subscribers(side quest menu) on quest change of some id. The WaitForSeconds instantiation is done beforehand to minimize garbage collection.

Code listing 6.10: Scratching and typing animation effect

```
private IEnumerator UpdateQuestView(int? questId = null)
{
    ...
    var items = questLines.ToList().Zip(questLinesText.ToList().Zip(questTracker.
         getActiveQuests(), (x, y) => (x, y)), (x, y) => (x, y.x, y.y) );

    foreach (var (title, text, quest) in items)
    {
        if(questId.HasValue && questTracker.getQuest(questId.Value).name == title.text
             )
        {
            StartCoroutine(ScratchText(title, quest.name));
            yield return StartCoroutine(ScratchText(text, quest.notysekText));

        }
        StartCoroutine(DisplaySentence(title, quest.name));
        yield return StartCoroutine(DisplaySentence(text, quest.notysekText));
    }
    ...
}

private IEnumerator ScratchText(TextMeshProUGUI gui, string newText)
{
    WaitForSeconds w = new WaitForSeconds(0.02f);
    string originalStr = gui.text;
    if(newText != gui.text)
    {
        questAnimator?.SetBool("IsOpen", true);
        foreach (int x in Enumerable.Range(0, originalStr.Length))
        {
            gui.text = "<s>" + originalStr.Substring(0, x) + "</s>" + originalStr.
                 Substring(x);
            yield return w;
        }
    }
    yield return null;
}

IEnumerator DisplaySentence(TextMeshProUGUI gui, string display)
{
    ...
    WaitForSeconds w = new WaitForSeconds(0.02f);
    foreach (char letter in display.ToCharArray())
    {
        gui.text += letter;
        yield return w;
    }
}
```

Coroutine queue

The often occurring problem with Unity projects is that, when switching scenes, everything is destroyed. All the references to rendered sprites, game objects and even scripts are destroyed. But there often is some code that needs to be executed when the player arrives in the next scene. This problem has plagued this project for the longest time.

For example when going to a school lecture, the scene changes, but when the scene changes back to campus, how do we know what to execute? This was from the beginning solved by having massive hardcoded switch blocks, the input of which was ”previous scene”. This is hilariously bad as it required inclusive lists of all scenarios of scene switching that could happen, something like teleporting the player home after sleeping on the street would be impossible to catch.

After many iterations a quite elegant* solution emerged. A CoroutineQueue, this object would be accessible through the Status Controller which was not destroyed on scene changes. As input, it would receive a lambda function*, which returned true when executed, after execution it would be removed. This abstract piece of code allowed for trans-scene code execution

Code listing 6.11: Coroutine queue

```
public delegate bool WaitingForSceneEvent(string sceneName);
public event WaitingForSceneEvent OnSceneChange;

public List<WaitingForSceneEvent> list;

private IEnumerator OnEnableCoroutine()
{
    yield return new WaitUntil(() => StatusController.initialized);
    OnSceneChange += CheckWaiting;
}

bool CheckWaiting(string sceneName)
{
    list.RemoveAll(x => x(sceneName) == true);
    return true;
}
```

*Subjectively elegant, this is still not perfect as the code written here has no guarantees what objects are available.

*In C# a delegate is used

CHAPTER 7

User testing

The testing has been done in two phases. The first happened before any changes were made in the scope of this thesis (Section 3.1). The second has been done after the changes. From this testing, we should find whether our changes were effective, whether our UX design improved or whether further modifications need to be made.

7.1 Testing description

The needed user testing has been compiled into a form, which included a link to the game (on the continuous delivery platform) and a questionnaire. The survey was conducted through Google forms, a table encapsulating given questions and received answers, will be available on the transferable media attached to this thesis.

7.1.1 The questionnaire questions

The questionnaire contained a pool of questions used for both my and David Mikulka's thesis [1]. There was a place provided, at the end of the questionnaire, for ideas on how to improve the game.

Introductory questions

These questions gauge the audience taking the questionnaire. The game has been tested by students and professors of CTU FIT.

- What is your gender?
- How old are you?
- Have you played FitLife?

Answer summary

There were 46.2% Men (6 people), 38.5% Women (5), and 15.4% (2) who chose not to share their gender.

For the age group representation we ended up with 76.9% 19-29 (10), 15.4% 13-18 (2) and 7.7% 29-39 (1).

The last question if answered yes will let the questionee onto the rest of the questions, it will end the questioning if otherwise. There were 84.6% who played the game (11), and 15.4% who did not (2).

General questions

These were a compilation of both game design and game mechanics questions. I will be focusing on those that matter in the scope of this thesis.

- Did you encounter a bug during your play that forced you to restart the game?
- Are the controls intuitive and understandable?
- Is the quest system intuitive and uncluttered? Please state any problems
- How would you rate the rate of time and stats decay?
- Did you finish the game?

Answer summary

No one encountered any game-breaking bugs which required the game to be restarted. I find this to be a great success. This game has many edge-cases and before the big refactoring would encounter random exceptions on every corner.

Everyone found the controls intuitive and understandable, which also means that they worked well.

The quest system was rated 90.9% intuitive and uncluttered (10), and 9.1% No (1). There didn't seem to be any functional problems.

The rate of decay was rated 54.5% neutral (6), 27.3% a little fast(3), and 18.2% very slow (2).

The game was finished by 81.8% (9) people, the rest 18.1% (3) people found the game too boring or too long.

Implemeted changes following testing

People seemed to be confused by repeating dialogues on every new day, this has been promptly hot-fixed, as this was done deliberately to remind the player that a quest is available in that spot, this reportedly unintuitive behavior was described in section 4.2.5.

The testing showed a problem in game design with the unavailability of replenishing stats at night. After 18:00 when the Coffee shop and Canteen closed the player would not be able to replenish their stats until the next day. This was fixed by sleep replenishing energy and moving the closing of the establishments to 21:00, matching up with the sleep time.

7.2 Testing conclusions

The testing of FitLife was conducted with unmoderated user testing. Eleven people participated in the testing and gave valuable feedback on their experiences. The questionnaire questions aimed at the shortcomings of the previous version of the game. The testing showed that issues reported surrounding various UX problems, described in section3.1, were fixed and improved. The testers seemed to have a sense of direction, which was lacking previously.

The game seems to be for the most part, enjoyable. I was relieved to find that the game performed functionally without any discernable issues, but saddened that the game still has problems that cannot be simply fixed with code. The game was found by 2 people to be boring/too long. Sadly due to time reasons, we cannot fix this issue in the scope of this thesis.

The issues with FitLife do not seem to be rooted in implementation or functionality errors anymore, but in game design. For further project development, an improved game design together with more varying content is to be developed.

Conclusion

This thesis aimed to finalize the game of FitLife, a game about studying at FIT. Inspiration for how a game like this should be implemented was taken from similar games in the same genre. During the analysis of Design patterns, it was discovered that Design patterns are useful and applicable in game development. The game was a product of SP1 and SP2 subjects, where the game was in the hands of eight different people. FitLife was unrefined and rough, which was reflected in the user reviews after this period.

The design tackled issues regarding stability and lack of functionality of selected back-end components. Design patterns were used in the designing of each of the components. These components were then charted using class and activity diagrams.

The implementation consisted of refactoring and fixing of nearly all aspects of the game. The game was much more functional and stable as it headed into the second phase of user testing.

In the final user testing phase, there was no regression of issues from the first user testing, the testing finalized with players rating the game mostly positively. The feedback from this phase touched mostly on game design issues, which require time, that was sadly unallocated to correct.

FitLife can definitely be improved upon in the future with more world to explore, with more quests, with better game design. But hopefully in its current state, it can bring the target audience and the reader some enjoyment.

Bibliography

[1] Mikulka, D.: *FitLife - hra o studiu na FIT*. Bachelor's thesis, Czech Technical University in Prague, Faculty of Information Technology, 2022.

[2] Nintendo: Mario history. 1985, [Online. Accessed: 2022-04-26]. Available at: `https://mario.nintendo.com/history/`

[3] Zelda fans: Zelda Wiki. 2007, [Online. Accessed: 2022-04-26]. Available at: `https://zelda.fandom.com/wiki/Locations_in_A_Link_to_the_Past`

[4] Electronic Arts: SimCity 2000. 2016, [Online. Accessed: 2022-04-26]. Available at: `https://www.ea.com/games/simcity/simcity-2000`

[5] ConcernedApe: Official Stardew Valley website. 2016, [Online. Accessed: 2022-03-20]. Available at: `https://www.stardewvalley.net`

[6] Pokemon: Official Pokémon website. 2007, [Online. Accessed: 2022-03-17]. Available at: `https://www.pokemon.com/us/pokemon-video-games/pokemon-diamond-version-and-pokemon-pearl-version/`

[7] Hemmann, K.: Mythical Landscapes and Imaginary Creatures: Pokémon and Japanese Regionalism. *Proceedings of the Association for Japanese Literary Studies, Universidade de Notre Dame*, ročník 14: s. 261–271.

[8] Tobyfox: Steam store. 2015, [Online. Accessed: 2022-03-17]. Available at: `https://store.steampowered.com/app/391540/Undertale/`

[9] Seraphine, F.: Ethics at Play in Undertale: Rhetoric, Identity and Deconstruction. In *DiGRA Conference*, 2018.

[10] Square Enix: Steam store. 2019, [Online. Accessed: 2022-03-19]. Available at: `https://store.steampowered.com/app/921570/OCTOPATH_TRAVELER/`

[11] Qu, J.; Song, Y.; Wei, Y.: Design patterns applied for game design patterns. In *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 2016, s. 351–356, doi:10.1109/SNPD.2016.7515924.

[12] Gamma, E.; Helm, R.; Johnson, R.: *Design patterns elements of Reusable Object Oriented Software*. Addison Wesley, 1998.

[13] Nikolaeva, D.; Safi, M.; Mihailov, M.; aj.: Algorithm A and Design Patterns used in Unity Video Game development. In *2020 International Conference Automatics and Informatics (ICAI)*, 2020, s. 1–3, doi: 10.1109/ICAI50593.2020.9311327.

[14] Qu, J.; Wei, Y.; Song, Y.: Design patterns applied for networked first person shooting game programming. In *15th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, 2014, s. 1–6, doi:10.1109/SNPD.2014.6888715.

[15] Szallies, C.: On using the observer design pattern. *XP-002323533,(Aug. 21, 1997)*, ročník 9, 1997.

[16] John French: Game dev beginner blog. 2021, [Online. Accessed: 2022-04-4]. Available at: `https://gamedevbeginner.com/singletons-in-unity-the-right-way/`

[17] Unity technologies: Unity documentation. 2020, [Online. Accessed: 2022-04-3]. Available at: `https://docs.unity3d.com/Manual/webgl-building.html`

[18] Google: Google Research - Magenta. 2016, [Online. Accessed: 2022-04-4]. Available at: `https://magenta.tensorflow.org`

[19] Github: Github Actions. 2019, [Online. Accessed: 2022-04-29]. Available at: `https://docs.github.com/en/actions`

[20] GameCI: Unity Actions. 2019, [Online. Accessed: 2022-05-01]. Available at: `https://github.com/game-ci/unity-actions`

[21] Unity technologies: Unity. 2017, [Online. Accessed: 2022-04-29]. Available at: `https://unity.com/unity/features/editor/art-and-design/cinemachine`

[22] Microsoft: Microsoft docs. 2017, [Online. Accessed: 2022-04-29]. Available at: `https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/delegates/`

APPENDIX A

Acronyms

CD	Continuous delivery.
CI	Continuous integration.
CTU	Czech Technical University.
FIT	Faculty of Information Technology.
NPC	Non player character.
PC	Personal computer.
RPG	Role playing game.
UI	User interface.
UX	User experience.

APPENDIX B

Contents of enclosed SD-Card

```
│ readme.txt .................. the file with SD-Card contents description
├─ src ....................................... the directory of source codes
│  ├─ impl ....................................... implementation sources
│  └─ thesis .............. the directory of LaTeX source codes of the thesis
└─ text ........................................... the thesis text directory
   ├─ thesis.pdf ........................... the thesis text in PDF format
   └─ fitlife_questionnaire.xlsx ... google forms questions and answers
```