

Archetype Games

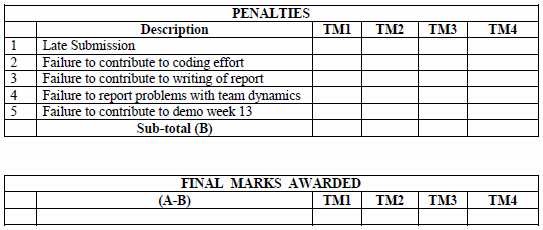
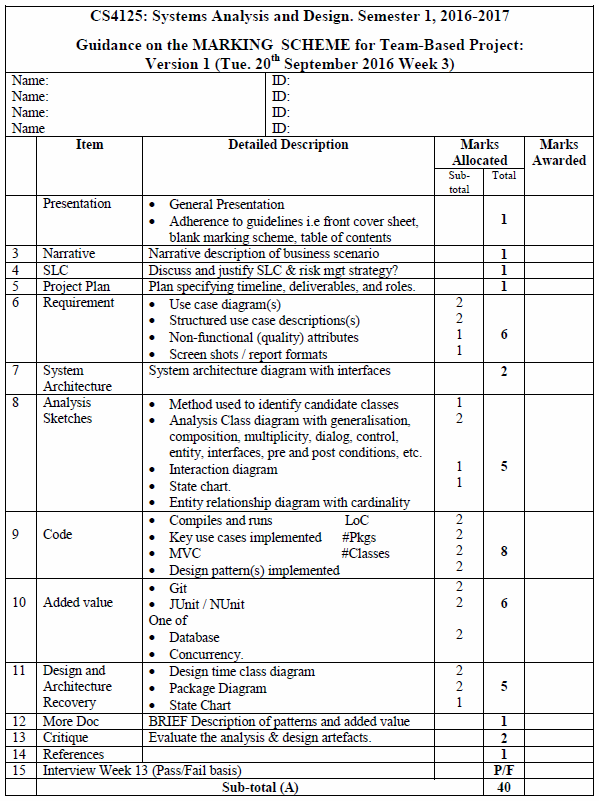
CS4125: System Analysis and Design

Dean Maloney (14140306)

Fergal Bowers (14151324)

Philip Waldron (14173026)

Pawel Pencherzewski (14199041)



# Table of Contents

Narrative Description

Software Life Cycle

Waterfall

V-Model

Agile

Project Plan

Requirements

Use Case Diagrams

Use Case Descriptions

Structured Use Case Descriptions

Non-Functional Requirements

Discussion on Tactics to Support Quality Attributes

Screen Shots

System Architecture

Discussion

Architectural Decisions Taken

UML Workbench

Implementation Language

Architecture Diagram

Analysis Sketches

Identifying Candidate Classes

Class Diagram

Interaction Diagram

State Chart

Entity Relationship Diagram

Code Implementation

Added Value

Git

Junit/NUnit

Database/Concurrency

Design and Architecture Recovery

Design Time Class Diagram

Package Diagram

State Chart

Description of Patterns and Added Value

Critique

References

# Narrative Description

Overview  
The gaming industry has been undergoing a revolution over the past two decades, as highly-established companies such as Sony and Microsoft contribute quality products using the latest technology to gaming communities across the globe. Gaming on PC has also only increased in popularity over recent years through platforms such as Steam and Origin, and communities are much more plentiful on this platform due to the benefits of having a keyboard to type messages to fellow gamers.   
  
However, it is true to say that an alarmingly high amount of highly-anticipated software is regularly abandoned by the community for a variety of reasons. For example, with the downloadable content (DLC) structure becoming more and more popular, companies have focused on profits ahead of their community. By offering items which increase the player’s chances of being successful while playing the game in exchange for real-life money, dubbed as pay-to-win (P2W), developers alienate their community, leading to falling player bases and frustration amongst those who expected a satisfying multiplayer experience. Payday 2, an immensely popular online game, adapted a controversial P2W DLC schedule without consulting the community, leading to players permanently abandoning the game and outrage on online gaming communities such as Reddit and Steam forums. They were forced to rescind their changes a couple months later, but the damage had already been done. Pokémon Go similarly had a release plagued with server issues, and made the issue worse by not appointing a public relations officer to communicate with the player base. The game’s initial hype died very suddenly due to the resulting anger and confusion amongst the community who wanted answers.  
  
It is important to look back at how communities were successfully built surrounding the release of a multiplayer game and its legacy. Counter-Strike, a game that was originally a modification of the revolutionary Half-Life, was developed with the community in mind. Its player base soared, which prompted Half-Life’s developers, Valve, to develop a Counter-Strike series. To this day, all four of the main-series Counter-Strike games have a large number of regular online players and are regularly updated, with Counter-Strike: Global Offensive (CS:GO), the most recent entry in the series, being the fourth most popular streamed game on Twitch, the current leading game streaming website. While games which have a big impact at conventions such as E3 are often anticipated to become the “next big thing” to dethrone games such as CS:GO, they usually collapse due to repetitive gameplay or due to the aforementioned DLC issue. None of the Counter-Strike games implement a P2W system (CS:GO does include DLC weapon skins, but these do not affect gameplay), instead solely relying on the player’s skill and knowledge of the game. This has caused the game to be a massive hit in the competitive scene of eSports, a gigantic community of its own which American cable company ESPN has shown their interest in.

We at Archetype Games take inspiration from the successes and learn from the mistakes of the disasters throughout gaming’s incredible history. We seriously value public relations and the wishes of the community as we focus on implementing whatever project lies ahead of us. By supplying the player base with incentives such as leader boards, patches, competitions and regular news updates through our client service, we make the greatest effort to ensure the needs, wants and expectations of the loyal player are presented and exceeded. We also offer a platform for the community to communicate amongst each other, befriend each other and to clash in a competitive game. We value the skill of the player, and we ensure that they have every opportunity to display it.

# What is Archetype Games?

Archetype Games offers a client program that offers players numerous ways to engage with others in a competitive or friendly manner. It establishes itself as a hub for gaming and communication purposes, and it allows users to register and join a game, view leader board entries, enter a chat room or maintain a friends list.   
  
The client communicates with the server, which maintains data for each user and handles requests and matchmaking services.

# Software Lifecycle

The team considered numerous software lifecycle models during the planning stage of implementing the project, and we eventually came to the conclusion that the Waterfall Model and V-Model were not the correct approaches.

# Waterfall Model

We wanted to remain as flexible as possible in case we realized during the implementation phase that the resources cannot be dedicated to implementation a planned functionality. We instead sought a lifecycle model which allows revisions of design instead of forcing the implementation into the wrong direction.

-To quote David Parnas, “Many of the [system’s] details only become known to us as we progress in the [system’s] implementation. Some of the things that we learn invalidate our design and we must backtrack.”

With the Waterfall Model, there would have been no adaptability to any requirements that may be re-defined by client. If we wanted to make a change to the requirements once the design stage was completed, we would have no choice but to modify the design to allow the requirements change to occur. Dedicating resources towards this would not be worthwhile, and this flaw was a deciding factor as we decided against the Waterfall Model.

# V-Model

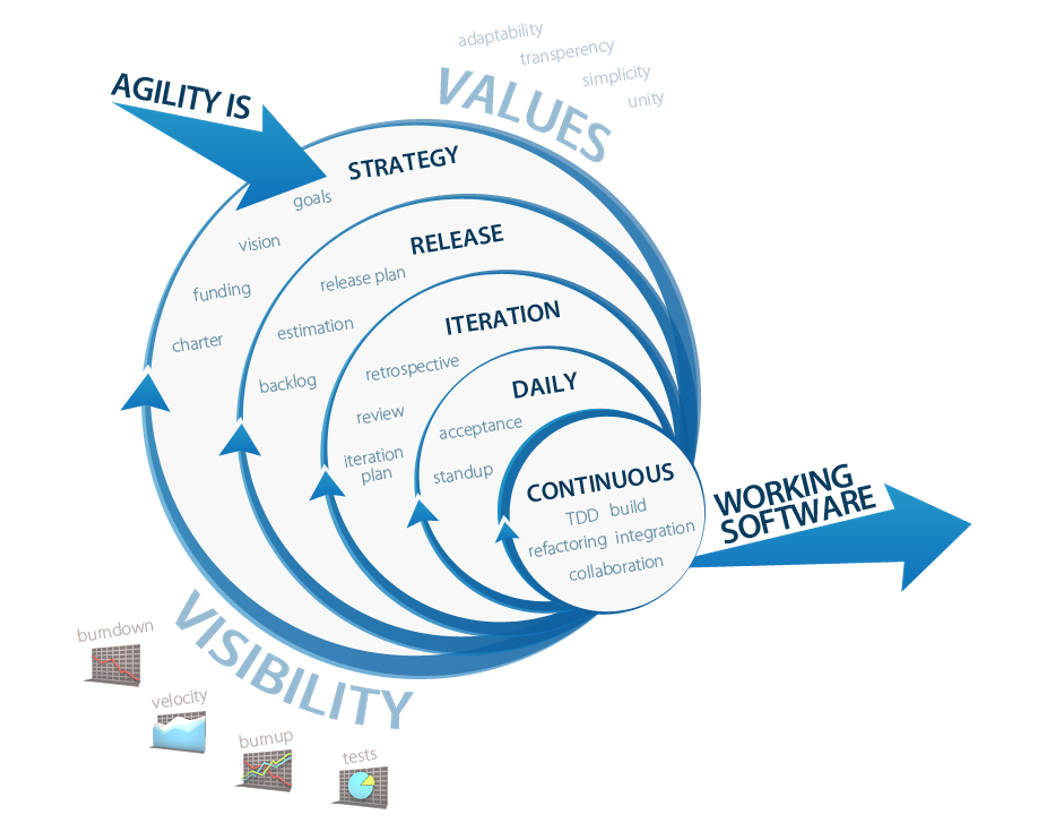
Since we wanted flexibility while implementing the project, we couldn’t ignore the V-Model’s rigid and linear nature. It does not natively allow us to respond to change, which would prove to be crucial during implementation. It also shares many similarities with the Waterfall Model and therefore maintains its drawbacks.

We wanted to be as efficient as possible once we reached the testing stage, and the V-Model does not allow for opportunistic approaches when it comes to finding the most effective way to execute testing our implemented project.

# Agile Software Development Model

Agile remained amongst our final choices for our software lifecycle model, and stood out as the optimal choice. The project’s nature required a software lifecycle model which was as flexible as possible and supported responding to change. The project also has plenty of room for expansion due to patches that will be released for the client throughout the project’s lifespan.  
  
The Agile Software Development Model was exactly what we needed to apply an iterative and incremental approach to development. It allows us to test and implement new features into the project to suit the user’s needs and to respond to public demand. The aim is to maintain a healthy community and player base for a long period of time, and Agile assists us in remaining successful in that regard.

A diagram follows which illustrates how we approached the development of our project under the Agile Software Development Model.

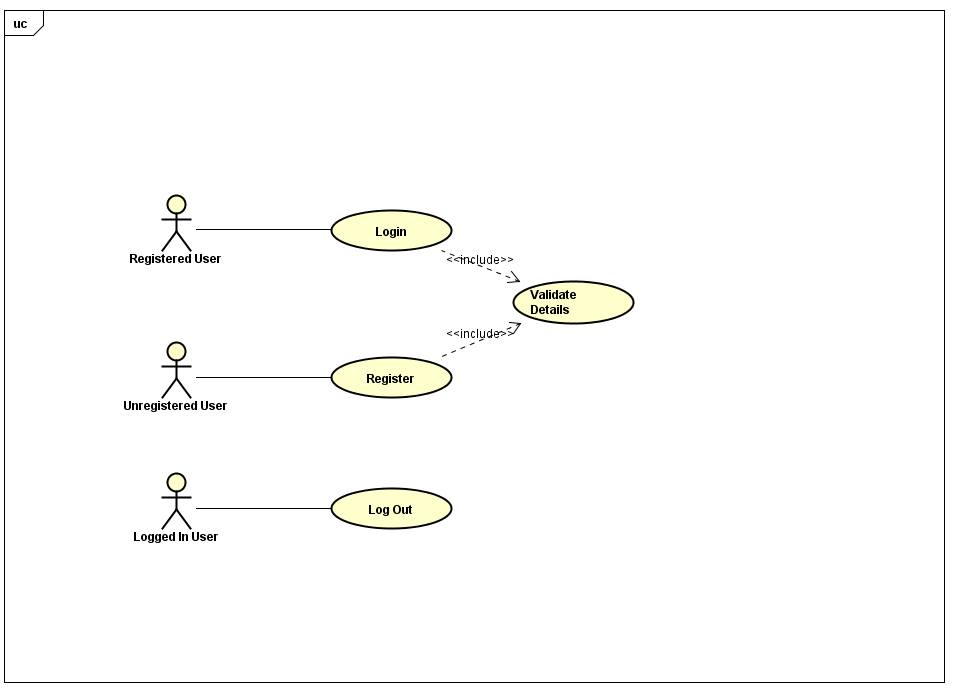


# Project Plan

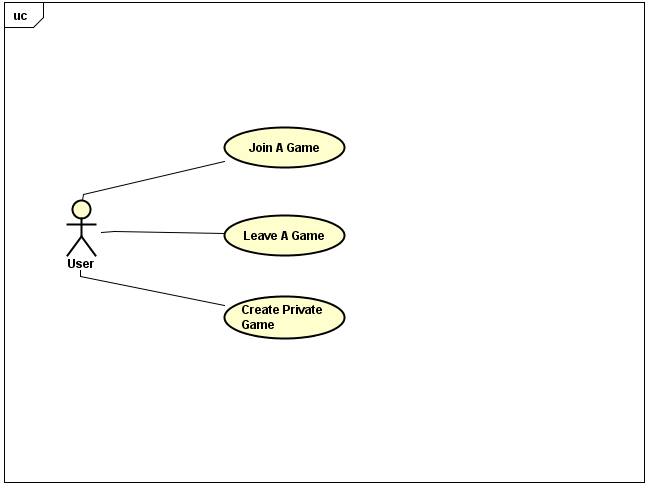
|  |  |  |  |
| --- | --- | --- | --- |
| **Heading** | **Description** | **Allocated to** | **Week** |
| Presentation | Company logo/design cover page | Philip | 7 |
| Narrative Description | Narrative description of business scenarios | Dean | 5 |
| Software Life Cycle | Discussion of software model used | Dean | 5 |
| Project Plan | Specifying jobs and roles | Philip | 4 |
| Requirements | Use case diagrams  Use case descriptions  Structured use case descriptions  Non-functional requirements  Tactics to support quality attributes  Screen shots of GUI | Dean  Fergal  Philip, Pawel  Philip  (?)  Group (?) | 6  6  6  6  10 |
| System Architecture | Discussion  Architecture diagram with interfaces | (?)  Group | 7  7 |
| Analysis Sketches | Identify candidate classes  Class diagrams  Interaction diagram  State chart  Entity relationship diagrams | Philip  Pawel (?)  Pawel (?)  Pawel (?)  Pawel (?) | 7 |
| Code | Code implementation | Group | 7 - 10 |
| Added Value | Git  JUnit/NUnit  Database/Concurrency | Philip  (?)  (?) | 10 |
| Design and Architecture Recovery | Architectural diagram  Design time class diagrams  State chart | Fergal (?)  Fergal (?)  Fergal (?) | 10  10  10 |
| Patterns and Added Value | Brief description of patterns and added value | Dean | 10 |
| Critique | Evaluate the analysis and design artefacts | Dean (?) | 10 |
| References | Sources used for learning and information | Group (?) | 10 |

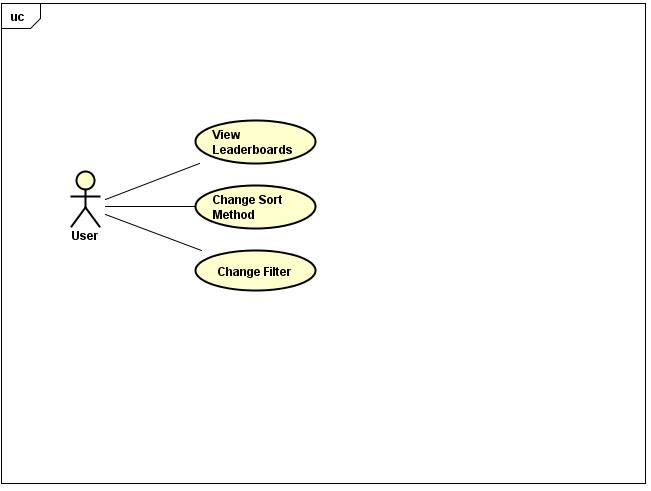
# Requirements

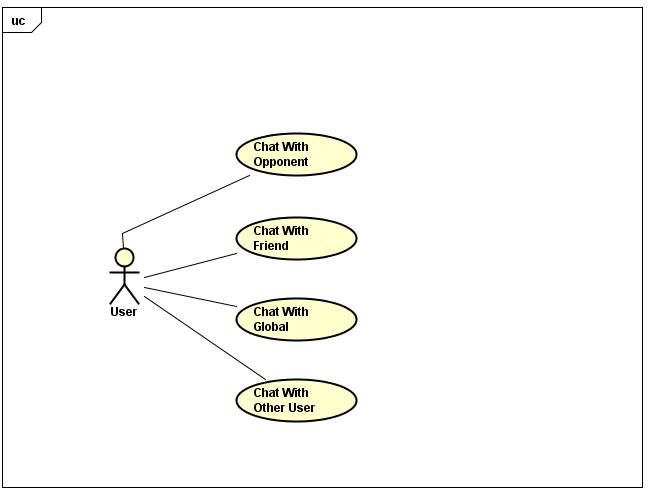
# Use Case Diagrams











# Use Case Descriptions

**Use Case 01:** Chat in Global

**Description:** A user sends a message to the global chat room

**Actors:** User

**Pre-Conditions:**

1. User is logged in.
2. User is connected to the global chat room

**Flow:**

1. User types a message into the text field under global chat.
2. User presses "Send".
3. The Message is send to the chat room.

**Alternate Flow:**

User types a message into the text field under global chat.

1. Request denied: Cannot connect to chat.

**Post-Conditions:**

1. User's message is added to the global chat log.

Non Functional Requirement: Performance  
The server should receive and send the message within 3 seconds.

**Use Case 02:** Chat in Private

**Description:** A user sends a message privately to another player

**Actors:** User, User

**Pre-Conditions:**

1. User is logged in.
2. Other user is online.

**Flow:**

1. User selects other user from playerlist.
2. User selects "Private Message".
3. New chat window is created.
4. User types a message into the text field under private chat.
5. User presses "Send".
6. Message is sent to other user.

**Alternate Flow:**

1. User presses "Send".
2. Request Denied: Other User is offline.
3. New chat window is created.
4. If previous messages were sent between users, populate chat window with past messages

**Post-Conditions:**

1. Users message sent to other user

Non Functional Requirement: Performance  
The server should receive and send the message within 3 seconds.

**Use Case 03:** Chat with Friend

**Description:** A user sends a message to a user on friends list.

**Actors:** User

**Pre-Conditions:**

1. User is logged in.
2. Other User is online.
3. Other user is on user's friends list.

**Flow:**

1. User selects other user from friends list.
2. User selects "Private Message".
3. New chat window is created.
4. User types a message into the text field under private chat.
5. User presses "Send".
6. Message is sent to other user from friends list.

**Alternate Flow:**

1. User presses "Send".
2. Request Denied: Other User is offline.
3. New chat window is created.
4. If previous messages were sent between users, populate chat window with past messages

**Post-Conditions:**

1. User’s message sent to other user from friends list.

Non Functional Requirement: Performance  
The server should receive and send the message within 3 seconds.

**Use Case 04:** Chat in lobby

**Description:** A user sends a message to all users in game lobby

**Actors:** User

**Pre-Conditions:**

1. User is logged in.
2. User is in a game lobby.

**Flow:**

1. User types a message into the text field under lobby chat.
2. User presses "Send".
3. The Message is send to the lobby chat room.

**Alternate Flow:**

User types a message into the text field under lobby chat.

1. Request denied: Cannot connect to chat.

**Post-Conditions:**

1. User's message is added to the lobby chat log.

Non Functional Requirement: Performance  
The server should receive and send the message within 3 seconds.

# Structured Use Case Descriptions

|  |  |  |
| --- | --- | --- |
| **USE CASE 05** | View Leaderboards | |
| **Goal in Context** | The leaderboards are displayed to the user in another window | |
| **Scope & Level** |  | |
| **Preconditions** | None | |
| **Success End Conditions** | The leaderboards are displayed | |
| **Failed End Condition** | The leaderboards are not displayed | |
| **Primary, Secondary, Actors** | User | |
| **Trigger** | ViewLeaderboards request comes in | |
| **DESCRIPTION** | **Step** | **Action** |
| 1 | User selects View Leaderboards button |
| **EXTENSIONS** | **Step** | **Branching Action** |
| 1a | User wants to view leaderboards:  1a. A new window opens, populated with all registered users |

Non Functional Requirement: Usability  
The UI should be easy to use for the intended user.

|  |  |  |
| --- | --- | --- |
| **USE CASE 06** | Change Sort Method | |
| **Goal in Context** | The leaderboards are displayed in a desired sorted order | |
| **Scope & Level** |  | |
| **Preconditions** | Leaderboards are currently displayed | |
| **Success End Conditions** | Sort method is changed | |
| **Failed End Condition** | The sort method is not successfully changed | |
| **Primary, Secondary, Actors** | User | |
| **Trigger** | ChangeSortMethod(sort) request comes in | |
| **DESCRIPTION** | **Step** | **Action** |
| 1 | User selects the method in which to sort the leaderboards |
| **EXTENSIONS** | **Step** | **Branching Action** |
| 1a | Sort method changed:  1a. The leaderboards are repopulated in the relevant order |
| **VARIATIONS** | **Step** | **Branching Action** |
| 1 | User may sort by:  Alphabetical order  Most wins  Highest win/loss ratio |

Non Functional Requirement: Extensibility  
More sort methods should be easy to add in future iterations.

|  |  |  |
| --- | --- | --- |
| **USE CASE 07** | Change Filter | |
| **Goal in Context** | A filter is applied and the leaderboards are displayed taking into account the new filter | |
| **Scope & Level** |  | |
| **Preconditions** | Leaderboards are currently displayed | |
| **Success End Conditions** | Leaderboards are displayed in relation to the filter | |
| **Failed End Condition** | Leaderboards are not filtered correctly | |
| **Primary, Secondary, Actors** | User | |
| **Trigger** | ChangeFilter request comes in | |
| **DESCRIPTION** | **Step** | **Action** |
| 1 | User types in a filter |
| 2 | User selects a button to apply the filter |
| **EXTENSIONS** | **Step** | **Branching Action** |
| 2a | Filter is applied:  1a. The leaderboards are repopulated taking to account the filter |

|  |  |  |
| --- | --- | --- |
| **USE CASE 08** | Add a Friend | |
| **Goal in Context** | Causes a pop up window to appear where the user can enter a name of another user to add as a friend. | |
| **Scope & Level** |  | |
| **Preconditions** | We know: adding user and the user being added | |
| **Success End Conditions** | One user successfully sent a friend invite, the other received it | |
| **Failed End Condition** | The invitation has not been sent/ the end user has not received an invitation | |
| **Primary, Secondary, Actors** | User sending and user receiving a friend invitation | |
| **Trigger** | AddFriend request comes in | |
| **DESCRIPTION** | **Step** | **Action** |
| 1 | User sends a friend invitation by entering a name |
| 2 | The other user receives a friend invitation through “respond to invite” tab |
| 3 | Receiving user can accept or decline the invitation |
| **EXTENSIONS** | **Step** | **Branching Action** |
| 1a | A user does not exist:  1a1. Display error and give another try |

Non Functional Requirement: Usability  
It should be easy to add the intended friend.

|  |  |  |
| --- | --- | --- |
| **USE CASE 09** | Remove a friend | |
| **Goal in Context** | Causes a pop up window to appear where the user can enter a name of a user he/she wants to remove from friends list and him/herself from the other user’s f/l. | |
| **Scope & Level** |  | |
| **Preconditions** | We know: the user removing and the user being removed | |
| **Success End Conditions** | A user successfully removed another user from their friends list and removed himself from their f/l. | |
| **Failed End Condition** | The user entered to be removed from the friends list has not been removed. | |
| **Primary, Secondary, Actors** | User removing a friend and the friend being removed | |
| **Trigger** | RemoveFriend request comes in | |
| **DESCRIPTION** | **Step** | **Action** |
| 1 | User enters a name that he/she wishes to remove from friends list. |
| 2 | Both users’ friends list are updated, removing each other from them. |
| **EXTENSIONS** | **Step** | **Branching Action** |
| 1a | A user does not exist:  1a1. Display error and allow another try |

Non Functional Requirement: Usability  
it should be easy to remove the intended friend.

|  |  |  |
| --- | --- | --- |
| **USE CASE 10** | Invite to a Game | |
| **Goal in Context** | Causes a pop up window to appear where he/she can enter a name of another user to send a game invite to. | |
| **Scope & Level** |  | |
| **Preconditions** | We know: inviting user and the user being invited | |
| **Success End Conditions** | One user successfully sent a game invite, the other received it | |
| **Failed End Condition** | The invitation has not been sent/ the end user has not received an invitation | |
| **Primary, Secondary, Actors** | User sending and user receiving the game invitation | |
| **Trigger** | InviteToGame request comes in | |
| **DESCRIPTION** | **Step** | **Action** |
| 1 | User sends a game invitation by entering a name |
| 2 | The other user receives a game invite through “respond to invite” tab |
| 3 | Receiving user can accept or decline the invitation |
| **EXTENSIONS** | **Step** | **Branching Action** |
| 1a | A user does not exist:  1a1. Display error and allow another try |
| 1b | A user is not online:  1b1. Display error and allow another try |

|  |  |  |
| --- | --- | --- |
| **USE CASE 11** | Respond to Invite | |
| **Goal in Context** | The user is displayed a list of friend and game invitations, which he/she can accept, decline or ignore. | |
| **Scope & Level** |  | |
| **Preconditions** | We know: the user, his/her invitations | |
| **Success End Conditions** | All invitations are displayed correctly and the user is able to respond to them successfully.  Responding to an invite removes it from the list. | |
| **Failed End Condition** | The invitations don’t display or are displayed incorrectly.  The invitations do not respond to actions taken. | |
| **Primary, Secondary, Actors** | The user viewing the invitations | |
| **Trigger** | RespondToInvite request comes in | |
| **DESCRIPTION** | **Step** | **Action** |
| 1 | User presses a Respond to Invite tab/button. |
| 2 | A list of invitations for this user is displayed |
| 3 | Accepting/declining an invitation updates the invitation list. |
| 4 | If user accepted friend invitation the friend lists of both users are updated |
| 5 | If user accepted a game invite he/she is sent to a game screen. |
| **EXTENSIONS** | **Step** | **Branching Action** |
| 5a | User sending invitation is offline:  5a1. Display information that user is offline |

Non Functional Requirement: Performance  
Friend list updates should occur within 3 seconds.  
Game lobby should be joined within 3 seconds.

|  |  |  |
| --- | --- | --- |
| **USE CASE 12** | Nickname Friend | |
| **Goal in Context** | User adds a nickname beside another friend’s username in his/her friend list. | |
| **Scope & Level** |  | |
| **Preconditions** | We know: user adding nickname, the names on his f/l | |
| **Success End Conditions** | User successfully appended a nickname to a friend’s username in his/her friend list. | |
| **Failed End Condition** | No nickname is displayed beside the friend that the nickname was added to. | |
| **Primary, Secondary, Actors** | The user adding the nickname | |
| **Trigger** | AddNickname request comes in | |
| **DESCRIPTION** | **Step** | **Action** |
| 1 | User presses Nickname a friend button |
| 2 | User types in the name of a friend he wants to add a nickname to |
| 3 | User accepts changes made |
| **EXTENSIONS** | **Step** | **Branching Action** |
| 2a | A user does not exist:  2a1. Display error and allow another try |
| 3a | The nickname entered is too long:  3a1. Display error and exit “nicknaming” action |

Non Functional Requirement: Usability  
it should be easy to nickname the intended friend.

|  |  |  |
| --- | --- | --- |
| **USE CASE 13** | View Friend Availability | |
| **Goal in Context** | User is displayed with a list of his/her friends, as well as their online status. | |
| **Scope & Level** |  | |
| **Preconditions** | We know: A user, his/her friends and their online statuses | |
| **Success End Conditions** | The friends are correctly displayed with correct online statuses for every individual user | |
| **Failed End Condition** | The friends are displayed incorrectly or not at all.  The online statuses are incorrect | |
| **Primary, Secondary, Actors** | User viewing his/her friends list. | |
| **Trigger** | ViewFriends request comes in | |
| **DESCRIPTION** | **Step** | **Action** |
| 1 | User presses View friends button/tab. |
| 2 | A list of all friends is displayed to the user with correct online statuses |

# Additional Non Function Requirements

The main menu should be extensible.  
It should be easy to add more functions on future iterations.

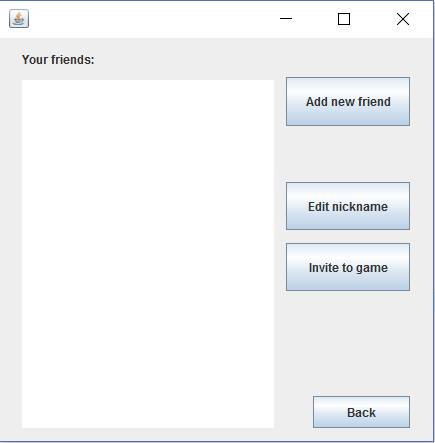
The program should support maintainability.  
It should be easy to fix defects, bugs and keep the service operational.

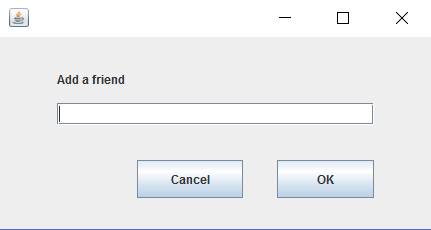
The program should be robust.  
Modules should be able to cope with errors during execution.

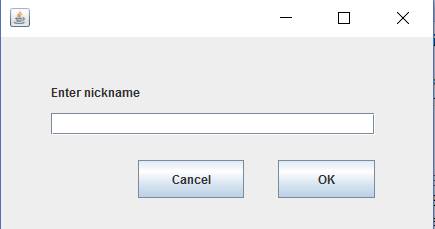
The program should be of good quality.  
It should be fit for its intended uses.

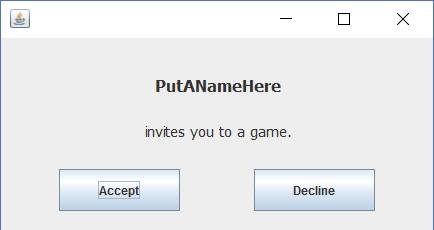
# Discussion on Tactics to Support Quality Attributes

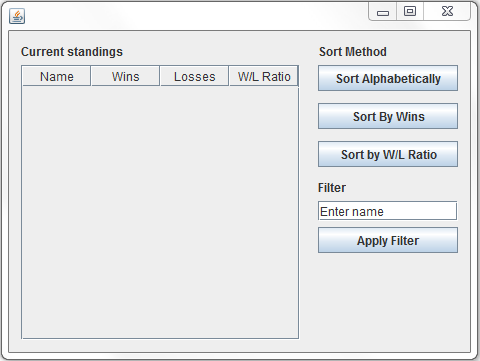
# Screen Shots







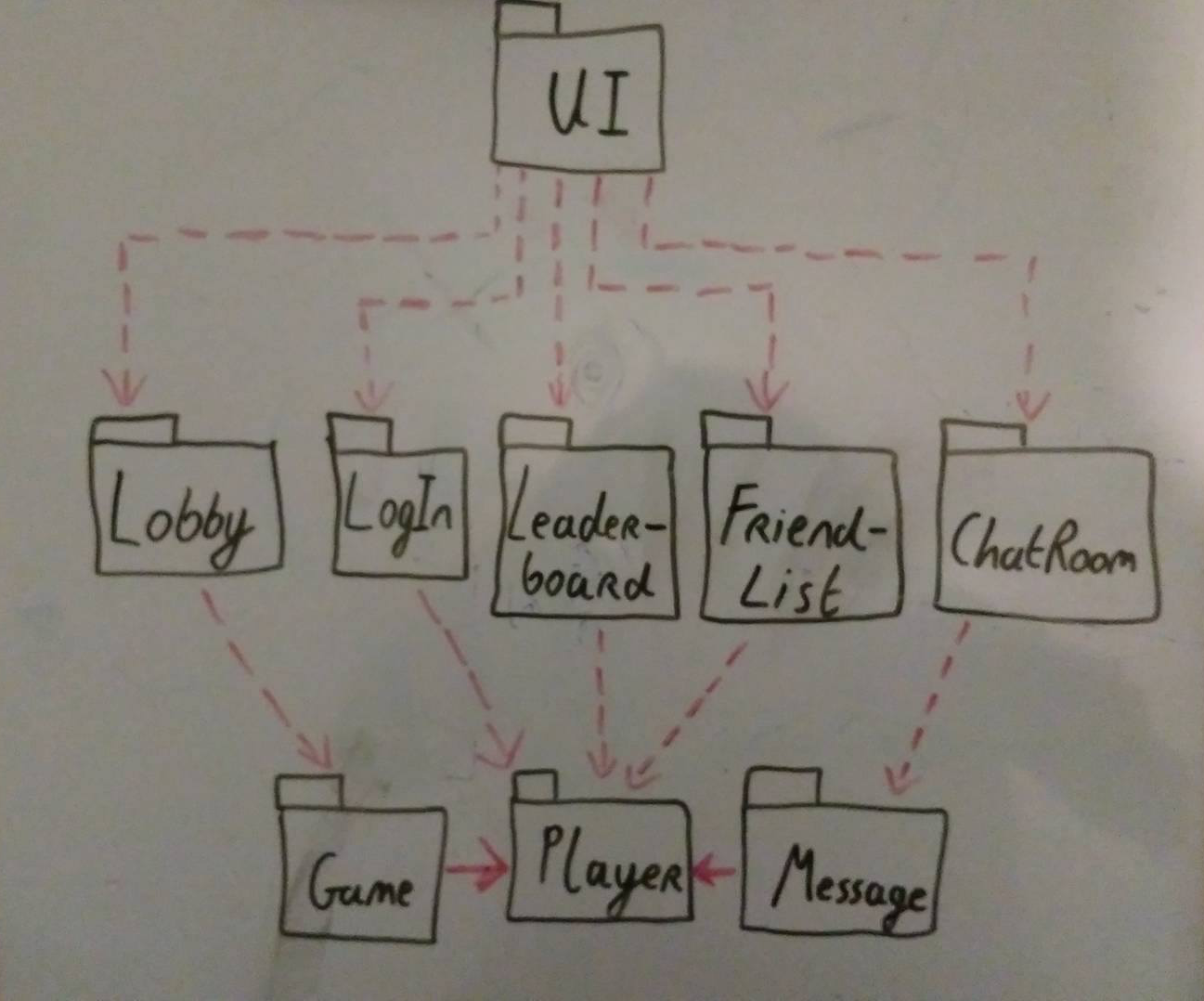




# System Architecture

# Discussion

# Architecture Diagram



# Analysis Sketches

# Identify candidate classes

Using a data driven design method, we came up with a list of potential candidate classes by using the noun identification technique.

Initial list:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Username | Password | Registration | Logging | Game | System |
| Play | Invite | Message | Profile | User | Notification |
| Request | Send | Accept | Decline | User Interface | Display |
| Friend | Friendlist | Dialog | Input | Leaderboard | Lobbylist |
| Chatroom | Player | User | Text | Connection |  |

Heuristics:

1. Too vague or to specific – RED
2. An operation – BLUE
3. An attribute - Orange
4. Out of Scope - GREEN
5. Other similar objects – Purple

Filtered list:

|  |  |  |  |
| --- | --- | --- | --- |
| User | Player | Game | Leaderboard |
| Lobbylist | Friendlist | Friend | Chatroom |
| Message | Invite | User Interface | Log |

## User

A user encompasses all the data related to the logged in user who uses the application. Their data is updated after each game they complete.

## Player

A user who is matched into a game becomes a player, accepting their inputs to play the game.

## Game

The game is the logic for the game to be played.

## Leaderboard

The leaderboard encompasses all the data to be displayed on the leaderboard, letting data displayed be sorted by various sorting methods.

## Lobbylist

Lobbylist keeps track of all open connections, filtering out unavailable connections (already in game, etc.).

## Friendlist

Friendlist keeps track of all the added friends of the current user.

## Friend

Friend encompasses the data of a friend of the user, allowing certain elements to be edited such as a nickname.

## Chatroom

Chatroom handles messages that the user receives from other users, friends, the server etc. and allows the user to input messages to be sent to their desired destination.

## Message

A message encompasses the data relevant to a sent message; its contents, sender and destination.

## Invite

There are two types of invites:

1. Game invite: Allows a user to challenge a friend directly to play a game, and allows the recipient to choose to accept.
2. Friend invite: Allows a user to send a friend request to another user, and allows the recipient to choose to accept.

## UI

Handles all graphical elements that the user may interact with once they launch the application.

## Log

Log is for taking input information from the user. It encompasses users logging in, out and registering.

# Class diagrams

# Interaction diagram

# State chart

# Entity relationship diagrams

# Code Implementation

# Overview of Contribution to Code

|  |  |  |  |
| --- | --- | --- | --- |
| **Package** | **Class** | **LOC** | **Author** |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  | | **Total LOC** | **Author** |
|  | Dean |
|  | Fergal |
|  | Philip |
|  | Pawel |
|  | **Total** |

# Added Value

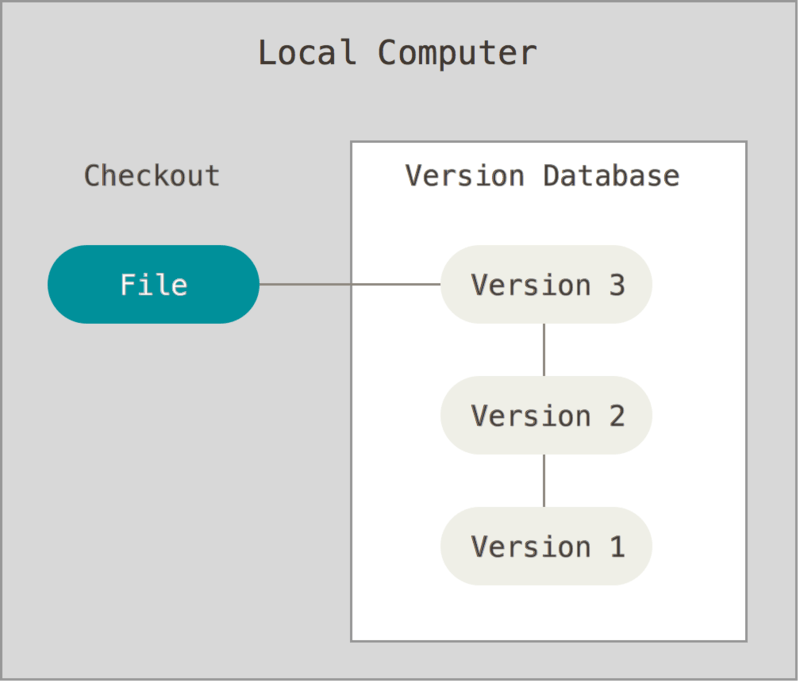
# Git

For our version control system, we looked at various version control systems that we could implement. We decided on Git, a distributed version control system.

# Local Version Control Systems

This is the simplest approach to version control, as it may be achieved copying files into a (time-stamped) sub directory. It is a common approach yet is fails to suit our needs as it is very error prone and does not support concurrent development well. Overwriting files and lack of concurrent development support may lead to loss of data.

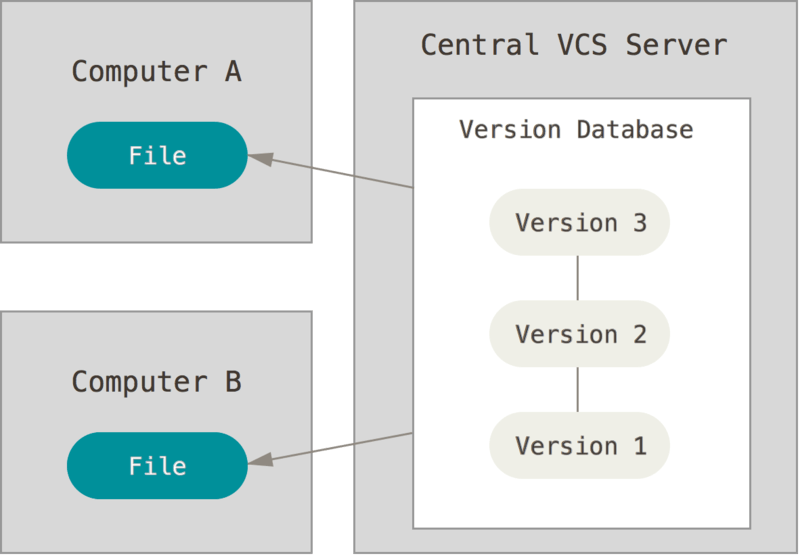
Developed Local VCSs with a simple database such as RCS would neither suit our needs as there still would be no support for concurrent development.



# Centralised Version Control Systems

The next step to consider is a centralised Version Control System (CVCS) as it supports collaboration with other developers. These systems, such as Subversion or Perforce, utilise a single server that contains all versions of the files. It is far easier to administer a CVCS than it is to keep track of local databases on every client.

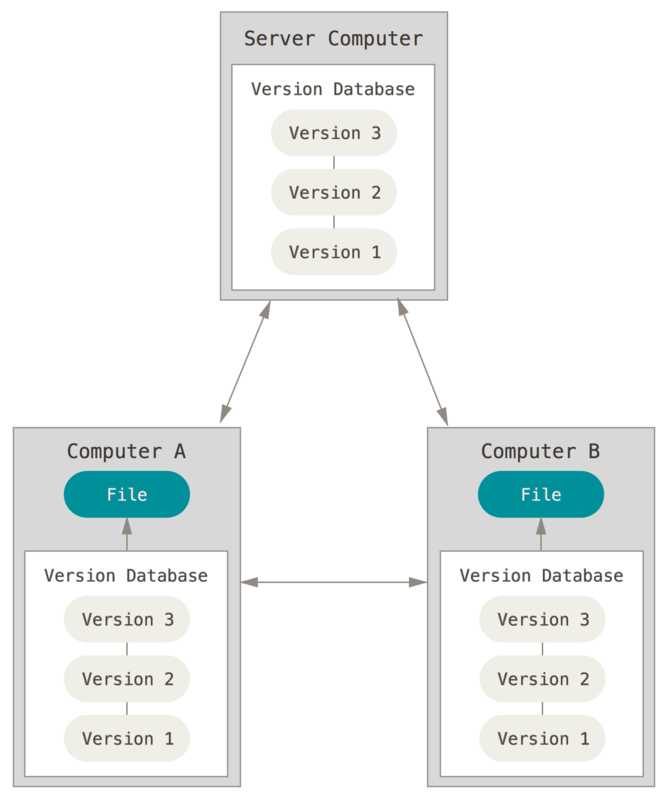
However, the downside comes as there is a single point of failure with a centralised server. If the server goes down, nobody can pull any data or make any changes to the files. In the case of hardware failure on the server, data may be lost if sufficient backups haven’t been made.



# Distributed Version Control Systems

This leads on to Distributed Version Control Systems (DVCSs). Pertaining to the reason why we chose a DVCS is that a popular DVCS is Git, in which some group members were already familiar with.

Like a CVCS, a DVCS highly supports collaboration with other developers, even as far as having several remote repositories to collaborate with different groups of people on the one project. But in contrast to a CVCS, when a client wants to make changes to some files, they mirror the full repository containing the version database. Thus if any server dies, it can be restored by any client that was collaborating with it.



We chose to use GitHub to support online project hosting using Git, as group members were familiar with GitHub already and it best suits our collaboration and version control needs.

# JUnit

# Database/Concurrency

# Design and Architecture Recovery

# Architectural diagram

# Class Diagram

# State Chart

# Description of Patterns and Added Value

# Discussion

Numerous design patterns are implemented throughout the project. The client system utilizes a combination of the observer and factory patterns. The UI system is the main example of the factory design pattern. Each UIFactory, excluding the UIFactory itself, creates an instance of the UI which it is dedicated to initializing. For example, the LogInUIFactory initializes a LogInUI with no arguments as it’s the first menu that the client generates. This makes the system easier to expand in the future as it does not lead to searching through code to make necessary changes.  
  
There are two main examples of the observer design pattern. The observer design pattern is a behavioural design pattern which defines a one-to-one dependency between, meaning when an object changes state, all of its dependants are notified and automatically updated. The first example also relates to the UIFactory. Classes such as the MainMenuController declare the abstract class UIFactory globally, and when a button is pressed to open up a new menu, MainMenuController initializes the UIFactory using one of its subclasses, for example, the ChatRoomUI. This allows the client to generate a new frame and update the interface with it. We had the option of swapping the panels of a single frame each time a new UI is initialized, but we wanted to have the option of keeping two frames open.

The second example of the observer design pattern is seen utilized by the GameModel class. As it receives word from the server regarding the result of a game, the GameModel takes its GameReport abstract class that it declared globally and initializes it with either the Win, Draw or Loss subclasses. The report() method is called and the initialized subclass creates a JOptionPane with output corresponding to the result of the match.

# Critique

We used the noun verification technique to generate a series of classes which appeared to be appropriate for our project. However, during implementation, it became clear that some classes were more abstract than we originally imagined, or served a purpose that another class already accomplished. We also realized that we did not consider classes which would prove to be vital to the project’s success.  
  
Choosing to adopt the Agile Software Lifecycle model meant that we had plenty of flexibility when it came to reacting to changes that had to be made to the project. This would prove to be a crucial contribution to the successes of the project.  
  
We also overestimated the importance of particular classes and packages. The Player package ended up only having one class contained in it; the User class, and we dedicated time and resources to expanding it only for it to be unnecessary. We should have been more careful while defining the role and importance of each class throughout the project.  
  
The majority of user interfaces (UI’s) were originally designed in a single class, but were later converted to Model-View-Controller (MVC) during the second iteration. This was mainly due to the team’s inexperience designing UI’s using MVC, and while they were implemented successfully, it was a slow process and the time spent implementing it could have been spent on implementing a new use case if each UI was originally built using MVC. As all team members have gained valuable experience building using MVC through this project, it will be something that can be considered for the future.

Design patterns were also not considered for the project until late in the iteration cycle. Many design patterns were considered but only the observer and factory design patterns were deemed appropriate for the project. Implementing the decorator design pattern required overhauling the infrastructure of the Match and Player class for minimal benefit. The implemented design patterns did help with tidying up the pre-existing code.  
  
There were a number of problems with the original design of the client:

1: The original plan involved a single server that handled all the client’s requests. Early on during implementation, it became apparent that three individual servers were required to handle file-related queries, the game/matchmaking and the chat.

2: Being able to view friend availability would prove to be very time-consuming to implement due to the server architecture change mentioned in 1. It would involve connecting to the server every time the user wished to quit the client and maintaining a separate text file listing all online users.

3: Inviting a user to a game, along with creating a private game, also proved to be very expensive to implement time-wise due to the problems named above. It would require both users to be online, and it would also require an overhaul of the MatchMaker class or a private game server of its own.

4: The same problems in 3 also applied to some of the use cases intended for the chat system, namely chatting with a friend or another particular user.

5: The Player package was required by both the server-side and the client-side, and there was no simple way to remedy this other than including an instance of the package in separate projects, one for the client and one for the server.

# References