CS319 - OBJECT ORIENTED SOFTWARE ENGINEERING
SPRING 2015

Design Report

Ali Kaviş - Burak Kantarcı - İnci Boduroğlu - Uğur Akkurt

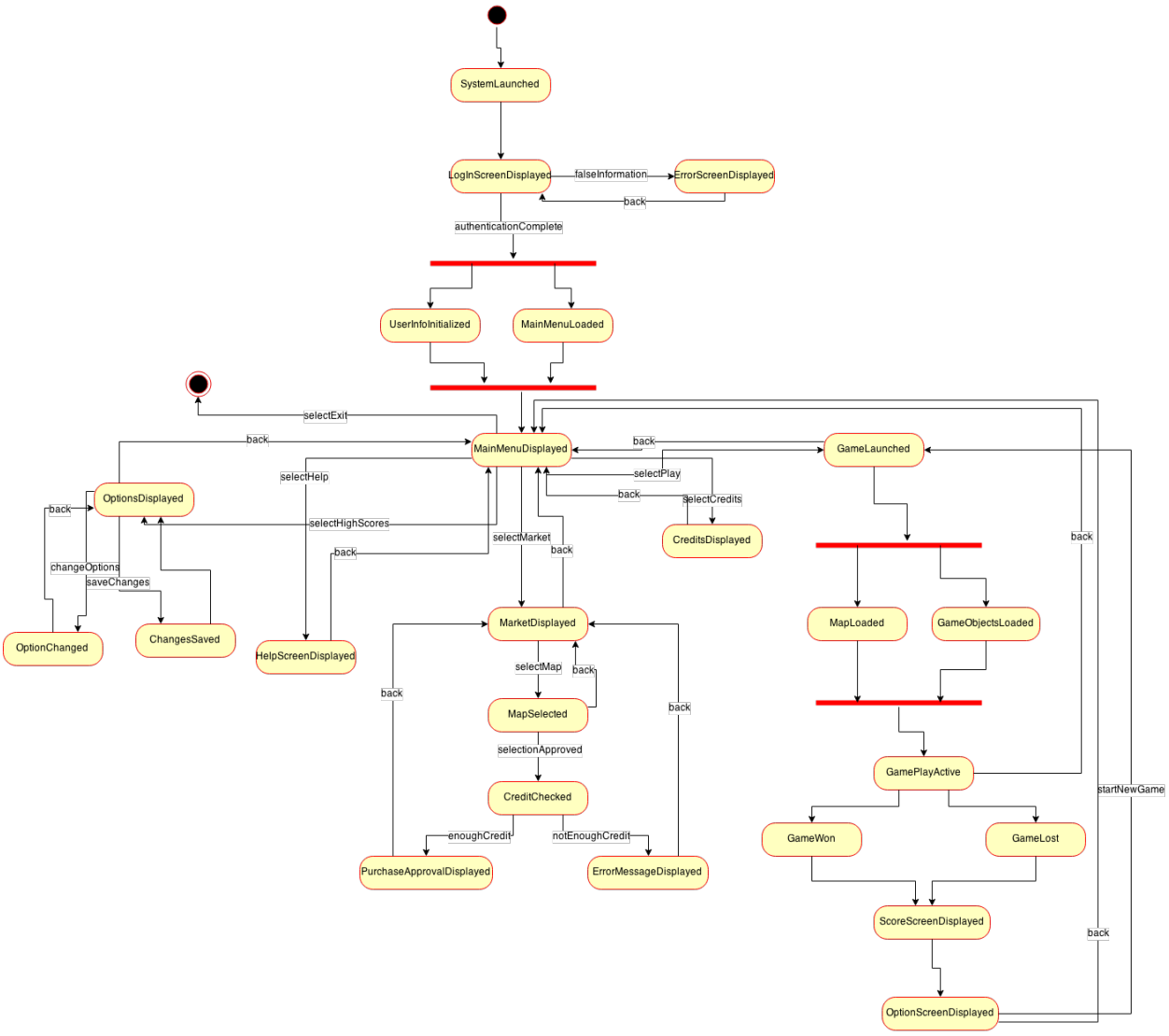
GROUP 4 | SECTION 2

19.04.2015

Table of Contents

Revised Analysis Report

State Diagram for Analysis Report



Conclusion for Analysis Report

In our project, principles of object-oriented programming is applied with its required design patterns to show the interactions between the objects. Showing our project design was the aim of this report by specifying functional - nonfunctional requirements and system design. In overview part, we showed how the game objects will look like. Usability, reliability,performance,

supportability and ease of implementation were our nonfunctional requirements. Scenarios, use case diagrams are stated to show the project's functional behavior. Class diagrams are indicated for the static structure and sequence diagrams and state diagram are used to show the dynamic structure of the game. In user interface part, we showed navigational path in main menu and screen mockups of authentication process, main menu, settings and gameplay(how maps will be seen during the game). All in all, we tried to create a detailed and informative analysis design report since it is important as the base of future expansions of the project when forming design report and implementation of the game.

Design Goals

Dependability

The system will be able to withstand invalid user input. Both while playing the game or at the authentication process. The system will react to the users inputs accurately while playing the game, providing an enjoyable gaming experience. The amount of system crashes is expected to be minimal since it is designed to be highly error protected. Boundary conditions will be considered to evaluate these exceptions. There are no serious risks to the users hardware or software by the system.

Performance

The system will be able to run in a high speed with a low response time. Since it will not be a distributed system which has different controller units in different machines/physical components and not internet enabled via LAN or WAN connections, it will also have a factor increasing the performance of the game, which is designed to run on a single client. In addition, system performs basic and efficient calculations to handle the gameplay. Memory usage of the game will be significantly low because the system does not require huge databases and vast number of objects. Necessary information about user profiles and preferences are saved into a text file, which is easily accessible by the system. There won't be many graphics used in our game which will increase the performance. The frame rate of the game play will be at least 20 fps to allow the graphics to run smoothly.

Cost

The cost of the building and maintaining is non existent since no costly external systems or functional databases are being used and the user can run the system with free software.

Maintenance

The maintenance of the system is trivial because of the high modifiability of it. The system design allows the system to be modified and extended easily. The user interface or graphic changes are trivial because of the data management structure of the system. The system can be run on any platform that runs Java, therefore it is highly portable. Since the programing language is Java and documentation choices, the code will be highly readable. Overall the required maintenance of the system will be minimal when the system is realized after the implementation phase.

End User

The system is very easy for the user to operate. The user interface and the functionality of the system will be easy to understand. While game will use common game I/O components such as mouse and keyboard, every user will adapt themselves with game easily. In addition, game interface will be compatible with common screen resolutions, therefore, there will not be a graphical failure of usage because of any hardware problem. The components of the game are easy to distinguish and since the interface will be highly responsive to the user commands the game play will be very easy to understand.

The learning time for the novice user will be minimal since the system and the concept of the game resembles many Flash based games. There will be a tutorial provided for the game play, but the aim is to make the system so that the novice user will not need any assistance.

Sub-system Decomposition

Our project focuses on designing an arcade game with interactive gameplay. There could be many different approaches to designing an interactive game design; however, the most appropriate object-oriented design pattern is model-view-controller, namely MVC design pattern. MVC constructs the main decomposition of classes into subsystems, which may have internal subsystem separation inside. The system is decomposed into three main subsystems: Model subsystem, View subsystem and Controller subsystem.

Model subsystem encloses entity objects that are designed to maintain and encapsulate the data, which is responsible application domain knowledge. These entity objects are the “models” of real-world entities and they are created/designed after analysis of application

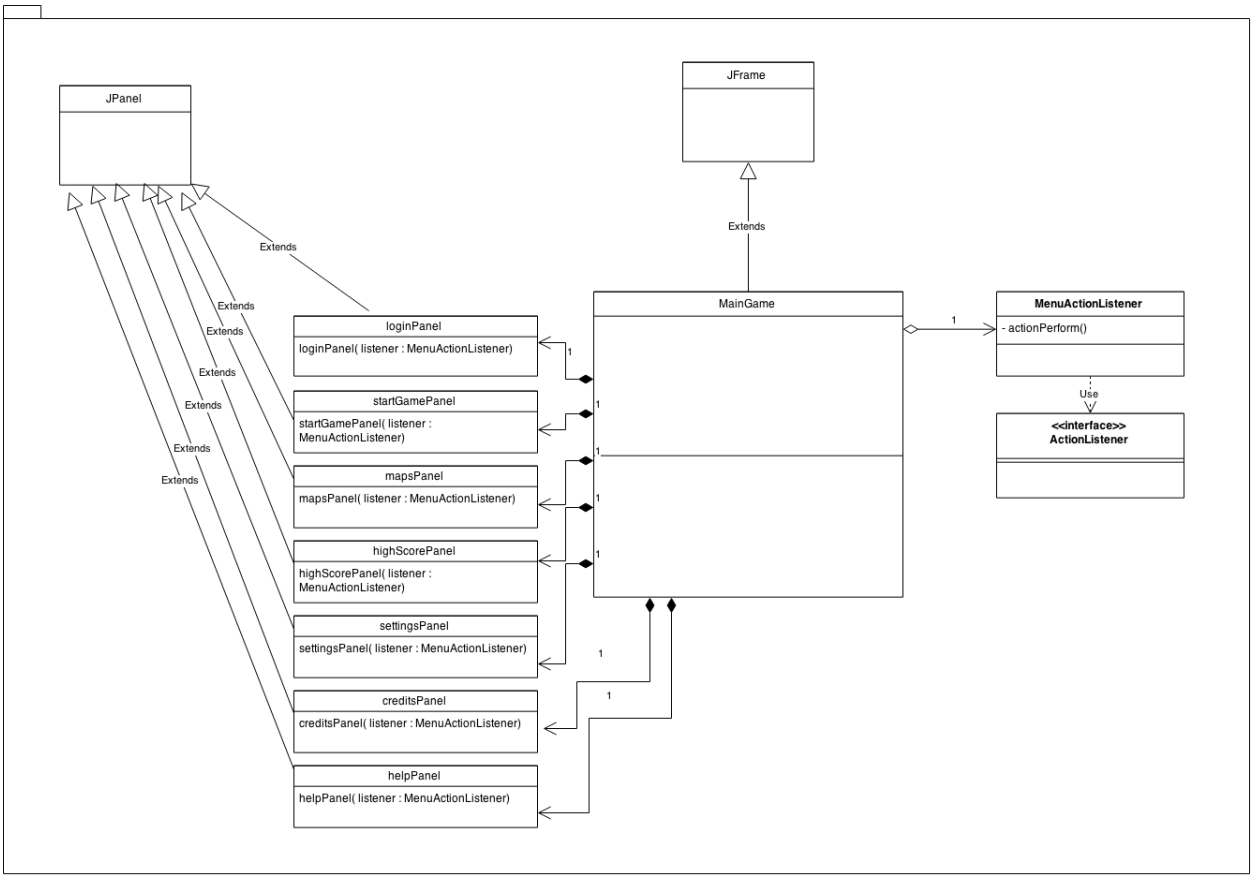
domain, which are the combination of an actual hockey game and brick-breaking game. Model subsystem is responsible for encapsulating the persistent information which is tracked and used by the system regularly. Model subsystem provides information to other, external subsystems via a common interface for its subsystem. Any external class or subsystem should not be able to access the classes directly, but use the provided interfaces to update, modify, obtain information and create necessary entities. Map and Profile classes act as the “interfaces” for model subsystem as a whole so that controller only knows about these two classes to make necessary changes.

View subsystem is basically responsible for rendering the user interface and displaying the relevant model objects with their current states during gameplay. View subsystem mainly consists of panels and frames, which render and present the user interface and detect inputs provided by the user. Since this subsystem is designed to provide the user interface and interaction between the system and users, we can name it as presentation level of the system. View does not access neither business logics nor model objects directly. However, view needs data from model objects constantly to render the graphical user interface, and view subsystem should respond according to the inputs provided by the user. To do so, view needs services from another subsystem, which corresponds to Controller subsystem in our design. View subsystem provides a common interface for the classes it encloses and these classes are provided with necessary information about entity objects via this interface.

Controller subsystem is the intermediate connection between view and model subsystems. What controller does is basically handling inputs feeded from view, updating relevant model objects according to both refresh rate and inputs, providing updated data from models to view and forming an intermediary layer between view and model. Controller has its own interface to provide data encapsulation. Controller both serves to model and view subsystems via this common interface.

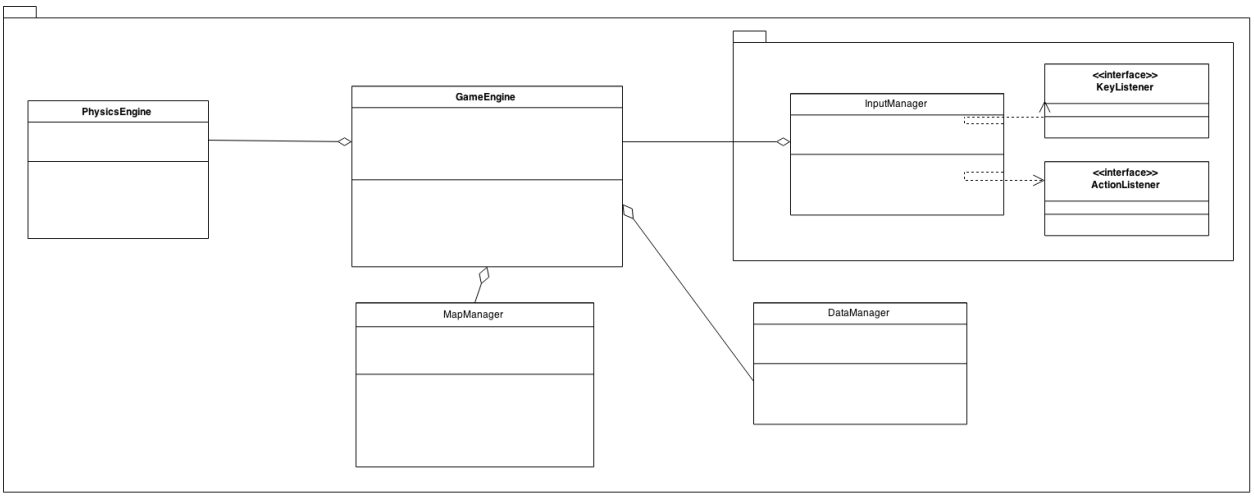
View

View subsystems does not have an internal subsystem decomposition. It consists of JPanel and JFrame classes:



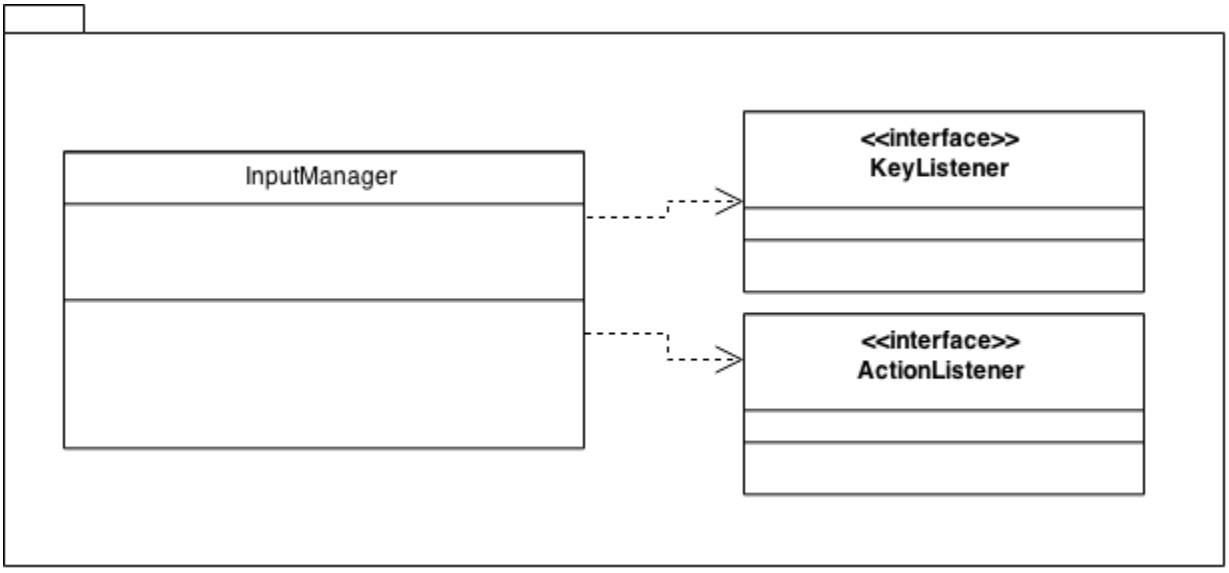
Controller

Controller subsystem consists of manager classes which handle different processes such as collision handling or data management of saved data files etc:



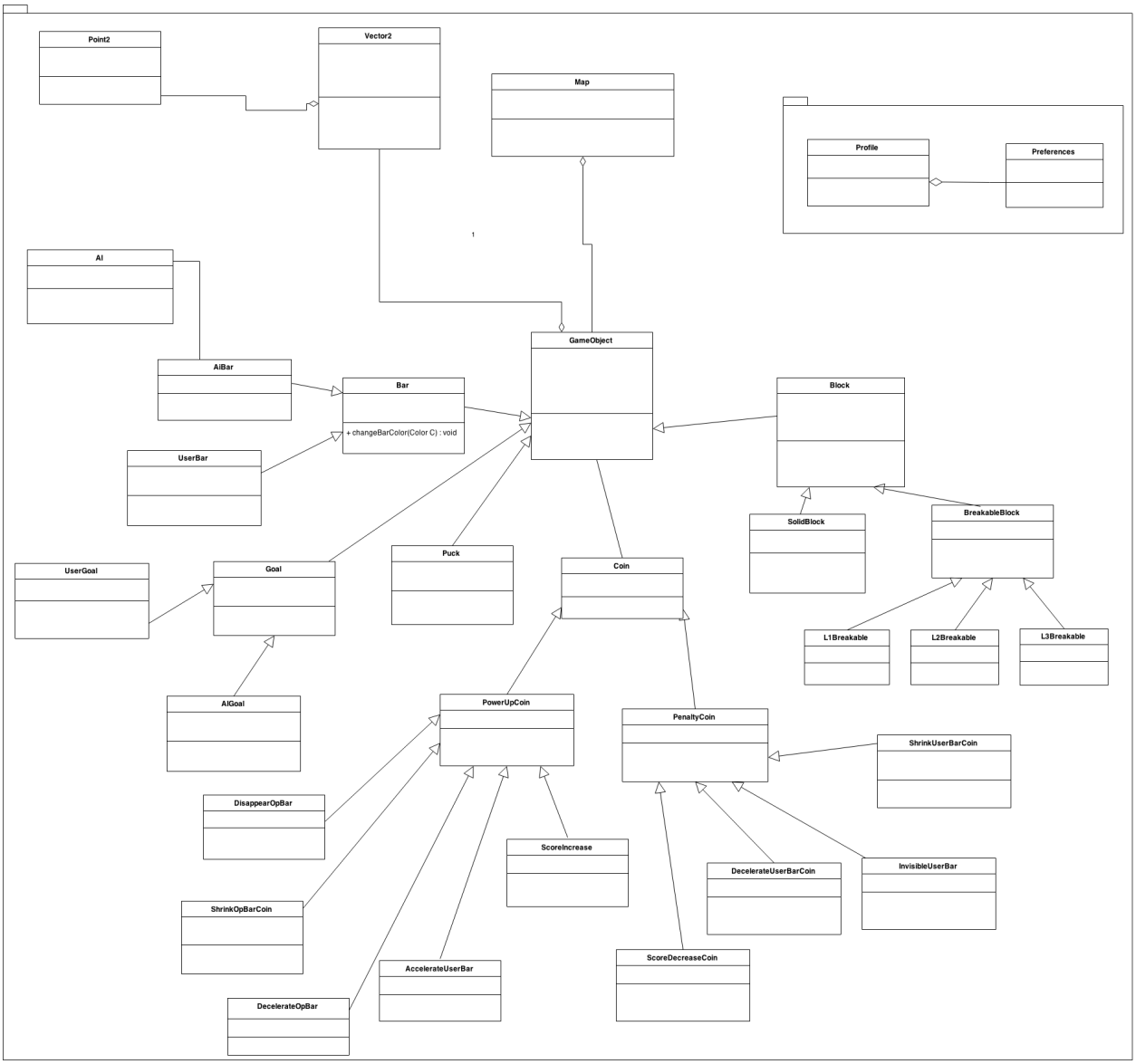
InputManager

InputManager listens to the inputs from buttons and keyboard and controls the flow of events in the game loop accordingly. It consists of InputManager class with two listener classes:



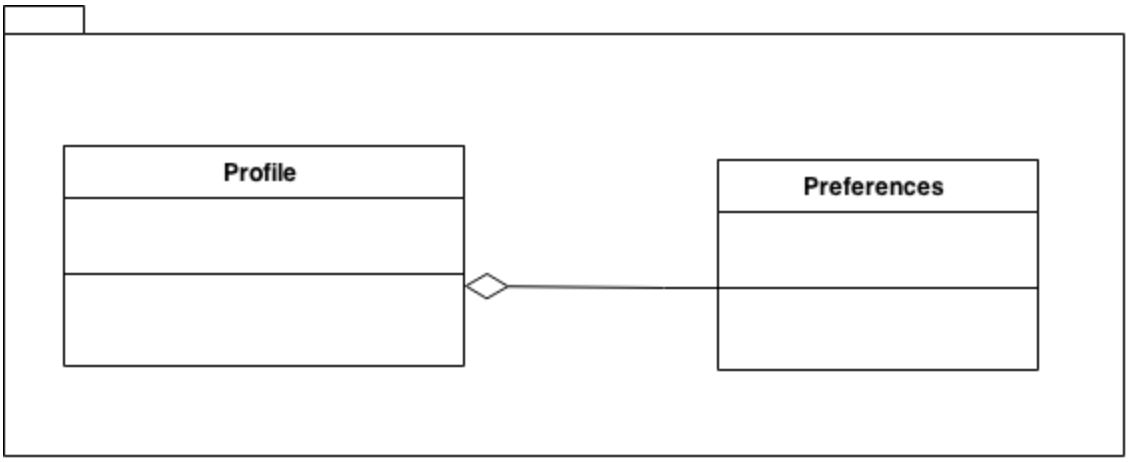
Model

Model subsystems is composed of Map class, the class representing the subsystem interface, GameObjects and its inherited children and the Data subsystem that handles the user profile information with authentication information and profile preferences and so on:



Data Subsystem

Data subsystem consists of the profile of each user and the respective preferences for the profile. This subsystem is created from flat text files and stored as part of model subsystem:



Architectural Patterns

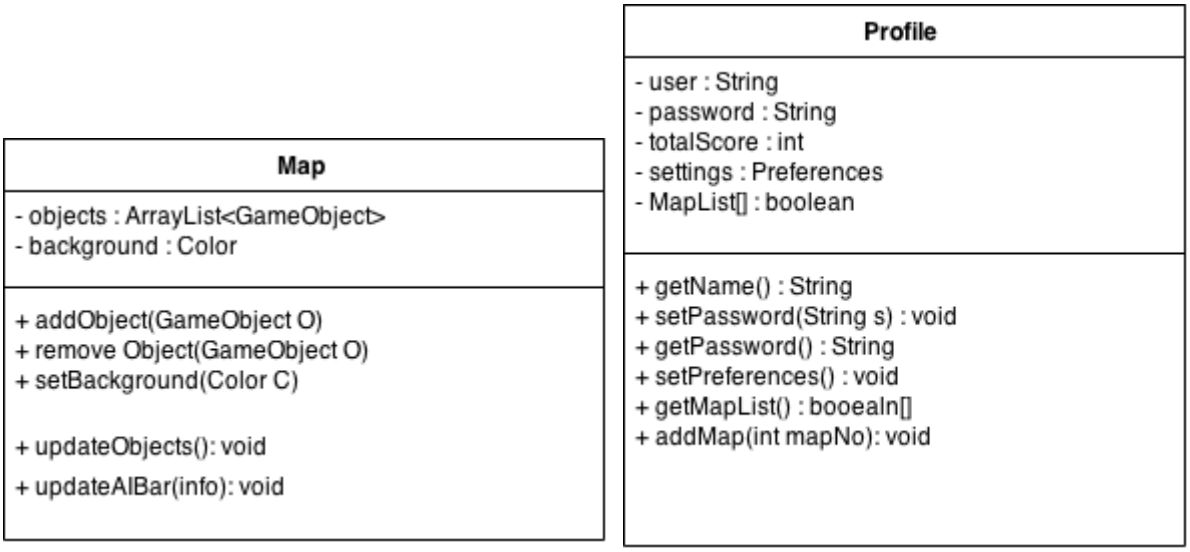
Facade Pattern

Facade pattern is a structural pattern that helps create a simple and more consistent relationship between subsystems. Facade pattern provides a common, easy to use interface for each subsystem with the purpose of providing two main benefits: it encapsulates the information of classes in one subsystem from another one by enabling the other subsystems to modify information in one subsystem to the extent they are allowed by the interface. In our design, we have GameEngine class as the Facade for Controller subsystem. It is not a true interface, but view only uses the functionalities in GameEngine to access necessary information from model objects and render the user interface accordingly. Game Engine Class, which hold references to all controllers of the system, is given below:

GameEngine
+ DataManager: DataManager + MapManager: MapManager + inputManager: InputManager + PhysicsEngineManager: Physics Engine + AIController : AI
+ initialize(): void + run(): void + pause() : void + update(): void + repaint(): void + exit(): void

For Model subsystem, Map and Profile classes are two “pseudo” interfaces that provide limited access for entity objects to Controller subsystem. In the implementation part of the project, these classes may be modified and designed in a different way so they can represent

true interfaces for their subsystem. However, with the current design, we are decreasing the coupling between subsystems via these three classes which represent the interface for their subsystems. Map and Profile classes, which hold references to GameObjects and userPreferences, are shown below:



Model-View-Controller Architectural Style

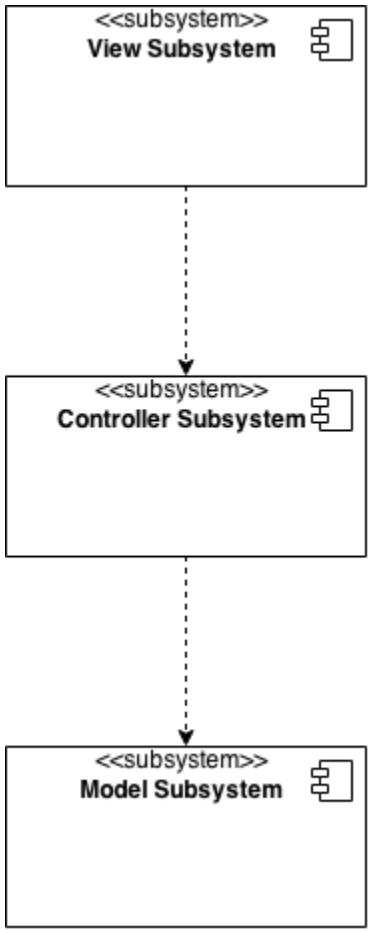
One of the architectural styles and design patterns that we are going to use to design the system will be Model-View-Controller (MVC). We choose MVC as one of the architectural pattern that we will use in our game design because MVC provides low coupling and high coherence among subsystems. For instance, user interface, business logic and entity objects (or data objects) should not be interdependent, otherwise even a minor change in any of these three will result in huge impact on rest of the system. In order to fix the impacts of even tiny changes in highly coupled systems, the cost to be paid is so vast that it may even require to set the relations between classes from the beginning. With the purpose of enabling changes to the system with little or no The subsystems are classified into three categories as Models, Views and Controllers, which is explained in details in the previous section of the report. The model subsystem does not depend on controller or view subsystems: it is independent of other subsystems. Model subsystem's responsibility is providing states of and information about the entity objects and accepting modification commands from other subsystems via its interface.

Controller depends on model but not on view subsystem. The dependency of the controller subsystem on model is at the level of objects' states and their behaviors for processing inputs and updating the entities accordingly. Controller can access the data from

model subsystem via model's interface and notify model objects about necessary updates on them as a result of received inputs, interaction between objects and system's regular refresh.

View, as the last main subsystem, has a dependency on controller subsystem. View is not involved in any part of business logic, collision handling, model updating and so on. Thus, view needs the functionalities of controller subsystem to pass the inputs received from the user and let controller handle the necessary calculations for proper updating of the model. Additionally, view requires data from entity objects to render the user interface and gameplay accordingly. Similarly, view depends on controller on this task as well. Controller is requested to provide the entity objects' properties and information with the latest states.

Model, view and controller subsystems are shown below with their runtime dependencies:



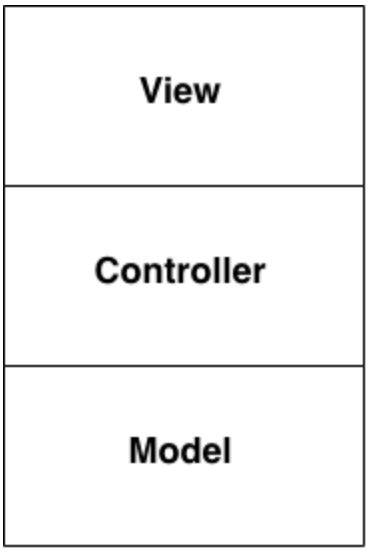
3-Layer-Architectural Style

To construct a solid relationship between our main subsystems, we can put these subsystems in three layers, as we discussed in the previous section. View depends on the controller subsystem and controller depends on model subsystem. View accesses controllers

functionalities via an interface without being able to make modifications on Controller subsystem's internals. Likewise, controller needs to know information related to states of model subsystem's objects, to the extent that the model interface enables it. As a result, this architecture of subsystems is an example of closed architecture, in other words opaque layering. As the name implies, in this architecture, each layer (alternatively each virtual machine) only knows about the layer immediately below themselves. For instance, controller knows about the model but not about the view, and view knows about the controller but not model directly. This architectural design puts an extra layer between the user interface and the entity objects that are visualized by the view, but this separation of subsystems in such a manner comes with a significant benefit of maintenance. As we discussed above, use of Façade design pattern for each subsystem perfectly fits with the opaque layering. None of the layers are interested in the internal implementations or relations between classes in other subsystems. They require to get the services they need by using the common interfaces provided by other subsystems. Thus, we also prevent subsystems to directly access the internals of other subsystems.

This structure of our subsystems has a counterpart and equivalent approach in software design architectures, which is 3-layer architectural style. 3-layer architectural style consists of 3 hierarchically related subsystems: the upper level is called presentation layer or user-interface, middle layer is called application layer and the bottom layer is called data layer. In our design, View subsystem corresponds to presentation layer, Controller subsystem corresponds to application layer in which the business logic is implemented and the data layer is equivalent to Model subsystem where the persistent data is managed.

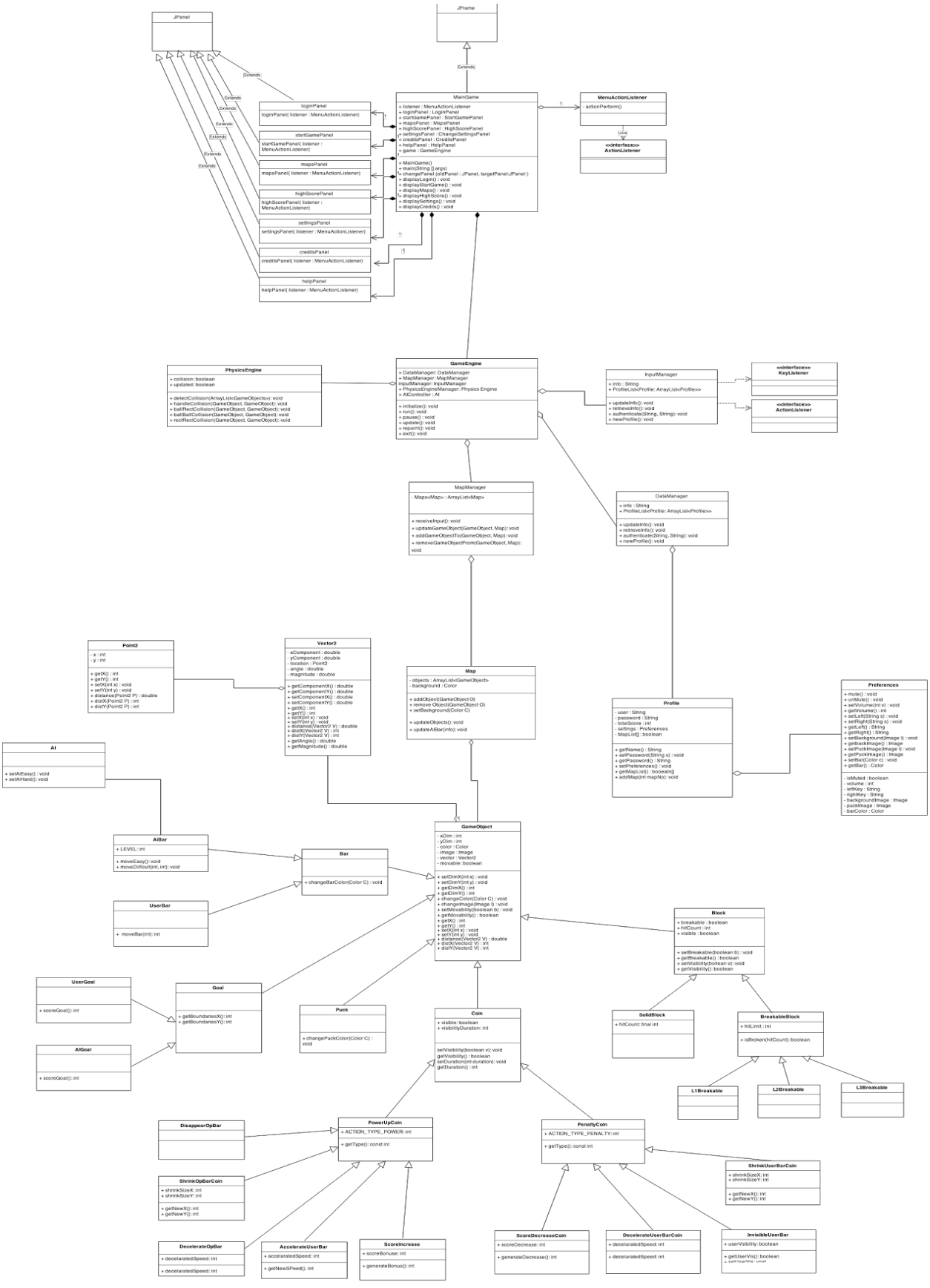
3-layer architecture is given below as a simple diagram. Every layer can only interact with layer immediately below them:



Since our system is highly interactive and has multiple views that needs to interact with controller and states of the model objects, MVC controller architectural style represented in 3-layer architectural style foster a highly coherent, consistent design with a remarkably low coupling between distinct subsystems.

Class Diagram

After applying different design patterns and architectural styles, our class diagram is constructed as a whole as shown in the following figure. This diagram does not specify any subsystems to give an overall view of the class relations, taxonomies and dependencies, while providing attributes and methods for each class in the system:


JFrame
JPanel
Extends
MainGame
+ listener : MenuActionListener
+ loginPanel : LoginPanel
+ startGamePanel : StartGamePanel
+ mapsPanel : MapsPanel
+ highScorePanel : HighScorePanel
+ settingsPanel : ChangeSettingsPanel
+ creditsPanel : CreditsPanel
+ helpPanel : HelpPanel
+ game : GameEngine
+ MainGame()
+ main(String [] args)
+ changePanel (oldPanel : JPanel, targetPanel:JPanel)
+ displayLogin() : void
+ displayStartGame() : void
+ displayMaps() : void
+ displayHighScore() : void
+ displaySettings() : void
+ displayCredits() : void
MenuActionListener
- actionPerform()
Uses
<<interface>>
ActionListener
loginPanel
loginPanel(listener : MenuActionListener)
startGamePanel
startGamePanel(listener : MenuActionListener)
mapsPanel
mapsPanel(listener : MenuActionListener)
highScorePanel
highScorePanel(listener : MenuActionListener)
settingsPanel
settingsPanel(listener : MenuActionListener)
creditsPanel
creditsPanel(listener : MenuActionListener)
helpPanel
helpPanel(listener : MenuActionListener)
PhysicsEngine
+ collision: boolean
+ updated: boolean
+ detectCollision(ArrayList<GameObjects>): void
+ handleCollision(GameObject, GameObject): void
+ ballRectCollision(GameObject, GameObject): void
+ ballBallCollision(GameObject, GameObject): void
+ rectRectCollision(GameObject, GameObject): void
GameEngine
+ DataManager: DataManager
+ MapManager: MapManager
inputManager: InputManager
+ PhysicsEngineManager: Physics Engine
+ AIController : AI
+ initialize(): void
+ run(): void
+ pause() : void
+ update(): void
+ repaint(): void
+ exit(): void
InputManager
+ info : String
+ ProfileList<Profile: ArrayList<Profile>>
+ updateInfo(): void
+ retrieveInfo(): void
+ authenticate(String, String): void
+ newProfile(): void
<<interface>>
KeyListener
<<interface>>
ActionListener
MapManager
- Maps<Map> : ArrayList<Map>
+ receiveInput(): void
+ updateGameObject(GameObject, Map): void
+ addGameObjectToGameObject, Map): void
+ removeGameObjectFromGameObject, Map): void
DataManager
+ info : String
+ ProfileList<Profile: ArrayList<Profile>>
+ updateInfo(): void
+ retrieveInfo(): void
+ authenticate(String, String): void
+ newProfile(): void
Point2
- x : int
- y : int
+ getX() : int
+ getY() : int
+ setX(int x) : void
+ setY(int y) : void
+ distance(Point2 P) : double
+ distX(Point2 P) : int
+ distY(Point2 P) : int
Vector2
- xComponent : double
- yComponent : double
- location : Point2
- angle : double
- magnitude : double
+ getComponentX() : double
+ getComponentY() : double
+ setComponentX() : double
+ setComponentY() : double
+ getX() : int
+ getY() : int
+ setX(int x) : void
+ setY(int y) : void
+ distance(Vector2 V) : double
+ distX(Vector2 V) : int
+ distY(Vector2 V) : int
+ getAngle() : double
+ getMagnitude() : double
Map
- objects : ArrayList<GameObject>
- background : Color
+ addObject(GameObject O)
+ remove Object(GameObject O)
+ setBackground(Color C)
+ updateObjects(): void
+ updateAIBar(info): void
Profile
- user : String
- password : String
- totalScore : int
- settings : Preferences
- MapsList[] : boolean
+ getName() : String
+ setPassword(String s) : void
+ getPassword() : String
+ setPreferences() : void
+ getMapList() : boolean[]
+ addMap(int mapNo): void
Preferences
+ mute() : void
+ unMute() : void
+ setVolume(int v) : void
+ getVolume() : int
+ setLeft(String s) : void
+ setRight(String s) : void
+ getLeft() : String
+ getRight() : String
+ setBackground(Image I) : void
+ getBackImage() : Image
+ setPuckImage(Image I) : void
+ getPuckImage() : Image
+ setBar(Color c) : void
+ getBar() : Color
- isMuted : boolean
- volume : int
- leftKey : String
- rightKey : String
- backgroundImage : Image
- puckImage : Image
- barColor : Color
AI
+ setAIEasy() : void
+ setAIHard(): void
AiBar
+ LEVEL: int
+ moveEasy(): void
+ moveDifficult(int, int): void
Bar
+ changeBarColor(Color C) : void
UserBar
+ moveBar(int): int
GameObject
- xDim : int
- yDim : int
- color : Color
- image : Image
- vector : Vector2
- movable: boolean
+ setDimX(int x) : void
+ setDimY(int y) : void
+ getDimX() : int
+ getDimY() : int
+ changeColor(Color C) : void
+ changeImage(Image I) : void
+ setMovability(boolean b) : void
+ getMovability() : boolean
+ getX() : int
+ getY() : int
+ setX(int x) : void
+ setY(int y) : void
+ distance(Vector2 V) : double
+ distX(Vector2 V) : int
+ distY(Vector2 V) : int
Block
+ breakable : boolean
+ hitCount : int
+ visible : boolean
+ setBreakable(boolean b) : void
+ getBreakable() : boolean
+ setVisibility(boolean v): void
+ getVisibility(): boolean
SolidBlock
+ hitCount: final int
BreakableBlock
+ hitLimit : int
+ isBroken(hitCount): boolean
L1Breakable
L2Breakable
L3Breakable
UserGoal
+ scoreGoal(): int
AiGoal
+ scoreGoal(): int
Goal
+ getBoundariesX(): int
+ getBoundariesY(): int
Puck
+ changePuckColor(Color C) : void
Coin
+ visible: boolean
+ visibilityDuration: int
setVisibility(boolean v): void
getVisibility() : boolean
setDuration(int duration): void
getDuration() : int
DisappearOpBar
PowerUpCoin
+ ACTION_TYPE_POWER: int
+ getType(): const int
PenaltyCoin
+ ACTION_TYPE_PENALTY: int
+ getType(): const int
ShrinkUserBarCoin
+ shrinkSizeX: int
+ shrinkSizeY: int
+ getNewX(): int
+ getNewY(): int
ShrinkOpBarCoin
+ shrinkSizeX: int
+ shrinkSizeY: int
+ getNewX(): int
+ getNewY(): int
DecelerateOpBar
+ decelaratedSpeed: int
+ decelaratedSpeed: int
AccelerateUserBar
+ accelaratedSpeed: int
+ getNewSPeed(), int
ScoreIncrease
+ scoreBonuse: int
+ generateBonus(): int
ScoreDecreaseCoin
+ scoreDecrease: int
+ generateDecrease(): int
DecelerateUserBarCoin
+ decelaratedSpeed: int
+ decelaratedSpeed: int
InvisibleUserBar
+ userVisibility: boolean
+ getUserVis(): boolean

Hardware/Software Mapping

The game demands several computations of the AI and user's behaviors. These computations are easy to calculate. For instance, the position of the bar of the AI will be calculated according to the puck's position. Therefore, computation rate is not too much for the processor. Single processor will be sufficient for the game. Similarly, the game will not require too much memory space. The game will require Java Runtime Environment to be executed. The game can run on any computer with any operating system that can run Java. Mouse and keyboard are the required hardware devices to get inputs from the user. Mouse is used for clicking to the buttons and the keyboard is used for providing login information and directing the bar during the game. So Hardware requirements consist of a keyboard and a mouse. There is no need for an internet connection.

Input manager which is a part of controller subsystem, receives input from these hardware devices and makes necessary calculations.After applying respective changes on the entity objects, view subsystem is notified about the changes in the entities which should be updated on output devices.

Addressing Key Concerns

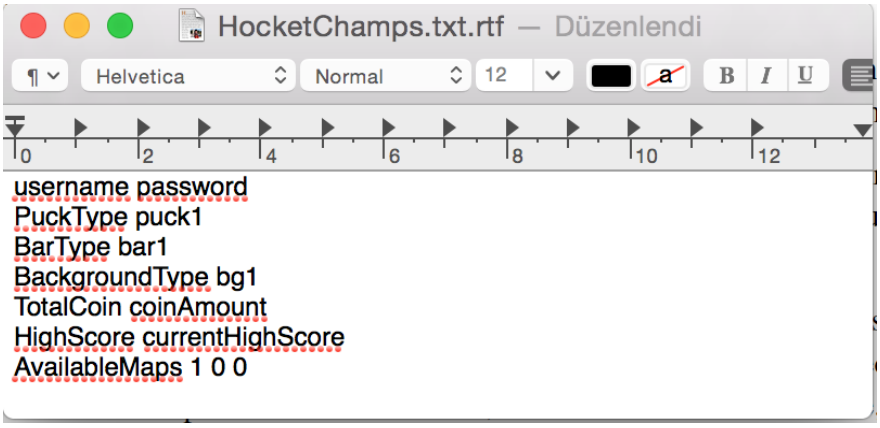
Persistent Data Management

Game data, such as user authentication information, profile preferences of each user and users' scores and unlocked maps will be stored via plain texts instead of databases. We chose flat files to store data because these data will be less complex and not have multiple writers to save things on these files. With a database we would have to write bunch of administrative tools to handle things, which will not provide an additional ease to handle data in our data storage where the data rather simple. Additionally, there will be only one modifier of the data file at a time who is the current user playing the game, and reading/writing data will not be done much other than logging in, setting new preferences for the profile, and adding the score of a game after finishing it, which is why plain texts are enough to keep and access data easily.

There will be no problem about concurrency when modifying the data files. Since there is only one user that is accessing the saved data file either to read or write, there will be no

concurrency in reading the data as well as writing to it. What's more, only one application will be accessing to these data files because our system does not allow multiple users to interact with the system and only one user who logged in to his/her profile should be able to use the functionalities of the system. Therefore,

Sample text file for keeping data is following:



Access Control and Security

Since there is only one actor that can use the system, the users can reach every feature of the game that is available to the user. For access to the core functions of the game, the user has to authenticate their information at the login screen of the game. There are no additional systems used therefore there are no security restrictions other than user authentication. Below shows a simple in game access control mechanism for game players:

	UserBar	AIBar	
Player	moveBar()		
AI		moveEasy() moveDifficult()	

Main key concern about the security is that the data management system does not have any permanent security. Data inside the flat files can be easily changed by accessing the directory of the program. However, user is not allowed to access the system data files while executing the game. The data file is only accessible by the system itself and if user makes a

change on his profile by purchasing a map, winning a game, or setting new preferences for the gameplay then the changes are made by the system in a secure manner. Our system does not provide security at the level of file encryption, but users are not allowed to make any change on the internals of the system including classes and flat text files via the program itself.

Global Software Control

The design decision is to use explicit event-driven control since the system needs to be able to react to the user's input instantaneously and tackle complicated algorithm computations at the same time, while embed procedure-driven control mechanism by providing an update mechanism which is controlled in a regularly fashion by the main loop of the game, which we can call the game loop.

Event-driven control flow is suitable for object oriented design by the way it allows the system to be highly reactive to the status of the objects in the system. By using this control structure, the input will be centralized and the controller structure of the system will be simpler. Thus, whenever a new input is received, system will adjust its state to react to the input and update itself accordingly.

Procedure-driven control should be necessarily embedded into program code, because our game is designed in such a way that if there is no user interaction to the system during gameplay, system should refresh itself and update model objects regularly to maintain the course of the game. This control flow reside in the update() and run() function of the GameEngine class, which will be implemented so that system regularly updates itself even if no input is received.

A centralized system comes with a bottleneck which the presence of a single control unit as we do in our design, however, system is separated into 3 layers, which interact with each other in a maintainable way, and concurrency is not a problem for our system, thus, we will not be experiencing the performance reduction due to a single controller unit.

Boundary Conditions

Within the initialization process, only main menu needs to be initialized by accessing their panels within the user interface. There will be no user logged in and so regarded maps and scores will not be loaded which will provide a fast execution of initialization. The user should enter his/her username and password that he/she created by signing up, to load his/her current score and unlocked maps which is done after the initialization process.

After the login process, the maps that user unlocked and his/her current score will be loaded to the system reading flat text. During the game, after user opens a map and plays a game with the computer, the score he/she gained from the match will be saved to the flat text automatically. The user can keep playing more games in different maps, and scores will be saved accordingly after each game without waiting for the termination of the program.

The system creates it's core controllers and views on every startup and destroys them at every shut down. The data is saved to the flat files before every shutdown, and the player data is loaded after every authentication. As a result of persistency of user data, when the game is closed or shut-down, no information is lost and saved into data files.

Startup : Click on the executable file on the computer
Shutdown : Click on the "close" icon on the gameplay screen or default window closer

Configuration: Configuration conditions are needed mostly in the game play. The system will check the status of the objects that are expected to be created before the start of the game. Since there is no external systems interfering with the system, configurations only apply to the mapped objects.

Exception handling: Hardware failures are not expected to have hardware failure in the game since there will be not much memory needed in the game to store data. Only plain texts will be used to store data and so it is not expected for storing operations to cause any faults in the hardware.

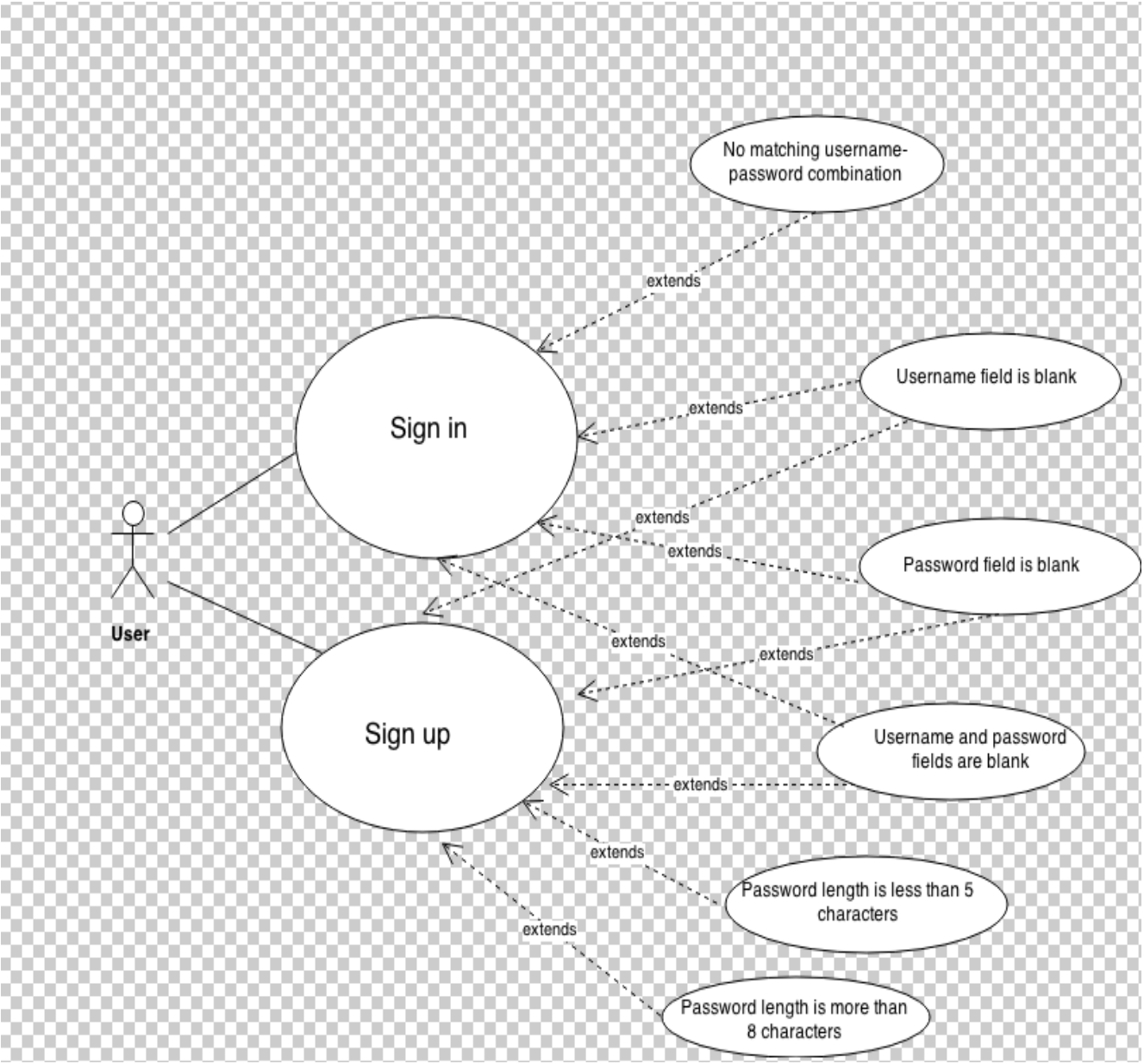
Software failure: Exceptions will be handled by warning the user if it is something wrong about the choice of user. Error conditions are as following:

-
- Logging in :
    - Username or password field may send as blank
    - Username has invalid characters such as ? , ! . - etc.
    - Password length is less than 5 and higher than 8

        - There is not matching username/password configuration
    - User settings :
        - User’s preferences unable to change settings of the gameplay (background, puck, bar)
        - While editing the settings system crashes
    - Gameplay
        - System does not recognize user preferences
        - System does not recognize the scores
        - AI does not react to the user’s behaviors

If user does anything during the sign in or sign up processes that does not fit with the requirements of valid username or password, the program will show a dialog to warn user.

Below is given a use case diagram for the boundary cases during sign in and sign up:



Conclusion

In our project design report, we aimed to use structures that fit with the object oriented design. Dependability, performance, cost, maintenance and end user are our design criterias. MVC pattern will be used for the object design. Model subsystem, View subsystem and Controller subsystem are the main subsystems in our design.Facade pattern and three tier architectural design styles will be used within the MVC pattern. Facade pattern will be used to be able to have a simple and consistent relationship between the subsystems by encapsulating

them. These subsystems will be in three tier architectural style when showing the relationships between them. Keyboard and mouse will be the hardware components in our program which will be controlled within the input manager. Initialization process of the program will be fast since no loadings of maps are needed without the signing in process. Software failures (exceptions) will be handled by showing dialog windows to the user to warn him/her. All in all, we aimed to show the general system design in detail complying with the object oriented design approaches.