Bilkent University

Department of Computer Engineering

CS 319 - Object-Oriented Software Engineering

Term Project - Design Report

Iteration 1

Project Name:	Walls & Warriors
Group No:	2B
Group Name:	OOPs
Group Members:	Ali Babayev Tunar Mahmudov Merve Sağyatanlar Çağla Sözen Emin Bahadır Tülüce

Table of Contents

1. Introduction

1.1. Purpose of the system

Walls & Warriors is a desktop game for solving puzzles. Puzzles in Walls & Warriors aim to entertain users while also training problem solving skills. Desktop version will implemented in a manner that will provide a simple to play, enjoyable and mind challenging game experience. Fundamentals of the game do not require a dynamic nature, however it does require user interactions. The distinguishable feature of this version of Walls & Warriors will be that players will be able to create their own shareable challenges, which are encoded to Base64 that will allow these custom challenges to be shareable. In essence, the intention while implementing Walls & Warriors will be to create a user-friendly, reflexive, fast and entertaining game.

1.2. Design goals

The primary design goal is to have classes that meet requirement demands correctly and use object-oriented design principles in a way such that it will comply with the original game design. Maintainability is one of the primary design goals to keep the implementation design elegant, easy to understand and enduring to change even as the complexity of the implementation increases. Extensibility is another design goal for Walls & Warriors in order to provide easy and synchronized addition of new features to the game. Having these goals in mind, this implementation will aim to provide a maintainable, extendible and reusable design.

2. High-level software architecture

2.1. Subsystem decomposition

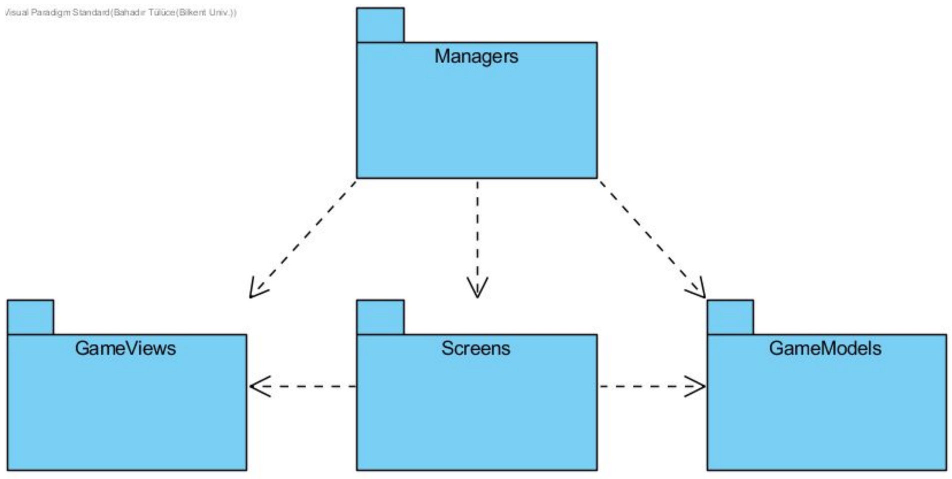

Figure 1: Subsystem dependencies

We have decided to decompose our system into 4 parts. These parts and their dependencies are given in *Figure 1*. Explanation for each subsystem is given in the subsections below.

2.1.1. Managers

This is the main subsystem for managing the game state and resources. Managers deal with the management and execution of file operations, data flows, screen transitions, and the algorithms that builds the main game logic.

2.1.2. Screens

Screen objects are the highest-level components for displaying the game state and menu interfaces to the user. The Screens subsystem encapsulates all of the screen objects and their logic. Screens are also responsible for processing image-based inputs (mouse click on a particular object). However, the Screens do not include any real logic about the game or any rendering method about the game components. Thus, it depends on GameViews and GameModels subsystems.

2.1.3. GameViews

This subsystem is used for rendering the game components to the screen properly. The current game state is displayed to the user through the elements of this subsystem. This subsystem is controlled by Screens subsystem.

2.1.4. GameModels

This subsystem is for storing the data about to the game and its current state. It holds data related to challenges in an abstract manner. This subsystem does not depend on any other subsystem since it is the core element of the game logic.

2.2. Hardware/software mapping

Our game will be completely implemented on software and it will be mapped to hardware through some standard libraries. We will use Java which will make our implementation easier, because of the following reasons: It promises platform-independent applications and provides powerful object-oriented tools such as inheritance and polymorphism (See *Section 4.3* for applied design patterns).

- For screen display; we will use a JavaFX stage, which handles both GPU rendering and platform-independent frame creation.
- For sound outputs; the `javax.sound` package provides us an appropriate interface.
- For mouse inputs; we will make use of the mouse listeners in JavaFX.

2.3. Persistent data management
Our game will need access to persistent data storage of the system to keep some player-related information. This data is planned to be stored under the home folder of the user. A hidden folder named `.wnwdata` will be created under the home and three different files. The necessary information to store and their storage file is listed below respectively.

- The data of campaign challenges in `/campaign_challenges.dat`
- The data of imported custom challenges in `/custom_challenges.dat`
- The progress of campaign challenges in `/player_progress.dat`
- The solved/unsolved flags of custom challenges in `/player_progress.dat`
- The settings information of the last session in `/settings.dat`

2.4. Access control and security
As mentioned in *Section 2.3* we will not use any external databases, we will only use the internal storage so we will not implement access control. Also our system will not support network connection so that it will be impossible to be exposed by malicious softwares.

2.5. Boundary conditions

2.5.1. Application setup
Walls & Warriors will not require installation of third party extensions or any setup for itself. During start up, files containing challenge data and user progress data will be read from `.wnwdata` hidden folder. The game will start up with the main menu screen that provides routing within the game.

2.5.2. Terminating the application
Walls & Warriors can be terminated by clicking the exit button of the game window. This feature will allow the user to terminate the game regardless of which screen the user is currently using.

2.5.3. Input/output exceptions
The possible failures of the game may be caused by several cases. During startup, if there occurs a file reading problem, which may be caused by an exception or an invalid challenge, the game may have issues loading Campaign Challenges Screen and Custom Challenges Screen properly. The second case may be resolved by detecting and deleting the problematic challenge data. Also, Challenge export process may fail due to a access permission issue. It may be resolved by changing the access permission settings for the regarding folder through giving necessary instructions to the user.

There will be some input fields in the challenge editor. The user might enter too long or invalid string to these fields. Also, user may left the input fields empty. Either case can be resolved by popping up warning messages and waiting for valid inputs.

2.5.4. Critical errors

In any case of an application collapse, the game will recover in the same manner with a regular start up. However, in such case, user progress data will not be saved.

3. Subsystem services

3.1. User Interface Layer

3.1.1. Basic GUI Layer

This layer will contain the basic components provided by JavaFX library such as buttons, text fields and scrollable lists. They will make the implementation easier and also will increase the reliability of the program since JavaFX standard library components are stable and reliable tools used by many other JavaFX programs.

3.1.2. Advanced GUI Layer

This layer will contain the game components such as grid, walls and knights. They will be rendered through a `GraphicsContext` object of a `Canvas` component. We have decided to implement and render them ourselves, since basic GUI components won’t be sufficient for displaying good-looking game elements.

3.1.3. Mouse Input Layer

This layer will be embedded within the other GUI layers since the mouse clicks are directed to the GUI components (for both basic and advanced). Clicks to basic GUI layer will be handled through the mouse listeners provided by JavaFX library. For clicks to advance GUI layer, we will implement our own handlers, on top of the click listener of the `Canvas` component.

3.2. Application Layer

3.2.1. Representing the Challenge Data

Representing challenges as objects in memory was one of the very challenging tasks of our design. Since the challenge contains different components (knights, towers and walls) with similar attributes, we did not wanted to use duplicate data to represent them. The components have different behaviors and shapes so this made the design harder. Other than that, the walls are complex structures to be easily represented on a grid. A coordinate is not sufficient to represent a wall, so we came up with some generic structures for wall representation. We have decided to create a hierarchical structure for Challenge Data representation. This design of this representation is further discussed and demonstrated in *Section 4.2.4.*

3.2.2. Working with the Challenge Data

Our objects which contain the challenge data should be manipulated during various actions (play challenge, edit challenge, import challenge, export challenge). We have decided to put the classes about these algorithms in Managers subsystem and make them accessible to all other subsystems. This way, the necessary method calls to algorithms are done without any further instantiation or configuration.

3.3. Storage Layer

Our game will require access to storage devices as previously discussed in *Section 2.3*. This layer will interact with the challenge data part of the application layer. The stored information will be in Base64 form to make it consistent with shareable challenge codes. The translation of data between these two layers (application data & storage) will be done through a two-way conversion method.

4. Low-level design

4.1. Object design trade-offs

4.1.1. Functionality vs Usability

Our game offers a very basic control system such which is drag and drop. We have chosen usability over functionality as we've made mouse as the only interaction device. Then, the game becomes easy to understand and play. Since our game will be easy to interact so one can easily adapt as well. We could have introduced keyboard presses to the game to increase functionality but we did not choose that to make it too complex for the players with ages below 12.

4.1.2. Efficiency vs Portability

Our game will be implemented in Java environment and will use Java virtual machine which reduces efficiency but makes our program OS independent. Thus, we have chosen portability over efficiency. We predict that even the inefficiently-implemented game algorithms will not require significant resources (in terms of time and memory) since the game itself is a simple one.

4.1.3. Reliability vs Rapid Development

In our design we will make our game reliable enough so that a normal player won't be able to end up with impossible or invalid challenges. However, one can apply reverse-engineering on our Base64 challenge codes to create instances of those problematic challenges. We’ve decided to leave out this reliability issue for the sake of rapid development.

4.2. Final Object Design

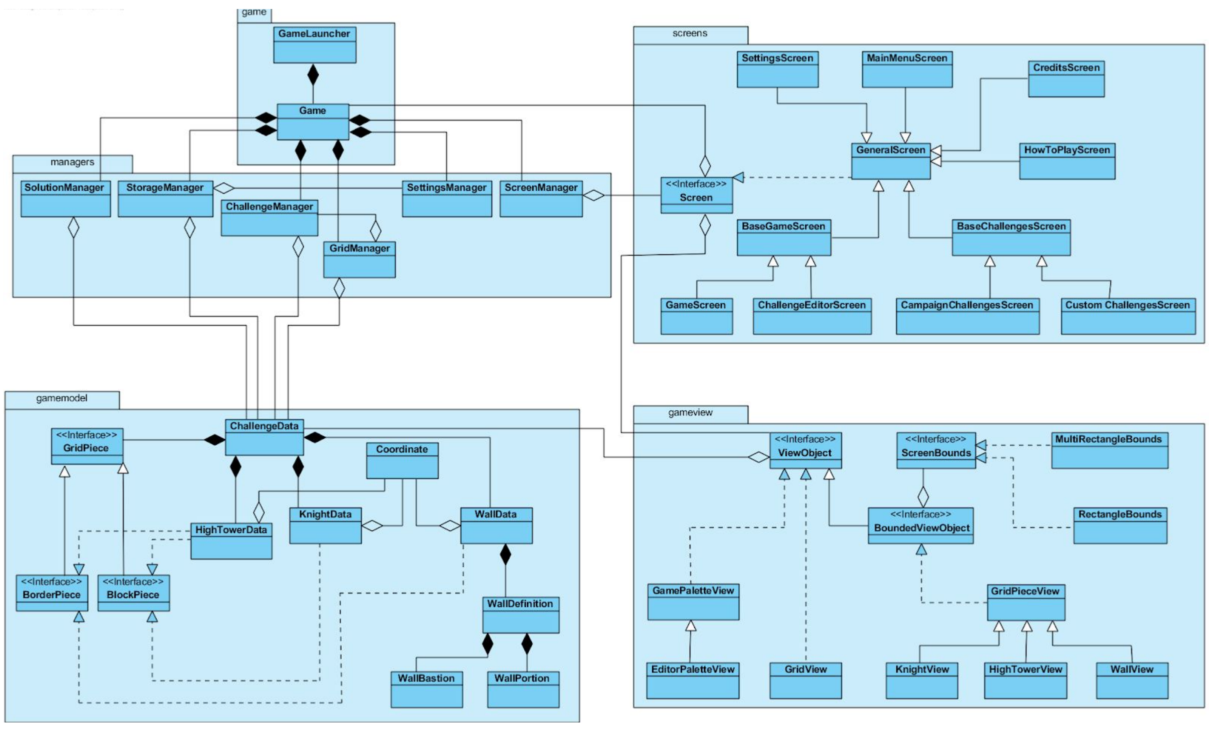

Figure 2: The general class diagram

This section will discuss the classes and their design. As shown in *Figure 2,* we have designed our subsystems’ low-level structure and their interactions in an packaged fashion. The packaging of the classes are done in parallel with the subsystem definitions (See *Section 4.4* for further explanation on packaging). The following subsections will demonstrate the further details of the classes.

4.2.1. Game Instance Classes

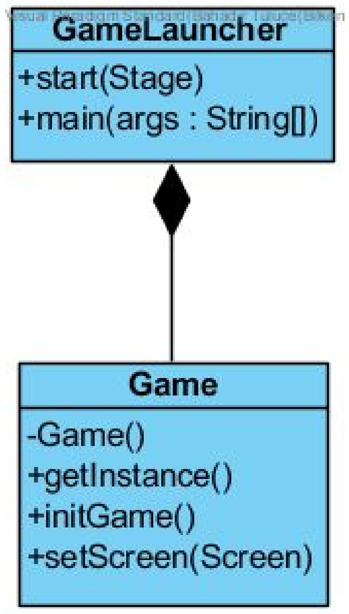

Figure 3: Game instance classes

4.2.1.1. Game

Game class instantiates `ChallengeManager`, `GridManager` and `ScreenManager` on its initialization. It also ensures that only one instance of Game is present at any given moment. (See *Section 4.3.1* for further explanation on Singleton Design Pattern)

4.2.1.2. GameLauncher

`GameLauncher` contains the main method of the program and it launches the game as a JavaFX application.

4.2.2. Manager Classes

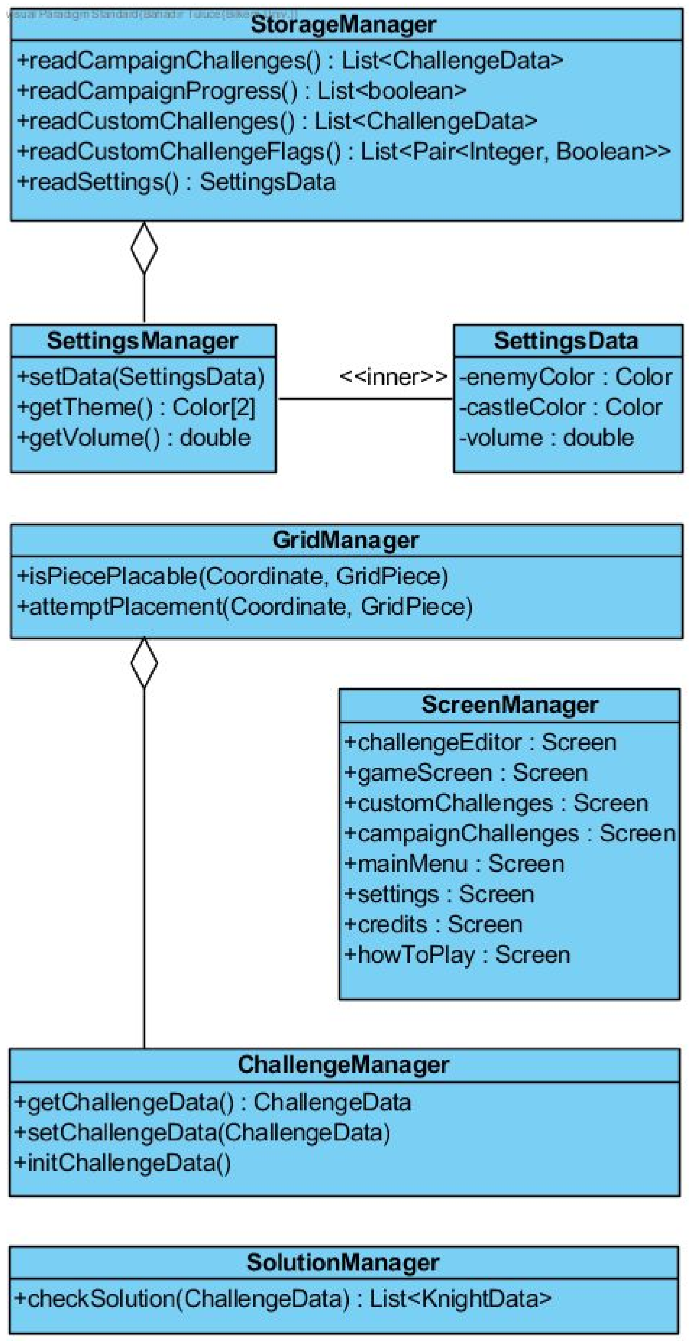
Visual Paradigm Standard
StorageManager
+readCampaignChallenges() : List<ChallengeData>
+readCampaignProgress() : List<boolean>
+readCustomChallenges() : List<ChallengeData>
+readCustomChallengeFlags() : List<Pair<Integer, Boolean>>
+readSettings() : SettingsData
SettingsManager
+setData(SettingsData)
+getTheme() : Color[2]
+getVolume() : double
<<inner>>
SettingsData
-enemyColor : Color
-castleColor : Color
-volume : double
GridManager
+isPiecePlacable(Coordinate, GridPiece)
+attemptPlacement(Coordinate, GridPiece)
ScreenManager
+challengeEditor : Screen
+gameScreen : Screen
+customChallenges : Screen
+campaignChallenges : Screen
+mainMenu : Screen
+settings : Screen
+credits : Screen
+howToPlay : Screen
ChallengeManager
+getChallengeData() : ChallengeData
+setChallengeData(ChallengeData)
+initChallengeData()
SolutionManager
+checkSolution(ChallengeData) : List<KnightData>

Figure 4: Manager classes

The purpose of the `managers` package is separating out complicated algorithms and application-dependent functions from GameModel and GameView classes. Using these managers, other classes can perform complicated tasks such as reading challenges from disk and checking the solution of a chellenge, with single method calls.

4.2.2.1. StorageManager
`StorageManager` class allows `ChallengeData` to be stored as files in the game and saving of the user process as well as the preferred game settings. Using the information obtained from this class, `CustomChallengesScreen` and `CampaignChallengesScreen` will update themselves accordingly to indicate user progress.

4.2.2.2. SettingsManager
`SettingsManager` class allows changing the game settings according to the interactions received from `SettingsScreen`. This class will provide controlling of several features in the game like volume and theme.

4.2.2.2.1. SettingsData
This is an inner class of `SettingsManager` class. Using this class, theme and volume of the game, indicated in `SettingsManager` will be stored and passed among objects.

4.2.2.3. GridManager
This is a class for manipulating the grid in game and in challenge editor. Utilising this class, `ChallengeData` will be updated according to the placement attempts.

4.2.2.4. ScreenManager
This class is for storing with the `Screen` instances in the game. This manager has public `Screen` type attributes for making transitions available within the game. The functionalities and design of `Screen` instances are explained in *Section 4.2.3.*

4.2.2.5. ChallengeManager
This class is for communicating with the `ChallengeData` object. Utilising this object, status of the current `ChallengeData` will be initialized, changed or received to be used in other classes for the regarding uses.

4.2.2.6. SolutionManager
This class is for checking a `ChallengeData`’s solved/unsolved status. It executes an algorithm on the grid of the given `ChallengeData` and returns a `List` of incorrectly positioned `Knight` objects if any.

4.2.3. Screen Classes

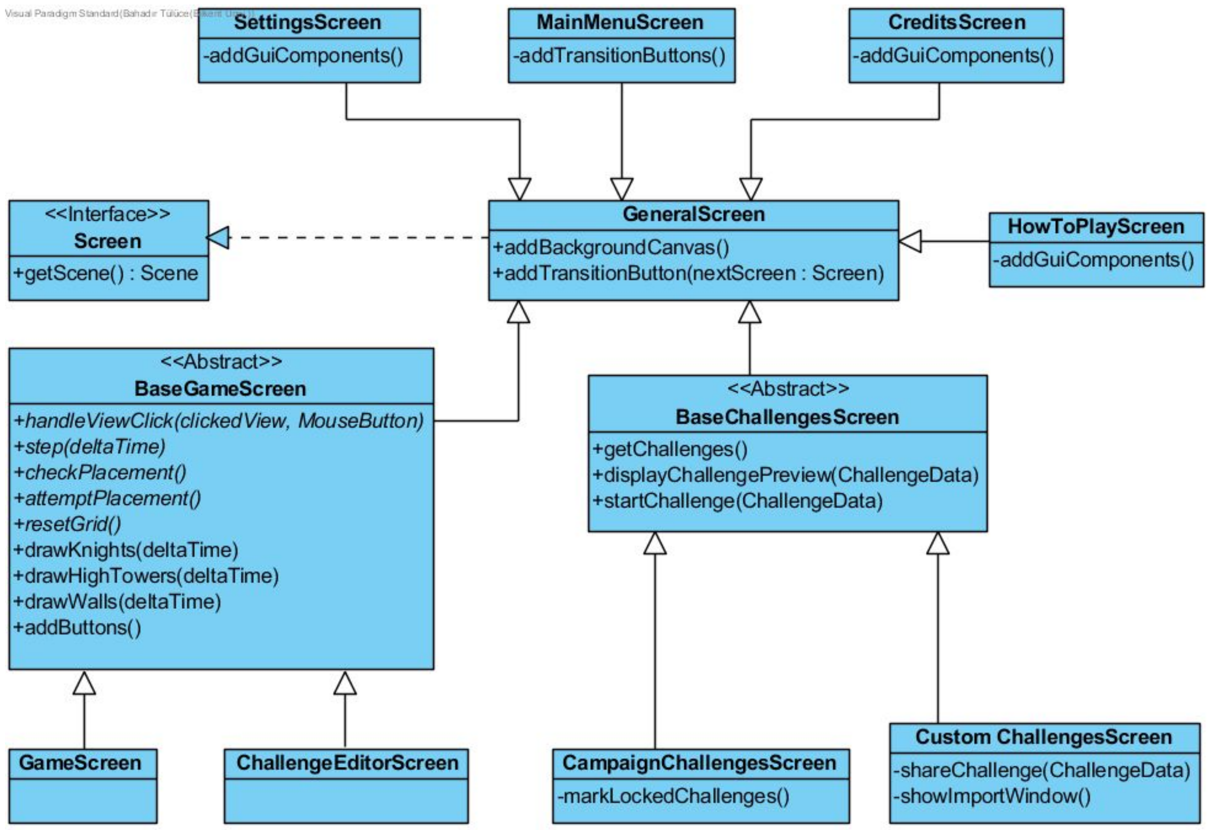

Figure 5: Screen classes

The `screens` package is consisting of all the screens in the game. Screens will interact with the user in order to receive their input to perform actions in the game and update the display accordingly. All of them return a Scene object that gets modified as the display is updated.

4.2.3.1. Settings Screen
This class will set the Graphical User Interface (GUI) components for the settings screen to provide the user with an interactable user interface to change the settings of the game.

4.2.3.2. Main Menu Screen
This class will structure the main menu screen. It will add and structure the buttons for screen transitions.

4.2.3.3. Credits Screen
This class will structure the credits screen. It will add the GUI components for this screen including the credits information and the main menu button.

4.2.3.4. How To Play Screen
This class structures the HowToPlay Screen. It will add the GUI components for this screen including the information on How To Play in a form of tabs/pages.

4.2.3.5. Base Game Screen
This abstract class defines a general screen structure for the Game Screens. This screen will provide a generic definition of standard structure for playing and creating challenges using the mutual components. But the distinguishing methods will be implemented separately in the sub-classes.

4.2.3.6. Game Screen
This class defines the structure of the game screen (for both custom challenges and campaign challenges) extending the `BaseGameScreen`.

4.2.3.7. Challenge Editor Screen
This class defines the structure of the editor screen for Challenge Editor extending the `BaseGameScreen`.

4.2.3.8. Base Challenges Screen
This abstract class defines a general screen structure for Challenges Screens. This screen will provide a generic definition of standard methods for viewing the challenges, but the distinguishable methods will be implemented separately in the sub-classes.

4.2.3.9. Campaign Challenges Screen
This class implements the distinguishable features for Campaign Challenges Screen. Extending the `BaseChallengesScreen`, it implements an additional method for marking locked challenges.

4.2.3.10. Custom Challenges Screen
This class implements the distinguishable features for Custom Challenges Screen. Extending the `BaseChallengesScreen`, it implements an additional methods for the process of importing challenges.

4.2.3.11. Screen
This interface defines a generic structure for all screens in this implementation.

4.2.3.12. General Screen
This class is the parent class for all `Screen` classes. It will structure the generic elements in all screens like transition buttons and the background.

4.2.4. Game Model Classes

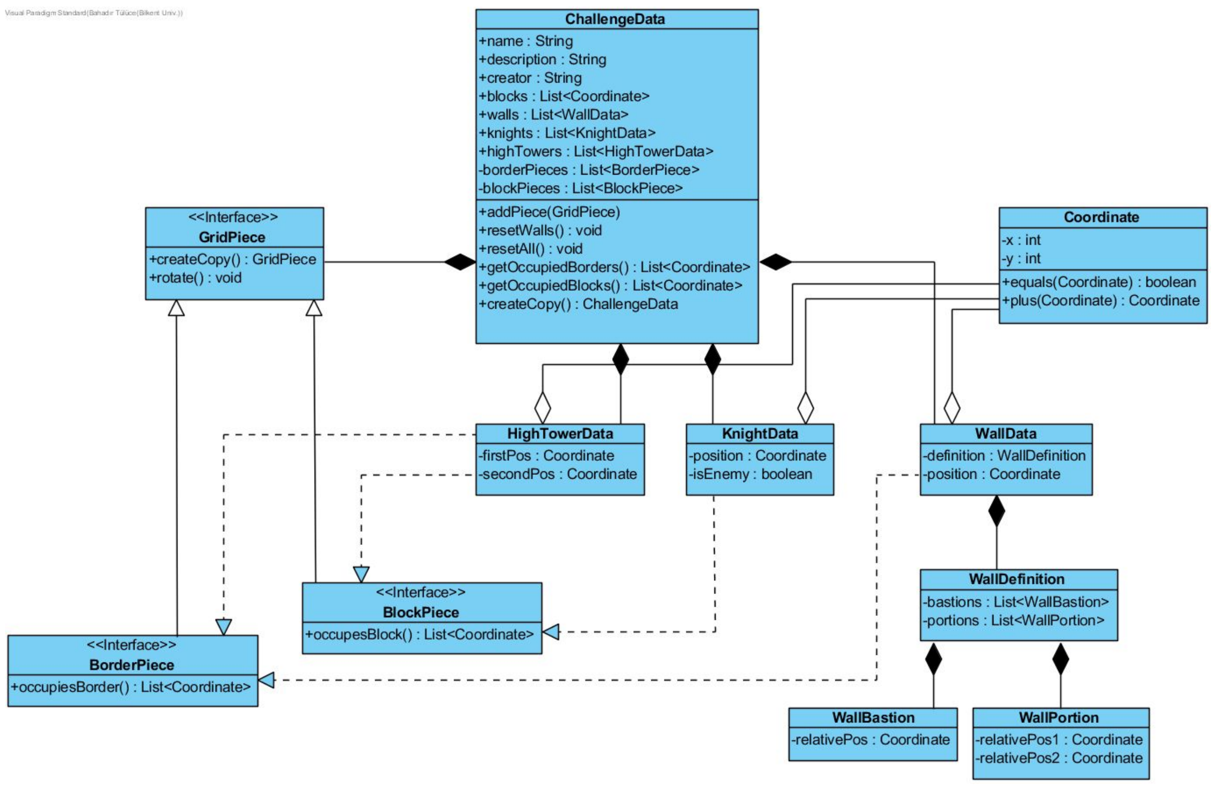
Visual Paradigm Standard(Bahadır Tülüce(Bilkent Univ.))
ChallengeData
+name : String
+description : String
+creator : String
+blocks : List<Coordinate>
+walls : List<WallData>
+knights : List<KnightData>
+highTowers : List<HighTowerData>
-borderPieces : List<BorderPiece>
-blockPieces : List<BlockPiece>
+addPiece(GridPiece)
+resetWalls() : void
+resetAll() : void
+getOccupiedBorders() : List<Coordinate>
+getOccupiedBlocks() : List<Coordinate>
+createCopy() : ChallengeData
<<Interface>>
GridPiece
+createCopy() : GridPiece
+rotate() : void
Coordinate
-x : int
-y : int
+equals(Coordinate) : boolean
+plus(Coordinate) : Coordinate
HighTowerData
-firstPos : Coordinate
-secondPos : Coordinate
KnightData
-position : Coordinate
-isEnemy : boolean
WallData
-definition : WallDefinition
-position : Coordinate
<<Interface>>
BlockPiece
+occupesBlock() : List<Coordinate>
<<Interface>>
BorderPiece
+occupiesBorder() : List<Coordinate>
WallDefinition
-bastions : List<WallBastion>
-portions : List<WallPortion>
WallBastion
-relativePos : Coordinate
WallPortion
-relativePos1 : Coordinate
-relativePos2 : Coordinate

Figure 6: Game model classes

4.2.4.1. ChallengeData

This class models all different Challenge objects in the game and keeps the regarding information. This class will include all the game objects that are existent in challenges (i.e. knights, high towers and walls). Information upon the challenge status is held in this object to be utilised in the gameplay.

4.2.4.2. GridPiece

This interface will define a generic functionality on grid pieces.

4.2.4.3. BorderPiece

This interface will define a generic structure for objects occupying some borders of the grid.

4.2.4.4. BlockPiece

This interface will define a generic structure for objects occupying some blocks of the grid.

4.2.4.5. Coordinate

This class will be used to define and store the positions of game objects on the grid.

4.2.4.6. HighTowerData
This class will be used to define the structure of a high tower object. The information is stored as two coordinates which define its position on the grid.

4.2.4.7. KnightData
This class will be used to define the structure of a knight object. The information is stored as a coordinate to define its position on the grid. Its also has a boolean called `isEnemy` which determines its type (i.e. color) in the game.

4.2.4.8. WallData
This class will store the structure and the position of a wall as a game object. This object had two attributes, `definition` in type `WallDefinition` and `position` in type `Coordinate`. These two attributes define the shape of a particular wall as described in the `WallDefinition` and its positioning on the grid.

4.2.4.9. WallDefinition

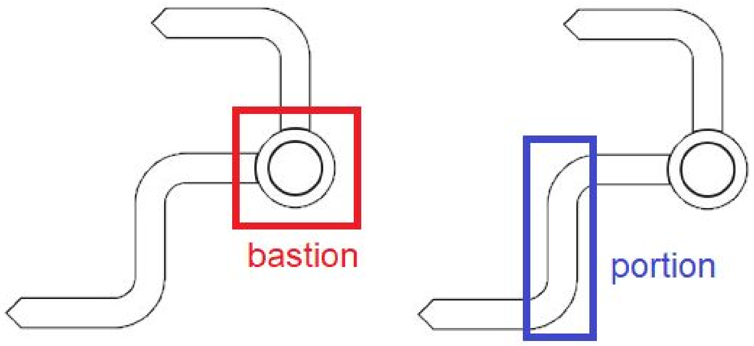

Figure 7: The defined meaning of a “bastion” and a “portion”

This class will be used to define the wall shapes. This class stores `List`s of two objects, one of `WallBastion` type and the other of `WallPortion` type to define the wall shape. Their meanings are given in the *Figure 7* and their design description will be given in the following subsections.

4.2.4.10. WallBastion
This class will be used to define the bastions of a wall. In this class, `relativePos` attribute of type `Coordinate` defines the relative positioning of the bastion with respect to the wall origin.

4.2.4.11. Wall Portion
This class will be used to define the portions of a wall. In this class, `relativePos1` and `relativePos2` attributes of type `Coordinate` define the relative positioning of the wall portion. The wall lies between these two given coordinates.

4.2.5. Game View Classes

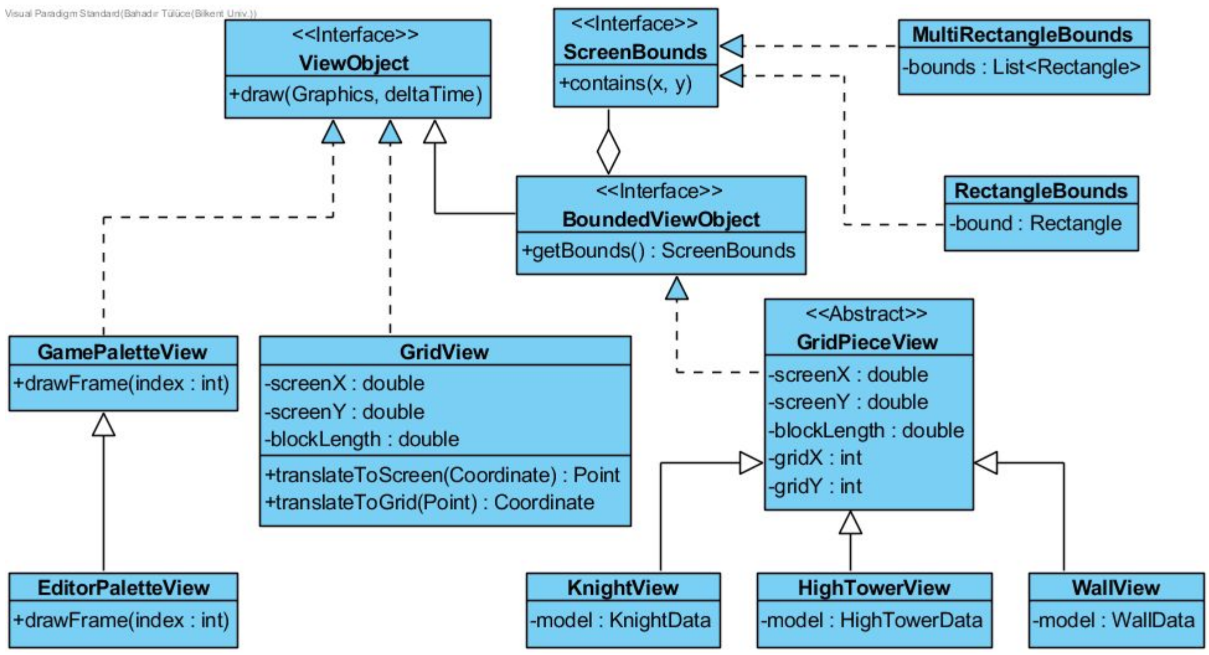

Figure 8: Game View classes

The purpose of game view package is displaying the game state and keeping trach of their screen bounds.

4.2.5.1. ViewObject

This interface defines a renderable canvas object. All of the game view classes implement `ViewObject` interface directly or indirectly and gets rendered on the canvas within the screen they belong to.

4.2.5.2. GamePaletteView, EditorPaletteView

`GamePaletteView` is the view object that displays the wall list that can be placed during a challenge. The `drawFrame(index)` method draws a frame for all these walls and puts their preview image on them. The difference of `EditorPalette View` is that, it overrides the `drawFrame(index)` method and draws an extra column for displaying knights and high towers that can be placed to build a challenge. No screen coordinate is necessary for both classes since the palette is fixed on screen.

4.2.5.3. GridView

This object will be capable of rendering the grid squares to the canvas using screen coordinates and the length of one block. It also can translate any given grid coordinate to screen coordinate or the opposite.

4.2.5.4. ScreenBounds
This interface defines a generic region in the screen.

4.2.5.5. RectangleBounds, MultiRectangleBounds
`RectangleBounds` represents a rectangle region on the screen where `MultiRectangleBounds` represents multiple rectangle regions.

4.2.5.6. BoundedViewObject
This interface extends the `ViewObject` interface with the addition of having bounds on screen. It will be used for handling mouse clicks which are done on view objects.

4.2.5.7. GridPieceView
This abstract class defines a general piece view on the grid. The `draw(Graphics, deltaTime)` method will be implemented in the sub classes according to their type.

4.2.5.8. WallView, KnightView, HighTowerView
The subclasses of `GridPieceView` class are the main view objects of the game. The `getBounds()` method will return `RectangleBounds` for `KnightView`. On the other hand, it will return `MultiRectangleBounds` for `HighTowerView` and `WallView` since they occupy multiple regions on the screen. All three classes have their model instances to update themselves according to their types.

4.3. Applied Design Patterns and Practices

4.3.1. Singleton Design Pattern
In our game, the screens and some of the algorithms (solution checking and piece placement algorithms) will access the appropriate game managers through a class named `Game`. To avoid conflicts between screens and managers, the class `Game` should have one and only one instance present at a time. This will allow us to reference to the same object and same manager throughout the entire process. Consequently, we have decided to apply Singleton Design Pattern on `Game` class. This design pattern enforces a single instances of the `Game` class as we wanted.

4.3.2. Model-View-Controller Design Pattern
In the game screen and editor screen, we have various objects (walls, knights, towers, grid and palette) related to the game . We decided to apply MVC Design Pattern on these objects. This way, we will be able to separate the game logic and the displaying process of these. We think that it will provide us an easier implementation process for both model and view algorithms. Since the unnecessary view attributes are separated from the models, it will be simpler to implement a

model algorithm. Similarly, since there are only the attributes related to the displaying properties in the view classes, it will be simpler to draw them on screen. Other than eases in implementation, the MVC design pattern also provides us extendibility. It enables the implementation of the same game on a different platform (e.g. Mobile, Virtual Reality) by re-writing only the view classes without touching the models and their algorithms.
Our MVC pattern is in the following form:

Model:	GameModels subsystem (`ChallengeData`, `WallData` etc...)
View:	GameViews subsystem (`GridView`, `WallView` etc...)
Controller:	`GameScreen` and `ChallengeEditorScreen` classes

4.3.3. Immutable Design Pattern

While implementing our game models, we would prefer a reliable and secure implementation process, since the models make up the core logic of the game. For this purpose, we should set some rules on data manipulation like restraining modify and access rights on our models. Thus, we have decided to apply Immutable Design Pattern on some of our crucial model objects. We have decided that a `Coordinate` should not be able to change, thus we will set its attributes as `final` and we have designed the `plus(Coordinate)` method as a “new-object-returning” method instead of “self-modifying” method. Similarly, we have used the same idea for `WallDefinition`s and its inner components (`WallBastion` and `WallPortion`) These classes cannot be modified after they are instantiated so it should be guaranteed that a wall won’t change its shape (i.e. its definition) after its creation.

4.3.4. Inheritance as an Object-Oriented Practice

We have noticed that some of our screen classes have very similar properties and functionalities which will cause duplication of code. Then, we have decided to create base classes for them and include only the differing parts in the subclasses. The major inheritance practices are in our screens subsystem (See *Figure 5*) and they can be summareized as follows:

- All screens have a background canvas which consists of a title on top and they all need to have transition buttons. We have decided to include these functionalities in a class called `GeneralScreen` and all the screens will inherit from this class.
- `GameScreen` and the `ChallengeEditorScreen` both have similar functionalities like drag/drop interactions, piece placements, grid piece drawings, grid resets etc. We have moved these functionalities in a class called `BaseGameScreen` and made the other two subclasses of it.

Other than the screens, we have also designed our interfaces in a hierarchical way. The interfaces inherit abstract methods from each other to extend the functionality of the implementing objects. Examples for this type of inheritance are as follows:

- As shown in *Figure 9*, `GridPiece`s can be `BorderPiece`s if they occupy a border on the grid, they can be `BlockPiece`s if they occupy a block on the grid, or they can be both (i.e. a high tower occupies a border and two blocks).
- As shown in *Figure 10*, `BoundedViewObject` is a `ViewObject` with bounds on screen.

4.3.5. Polymorphism as an Object-Oriented Practice
We used the polymorphism concept to keep track of the pieces that are placed on the grid, in `ChallengeData` class. This way, adding a piece to the grid is simply done through a single method named `addPiece(GridPiece)`. This method handles the placement all kinds of grid pieces. Also, `ChallengeData` class contains two polymorphic lists to deal with the occupied blocks. One list is made of `BlockPiece` interfaces that keeps track of the occupied blocks where the other is made of `BorderPiece` interfaces that keeps track of the occupied borders.

4.4. Packages

4.4.1. Internal Subsystem Packages
We have decided to split our classes into packages according to the subsystem they are belong to. This way, we can make use of package private accesses more efficiently and the classes will be in a more organized way. We thought that, this organization will make the implementation faster. According to our subsystem design, the following package names are set: `managers`, `screens`, `gamemodel`, `gameview`.

4.4.2. Other Internal Packages
We have decided that we might need extra functionality and extra utility classes for a better data manipulation and easier/readable code writing. This package might include classes like `FileUtils`, `DrawUtils` and `Rectangle2D`. Further utility classes will be added as needed in the implementation stage.

4.4.3. External Packages
There are a number of external packages for our application to be properly mapped to the hardware. All of these external packages are included in the standard library of Java. Examples for these packages might be `javafx.scene`, `javax.sound` and `java.util`. Further external packages will be included as needed in the implementation stage.

4.5. Class Interfaces
We have decided to make use of interfaces for creating extendible and generalizable classes. The interfaces are mainly involved in GameViews and GameModels.

4.5.1. Interfaces for GameModels

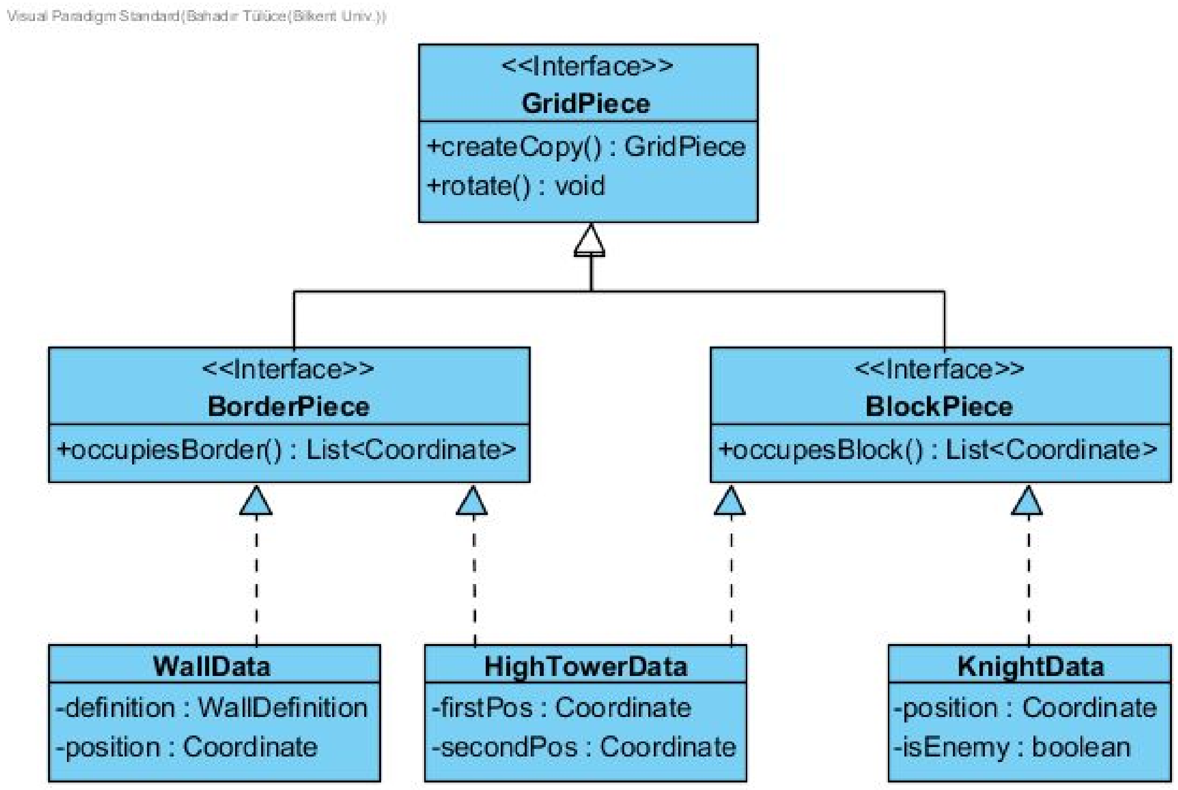

Figure 9: Interfaces in GameModels subsystem

The classes `WallData`, `KnightData`, `HighTowerData` implement the interface `GridPiece`. `GridPiece` interface has the general functionality on grid pieces like creating copy of a piece and rotating it. It has two subinterfaces: `BlockPiece` and `BorderPiece`. `BlockPiece` interface indicates that the implementers occupy some blocks of the grid, where `BorderPiece` implementers occupy some borders of the grid. See *Section 4.2.4* for the detailed explanation of their instances and further explanations on the methods.

4.5.2. Interfaces for GameViews

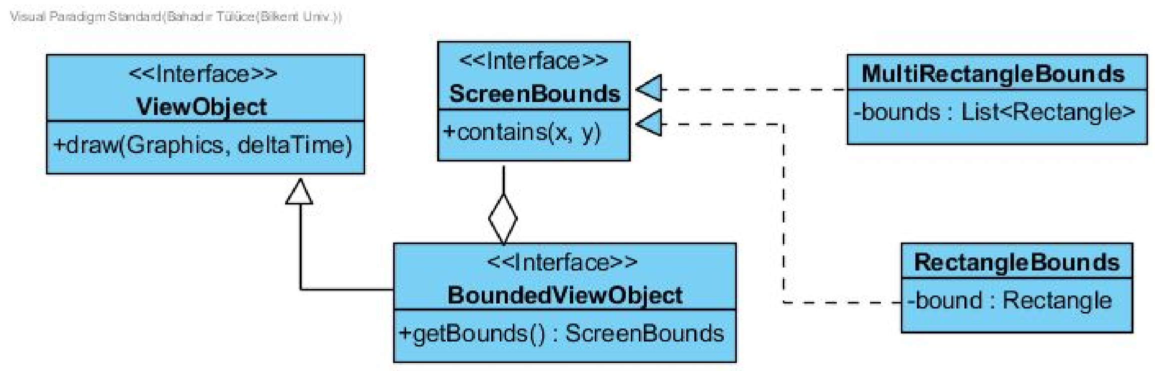

Figure 10: Interfaces in GameViews subsystem

We have decided that all the view objects should have the same generic `draw()` method so we have created `ViewObject` interface for that. The `BoundedViewObject` instances have bounds on the screen to make mouse click handlings easier. See *Section 4.2.5* for the detailed explanation of its implementing classes and further explanations on methods.