BILKENT UNIVERSITY

CS 319

Object Oriented Software Engineering

DESIGN REPORT ITERATION 2

28 April 2019

QUADRILLION

by The Group

Ana Peçini

Krisela Skënderi

Muhammad Hammad Malik

Ünsal Öztürk

This report is submitted to GitHub repository in partial fulfilment of the requirements of CS319 course project.

Table of Contents

1. Introduction

1.1 Purpose of the system

We aim to develop a 2-D desktop based IQ puzzle board-game, which is based on the Quadrillion board game designed by 'SmartGames': a company that develops IQ puzzles with innovative game mechanics which are designed to stimulate cognitive skills such as concentration, problem solving and logic.

There are 4 magnetic grid pieces that can be rotated and/or flipped and arranged together in any combination to make the board of the game. The grid pieces also have a fixed opposite- colored area, where none of the puzzle pieces can fit and thus, this space on the board has to be left empty while attempting a solution. Besides this, the game consists of 12 different puzzle pieces, each with a unique configuration and color. The main objective of the game is to arrange these different puzzle pieces in such a way that they perfectly fit on the created board, without leaving any empty spaces without overlapping any of the "blocks" (the opposite- colored area) positions on the grid. The original game comes with a booklet of grid arrangements that are sorted according to their difficulty level. Each arrangement suggested in the booklet can have at most one solution, but the player can come up with his own grid designs by following simple guidelines; these grid designs must have at least one solution but may have many as well [3].

Our purpose is to modify this game and create a digitalized version of this board-game 'Quadrillion'; We decided to name it 'Gazillion' as it would be a much more advanced version of the original board-game with many modifications which are later discussed. The dynamics of the original game are augmented by adding new immersive features while preserving the core mechanics of the board game.

The modifications and new features include:

· Three modes with different goals and interactive maps to make the game more interesting,

· Player inventory, health and budget,

· Time constraints for different levels to make the game more challenging,

· A shop, where power ups, different themes and sounds can be purchased.

This report consists of the design goals for implementing the proposed game, “Gazillion”, the high-level software architecture followed by subsystem services and low-level design.

1.2 Design goals

We will now proceed with information about the design goals of the proposed application; this is important to demonstrate in order to clarify and analyze the quantities on which we are going to focus on during the course of this project. The non-functional requirements of the system were mentioned and explained briefly in the analysis stage. The main design goals of our proposed system are as follows:

1.2.1 Ease of Use and Learning

As the game would generally be played by children in and around the ages of 7-10, we will have a special focus on designing and implementing the user interface of the system in a manner which makes it not only easier to understand but also stimulates interest in the users. The mechanics of the game would be generally be the same as the original board-game, ‘Quadrillion’ and would also be similar to many online board games of the like, with which children are familiar. The main menu will have less than 7 buttons and each of them will be large enough to be seen and easy to understand. There would be an Instruction button on the main menu as well, which informs the user about-- the objects of the game, the features of the game, the different modes and how to play each mode in simple English language. Our aim is to make it such that any user above the age of 7 can understand the functionality of the game, at most, in 10 minutes.

1.2.2 Portability

Portability refers to the ability of an application to move across environments, not just across platforms [4]. It is an important feature as it allows one to run the program on another platform. We don’t want our application to be restricted to a particular platform; we want it to be transferred, used and to be able to work on different computer environment. We want the application to reach a wide range of

users. Therefore, we decided to implement the application in Java because Java Virtual Machine provides platform independence.

1.2.3. Performance

As this proposed project is not going to be a database system or an online application but a desktop application on personal computers, we are not concerned with specifying workload requirements such as the performance of the application in case of a large number of users at the same time.

All visible user interfaces such as menus, respond in less than 1 second. The system should be responsive to the Player's actions and should run smoothly to maintain the Player experience above a certain standard. Our game will also ensure that it updates the frame every time an action such as placing the piece on the game board takes place with a frame rate of at least 40 frames per second.

1.3 Trade- Offs

1.3.1 Functionality vs Implementation Time

Up until now, we have decided to include many different features such as coins, health bars, timers, power ups, and also many different modes. We also want to implement an attractive user interface so that our target end users would enjoy the game and be captivated by the graphics. A good use interface facilitates usability and the user experience. However, we have limited time to complete the project and if the time is not favorable we may give up some extra user interface components such as animations and better- looking layouts.

1.3.2 Ease of Using and Learning vs Functionality

Although we added new features and modes to the game as mentioned and described in the analysis report, we did not want to over complicate things by adding more modes or complicated features. We also aim to make the flow of each mode as comprehensible as possible. By adding more features but it would decrease the ease of using the application rather than entertain the user as we must not forget that our target users are children above the age of 7. An average user between the ages of 7-25 will be able to understand the basics of the game in, at most, 10 minutes. Thus,

our focus is on making this application easier to learn and consequently easier to use instead of adding extra features which could confuse the player.

2. High-level Software Architecture

2.1 Subsystem Decomposition

The proposed system will be composed of three different subsystems: the logical subsystem, the user interface, and the management subsystem. This section will make sure to establish these subsystems in such a way that the dependencies of a given subsystem on another subsystem is reduced to a minimum, as much as possible.

The Model-View-Controller pattern was employed in the design of the system architecture, as we concluded that among the well-known system design patterns, the MVC pattern is the most appropriate for the design of our proposed system. The subsystem decomposition was carried out based on the principles of the MVC pattern, resulting in the following subsystems:

- Game Logic Subsystem: This subsystem models a game of Quadrillion using sparse matrix representations and geometric transformations. This subsystem is a collection of all entities that are related to the logic of the game, e.g. pieces, boards, grids, and timers. This subsystem maps to the model component of the MVC pattern.

- User Interface Subsystem: This subsystem models the user interface through which the player observes the state of the Quadrillion Subsystem. This subsystem is the collection of user interface components, such as panels, frames, buttons, listeners, and menus. This subsystem maps to the view component of the MVC pattern.

- Game Management Subsystem: This subsystem is composed of tools and utilities that allow input/output from and to the filesystem, as well as the classes that relay information from the user interface to the model and ask the model to change its state according to the input. For most of the part, this subsystem maps to the controller component of the MVC pattern.

The system architecture is therefore reduced to a state of minimum inter-subsystem dependencies, in comparison to other system architecture patterns,

such as the three-layer architecture pattern, which introduces an unnecessary hierarchy for our use cases.

The decoupling between the aforementioned subsystems will therefore positively impact the speed and efficiency in the implementation stage.

A more detailed description of each subsystem will be given in the next sections.

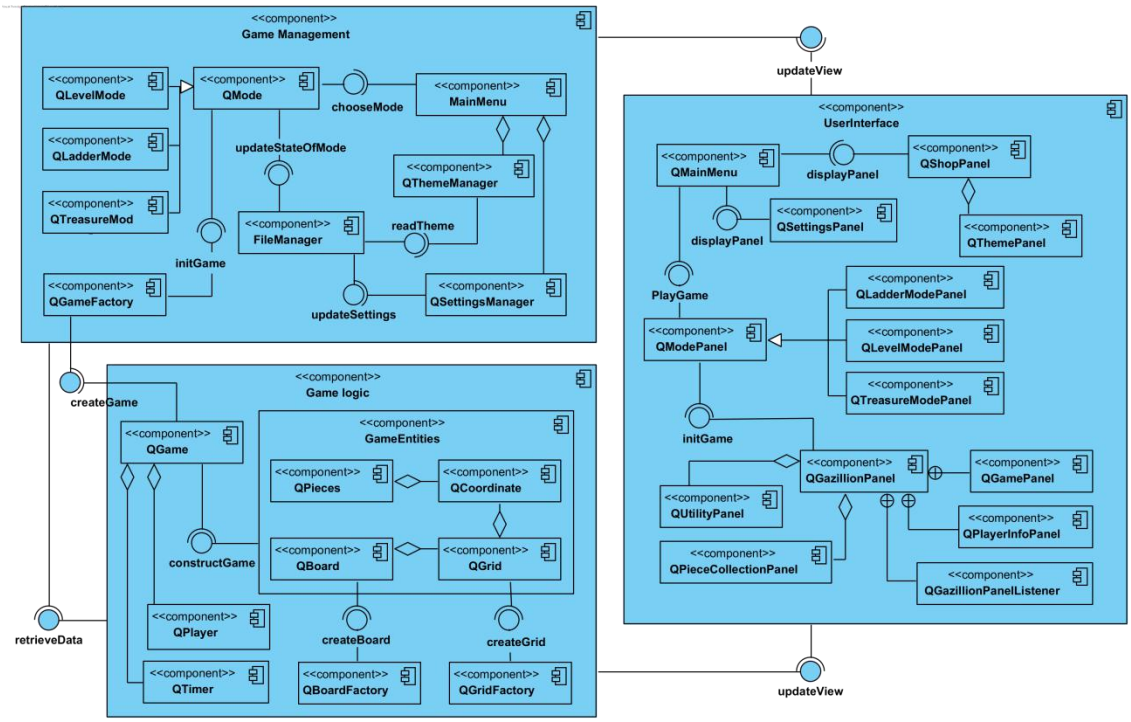


Figure 1. Component Diagram for All Subsystems

2.2 Hardware/Software Mapping

Gazillion will be developed in Java and compiled into an independent .jar executable. Its GUI will be implemented through Java Swing framework, which means that Java Runtime Environment will be required in order to be able to play the game. Since the system will run on the Java Virtual Machine, it will be easily portable to different platforms like Windows, OSX and Linux.

Regarding the hardware requirements of our system, the only input device required to play Gazillion is the mouse. The user can select menu options; navigate around the level maps and click on the level they want to play; select, drag, place, rotate, flip and remove puzzle pieces on the grid during a Quadrillion round; use different power ups by clicking on the corresponding buttons on the screen. For

different actions during the game, different combinations such as double click to rotate by 90 degrees will be used.

The system will work offline and no database will be used to store the game information.

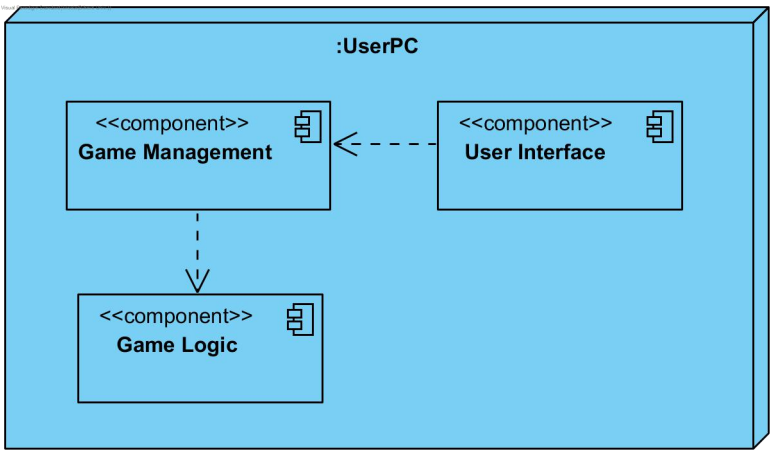


Figure 2. Deployment Diagram

2.3 Persistent Data Management

The proposed system does not make use of intensive read/write operations, and its subsystems are small enough to fit into memory at once. Therefore, it does not need to have a dedicated persistent data management subsystem or service, such as an RDBMS. However, it does need a simple collection of components that stores the state of player progress (e.g. high scores, coins, completed levels in Level Mode, power-ups, etc.) and configurations persistently on disk between application "uptime". For this purpose, the system will employ a simple file I/O library to persist progress data in XML format. The system also has to read-in music files in .wav format (format of choice is due to compatibility reasons), and images in .png format.

2.4 Access Control and Security

Our system will not support any login system and no password protected accounts will be created. Gazillion will not require internet connection to run since it will be an offline desktop application. No user information or game-related information will be stored after the application is closed, which means there is no security issue that our system should deal with.

Regarding the access control, the player is free to navigate back and forth between the different game modes and between the main menu and the game. He is free to adjust the volume and change the theme of the game to any of the unlocked themes displayed in the Settings menu.

The player information, such as health bar, power up inventory and coins will remain unchanged, and while the game is still running the progress that the player has made in a mode will not be erased when he switches to another one. This means that the player can still access the levels he has unlocked in Treasure Mode and Levels Mode, although he may not win any more rewards for completing them a second time. The locked levels are out of player's reach until he has successfully completed the previous level, and for the Ladder mode the game is restarted every time the player runs out of time or exits the mode. The Final Game within the Treasure mode is also locked, and the user cannot start it until he has collected all the 12 puzzle pieces. Additionally, the themes and power ups in the Shop menu are locked if the player does not have enough many to make a purchase.

2.5 Global Software Control

Our application is not a multi-user system and the only actor is the Player. The Player has access to every functionality of all the modes of the game.

2.6 Boundary Conditions

- Initialization

The proposed system will run on the JVM, and it will therefore require an installation of JRE 8u212. The system will not require any installation on the filesystem other than the presence of the .jar file in a specified directory.

Upon first time launch, the application will prompt the user to create a user profile. The progress of the player will be recorded under this profile, and the player will be required to enter a name for the profile.

Starting the application after a profile for the player has been created will prompt the player to either select the previously created profile, or to create a new profile. The player may then choose one of the previously created profiles to play as, or create a new profile to start the game anew.

Alternatively, the player may “import” a profile, which was generated on some other computer, but has physically been moved to the computer on which the player is about to play the game.

- Termination

The proposed system may be terminated by clicking the exit button situated in the menu, or by manually terminating the process using the facilities provided by the operating system. Upon receiving the signal for the termination of the program, the system will try to serialize the current state of the player’s profile and persist this information on the filesystem. If the system was terminated at a critical moment, such as termination during the persisting of the data to the filesystem, the information regarding the state of the player profile will be reloaded from a checkpoint, which is periodically updated in an atomic transactional manner when there is a change in the state of the player profile, the previous state will be persisted.

- Failure

In the case of an error during the execution of the program/services provided by the system, the system will first attempt recover from this error by trying to handle the exceptions thrown. For recoverable exceptions, such as an accidental division by zero, the game state will be rolled back to the previously saved state, in this case the prototype profile information saved by the system will be restored. For serious errors, such as hardware failure or illegal termination, the system will not attempt to provide any fault tolerance, and upon next initialization, it will try to restore the state that was most recently persisted. For disk failures, the player will have no other choice but to create a new profile.

3. Subsystem Services

3.1 Game Logic Subsystem

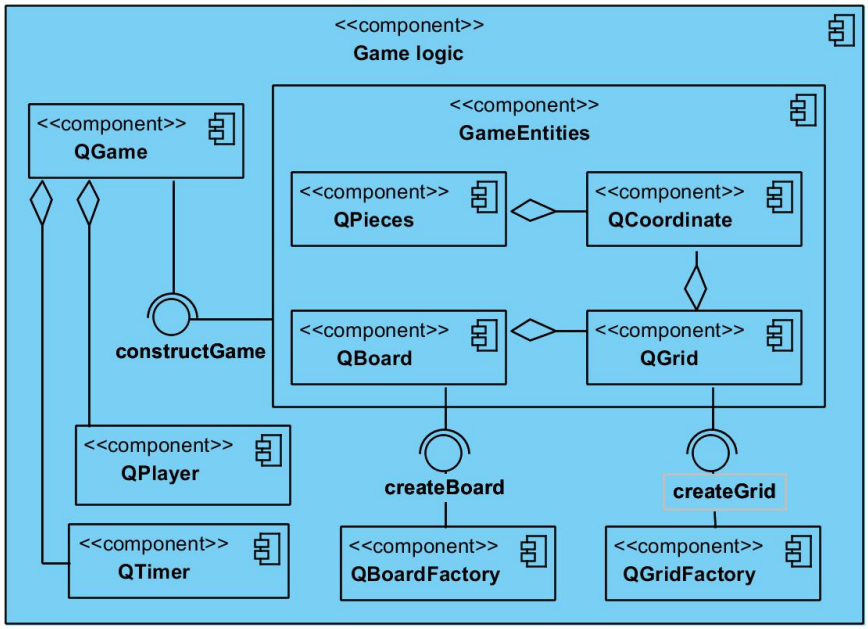


Figure 3. Component diagram for Game Logic Subsystem

The Game Logic Subsystem models the logic of the game by defining finer entities and aggregating them to form more high-level entities that provide high-levels of abstraction. This is achieved through constructing a hierarchy of objects that are highly dependent on one another in terms of functionality. For instance, a QBoard is composed of four QGrids, and a QGrid is composed of 16 conceptual QCoordinates; and whenever a QPiece is desired to be placed on top of the board, the piece requests from the board to be placed, which in turn requests the grids, which in turn requests the coordinates for the hosting of the piece. Any related signal regarding the placement of the piece is then propagated back to the hierarchy. This allows the dependencies to stay within the subsystem hierarchy. When an external component, such as a controller class in the Game Management Subsystem, requests the model to change its state, such as a request for piece placement; it does need to operate within the hierarchy of the Game Logic Subsystem. It communicates with both of the other subsystems: The User Interface Subsystem bases its state on the state of this subsystem (by rendering the current state of this subsystem), and the Game Management allows the manipulation of the state of the Game Logic Subsystem.

Whenever this subsystem modifies the state, the knowledge of the changes is propagated to the User Interface Subsystem.

3.2 User Interface Subsystem

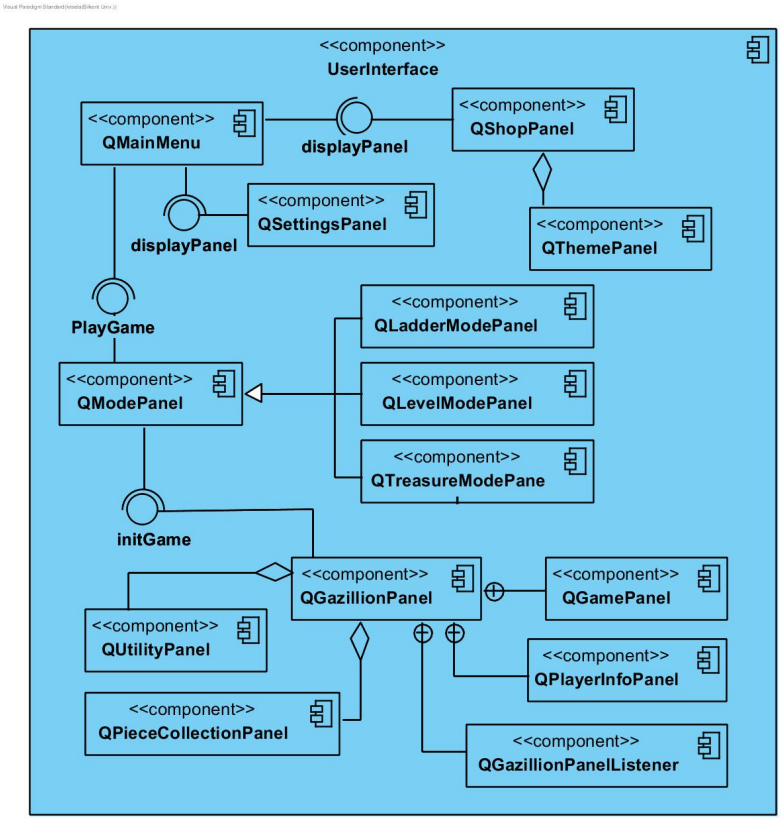


Figure 4. Component Diagram for User Interface Subsystem

The User Interface Subsystem is responsible for the translation of the state of the Game Logic Subsystem to a visually interactive context from the player's perspective. The system displays the relevant details of the model to the player, and provides an interface for the player to manipulate the state of the model through controllers that live within the Game Management Subsystem. This subsystem may not request the model to change its state directly, however it may relay the signals that it receives from the player to the Game Management Subsystem to be evaluated and requested from the model. This allows the subsystem to be easily extendable to newer use cases, due to reduced dependencies.

3.3 Game Management Subsystem

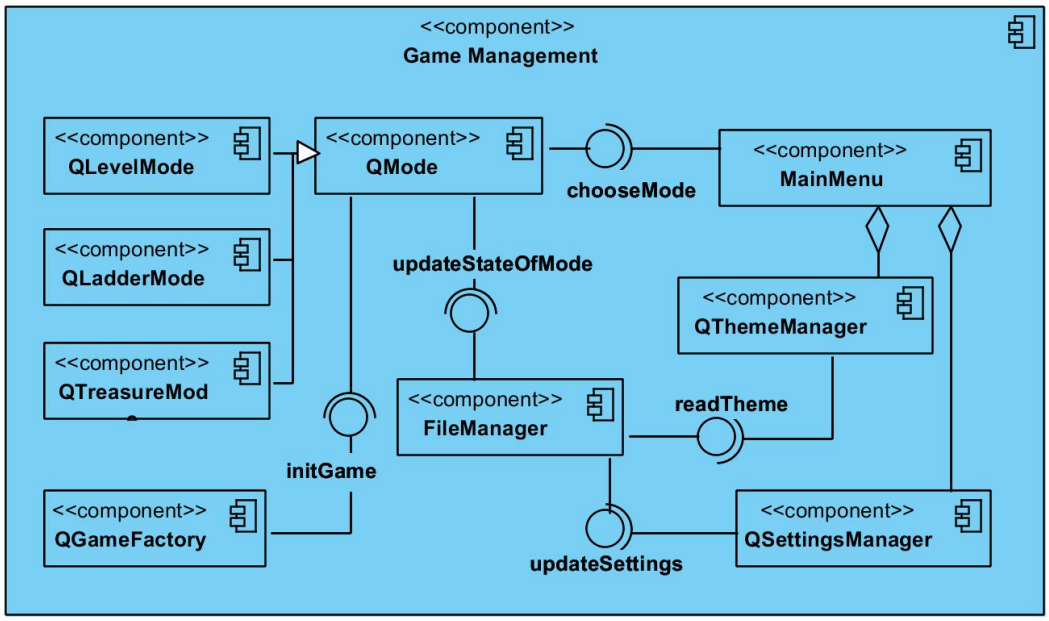


Figure 5. Component Diagram for Game Management Subsystem

This subsystem provides control and I/O components. The I/O components allow the serialization of player progress, preferences and settings configurations. The controller components get input from the User Interface Subsystem and evaluate the action to be taken for the manipulation of the game model. It communicates with both subsystems; however, it may not request the state of the model or it may not ask the User Interface Subsystem to change its state. It has the control of requesting piece placements, piece removals, power-up uses, and many other functional requests from the model.

4. Low-level Design

4.1 Final Object Design

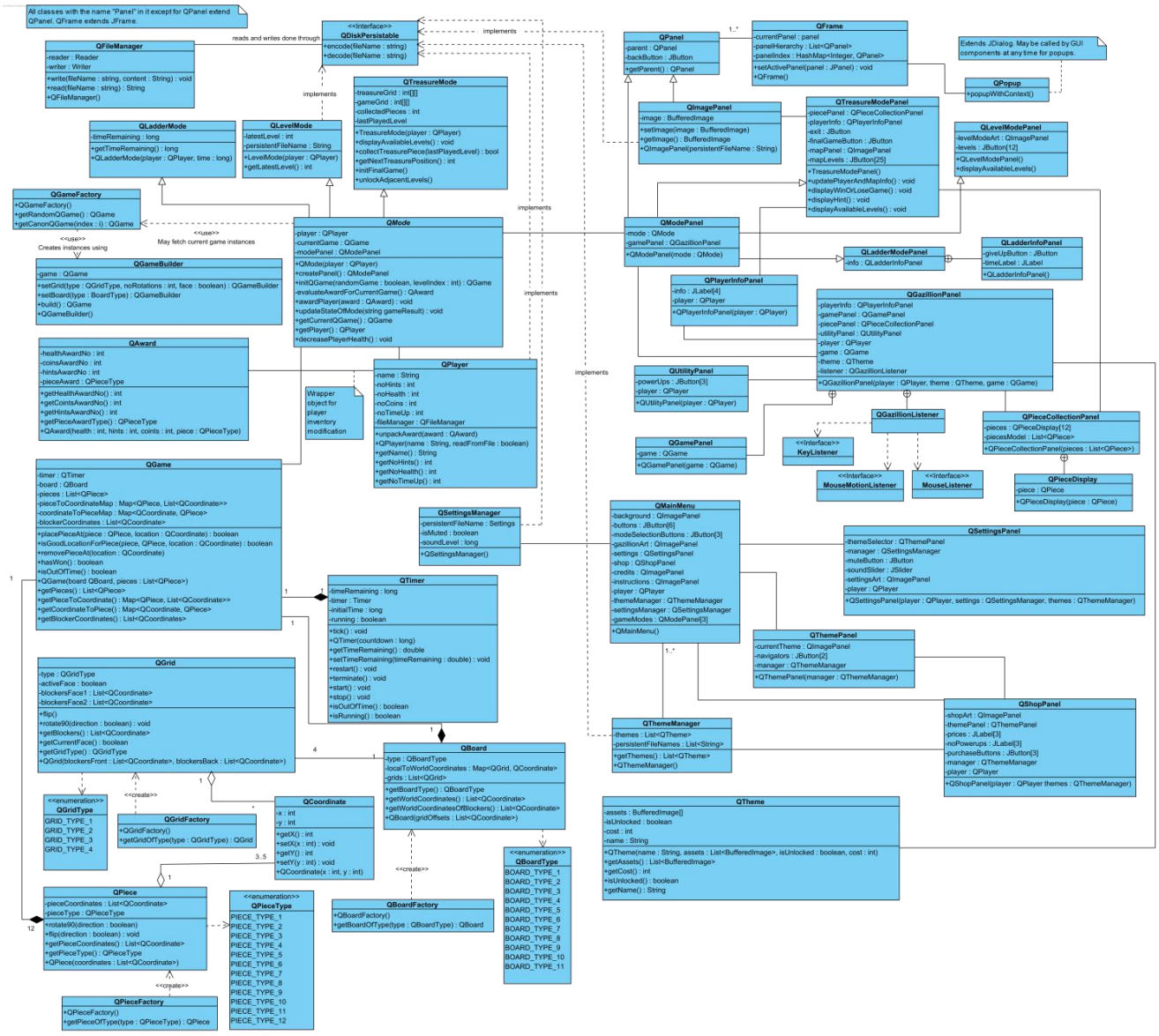


Figure 6. Final Class Diagram

4.2 Object Design Trade- offs

4.2.1 Sparse Matrix vs Row/ Column Major Representation of the Board

The underlying logic of the proposed system, i.e. digitalized Quadrillion, is based on data manipulation on a sparse-matrix representation. The game board is modeled as a matrix, which stored as a series of two- tuples marking the indices on which piece placements are possible. This allows the topology of the game board to be stored in an implicit geometric format, and allows queries and requests to be processed purely

semantically, which requires less code to implement. Another way to design the logic is to use a row or column major explicit matrix format for the game board and pieces. In this design, the geometry has to be explicitly represented in the forms of bounding boxes over subregions of the matrix. This allows the implementation to be more intuitive; however, with this implementation, the logical model forces the queries and requests to be a mixture of geometric and semantic statements, which increases the amount of code. The advantage of this representation is storage efficiency: for a piece placement or removal, the check-up times are , since they only require four bounding box checks and up to five memory accesses, without auxiliary storage. Sparse matrix approach achieves the same computational efficiency via storing extra information regarding the state of the game object (piece to coordinate and coordinate to piece mapping). Therefore, the trade-off between the two fundamental designs is one of storage vs. the simplicity of query/request processing. Since the extra data to be stored in memory does not exceed 1MB, it is not worth considering the advantages of an explicit matrix representation, because there are no storage problems.

4.2.2 Centralized vs Distributed Responsibility for File Persistence

In our design, we decided to distribute the responsibilities for persisting the files on the disk among the classes that require it, rather than having a single class that essentially reads all the necessary information for the system to operate. The advantage of having each class manage its own disk I/O through a utility class is that whenever a new class is added to the system, one does not have to change another class that is responsible for I/O, because such a class does not exist. Rather, the programmer has to define on which files the particular class will operate, and what will their names/directories will be. This allows flexibility when one wishes to extend the game by coming up and implementing new game modes. Another advantage of this approach is that there is no class that causes a bottleneck when it comes to providing assets that reside on the disk: if we were to have a class that answers requests from other classes regarding disk resources, that class would become a bottleneck if there are too many requests, which violates some of the performance related non-functional requirements. The disadvantage of this approach is that a class is completely oblivious to the files managed by other classes. For instance, the

QGazillionPanel does not inherently know what sounds the QSoundManager class offers to be played. This trade-off is alleviated by ensuring that a file is read by a particular class when a class, which is closely coupled, requests a resource that resides on the disk. In the case of QGazillionPanel and QSoundManager, when the panel requests a sound to be played by the manager, it first "queries" the manager regarding the existence of a sound file in memory. If there is no such sound file in the manager, the panel has to come up with a back-up plan. This plan could be one of the following. Either an exception is thrown and no sound is played, or the QSoundManager tries to load the file from the disk, or it plays a default sound. This causes some overhead; however it is not a serious overhead if no disk I/O is performed, which means that the advantages of distributing the responsibility of I/O outweighs the disadvantages.

4.3 Design Decision and Patterns

During the design of the system, many design patterns were employed for simple implementation of the system. This section is dedicated to the discussions of their trade-offs.

4.3.1. Factory Design Pattern

The factory pattern was preferred for the construction of QPiece, QGrid, and QBoard instances, along with the enumeration of the types of these objects according to the number of different functional instances produced by their corresponding factory classes. The main reason behind using this pattern is that the QPiece, QGrid, and QBoard instances are constrained by the original Quadrillion: There are 12 different pieces, 4 different grids, and 11 board configurations. Any other configurations for these objects are implicitly illegal (they are not guaranteed to create a Quadrillion game with a solution). Therefore, it is simpler to pre-define these entities, and request instances of these entities from a factory class.

4.3.2. Builder Pattern

A special type of the builder pattern was used in the construction of a QGame instance, for conveniently constructing QGame objects by specifying only the relevant

information that differentiates a QGame instance from another (i.e. grid & blocking tile configurations and grid topology) during construction.

This builder pattern is quite different than the usual builder pattern. This pattern makes use of many inner classes that makes sure that the fields of the object is set in a particular order. The first invocation of a method on the QGameBuilder sets a field in the QGame object of the builder, and returns an inner class of the QGameBuilder class, which in turn will be used to set another field of the QGame object. The next invocation on this inner class returns another inner class that sets another field, and so on. The order of the returned inner classes is in accordance to the ordering in which the fields of the QGame object are to be set. The grid types have to be set before the grid topology is determined, for instance, and this ordering is enforced as discussed above.

The builder pattern was used in the QGameFactory class to quickly construct QGame instances. For randomly generated QGames, the parameters are randomly determined using the pseudo random number generators offered by the Java API, which are then passed to the builder object. For hard-coded Quadrillion levels, the QGameFactory offers a method which returns a “canon” (as present in the game manual) level.

4.3.3. Prototype Pattern

The prototype pattern is used to cache and modify the state of the player profile in memory. The reasoning behind this pattern is to reduce disk I/O while trying to provide fault tolerance while persisting the player profile to the disk. The player profile is loaded from the disk on initialization and whenever a modification is made on the player profile (e.g. player gains coins after winning a game), the changes are not persisted back on the disk immediately, but rather the in-memory structure of the profile is changed. If this change is deemed to be non-erroneous or if this change does not result in an illegal state, the in-memory prototype is updated accordingly. Whenever a fault occurs in trying to update the player profile, the system rolls back to the prototype that was cached in memory. This mechanism is present throughout the implementation. Any class that implements QDiskPersistable interface makes use of

this mechanism for a better trade off in terms of disk I/O access time and fault tolerance.

This pattern is also used in QSoundManager, which caches a sound stream in memory. When a stream is consumed by a thread that plays the sound contained within the stream, the stream is “refreshed” by the prototype that resides in memory. This way, disk I/O is reduced, and it becomes easier to meet performance related non-functional requirements.

4.3.4. Façade Pattern

This pattern is used in QFileManager. The QFileManager class calls methods provided by the Java API for disk I/O. The classes that should persist files to the disk (e.g. level mode, player profile) call methods from this class instead of the methods provided by the Java API. This ensures that the implementation for disk I/O for the rest of the classes are independent of the methods provided by the Java API. For instance, if the class path is somehow altered by an external factor on some platform in the future, or if somehow the filesystem of the platform of deployment causes some problems with disk I/O, only the implementation of the QFileManager class should be changed, as all the other classes will call the methods provided by the interface of this class.

4.4 Packages

In our application we make a distinction between two kind of packages: packages introduced by us, which include the classes and interfaces closely interrelated with each- other and packages introduced by Java libraries such as swing.

4.4.1 Packages Introduced by Developers

- Game Logic Package

Logic package includes game entity classes and factory classes, responsible for the main logic and construction of the game.

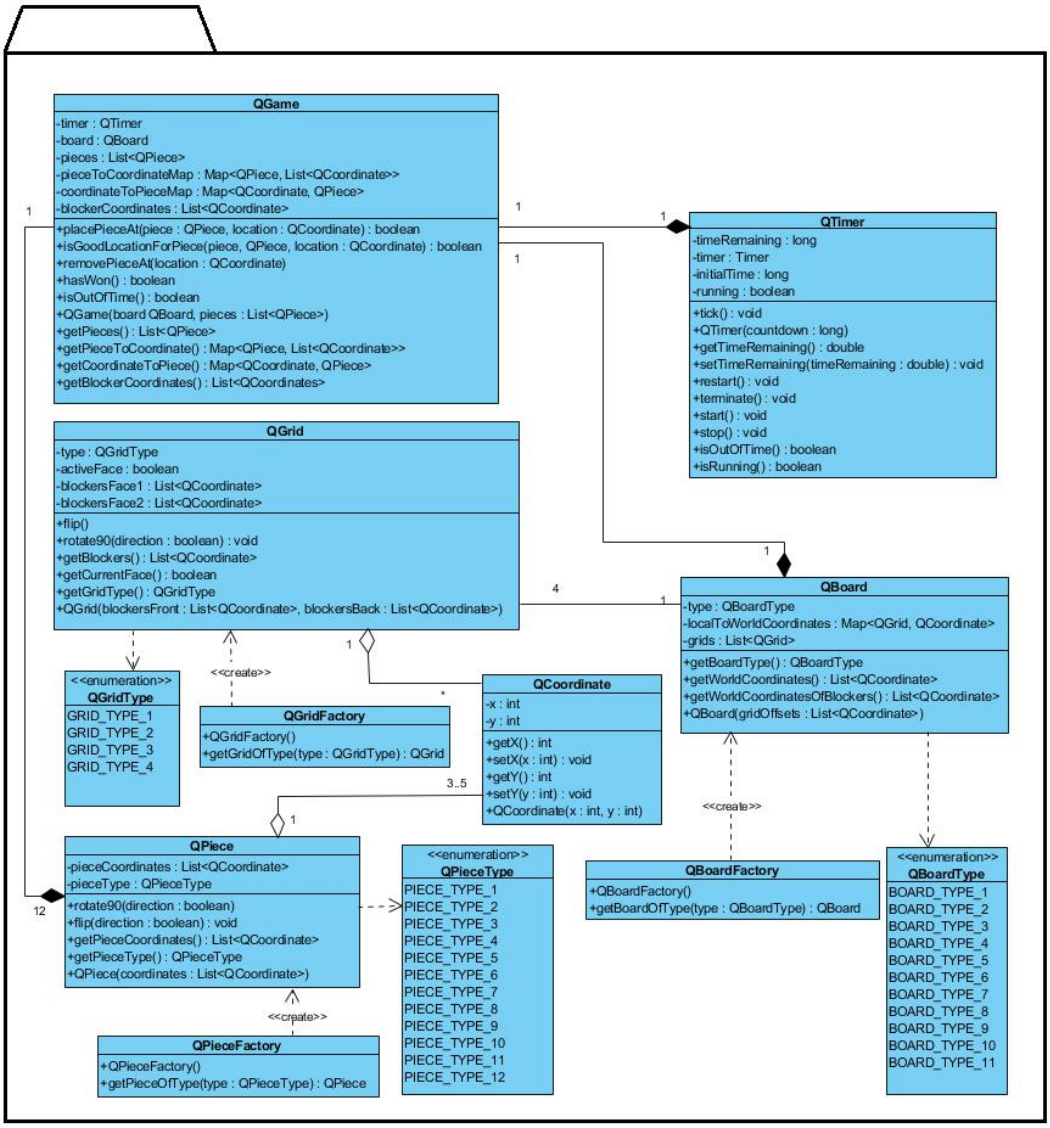


Figure 7. Game Logic Package

- Game Management Package

This package is responsible for the mode management of the game. It provides methods to switch between modes and keeps track of the current level. It includes the QMode, LadderMode, LevelsMode, TreasureMode classes. It also includes classes which provide reading and writing from files. The stored data includes player information, player progress, themes and settings status.

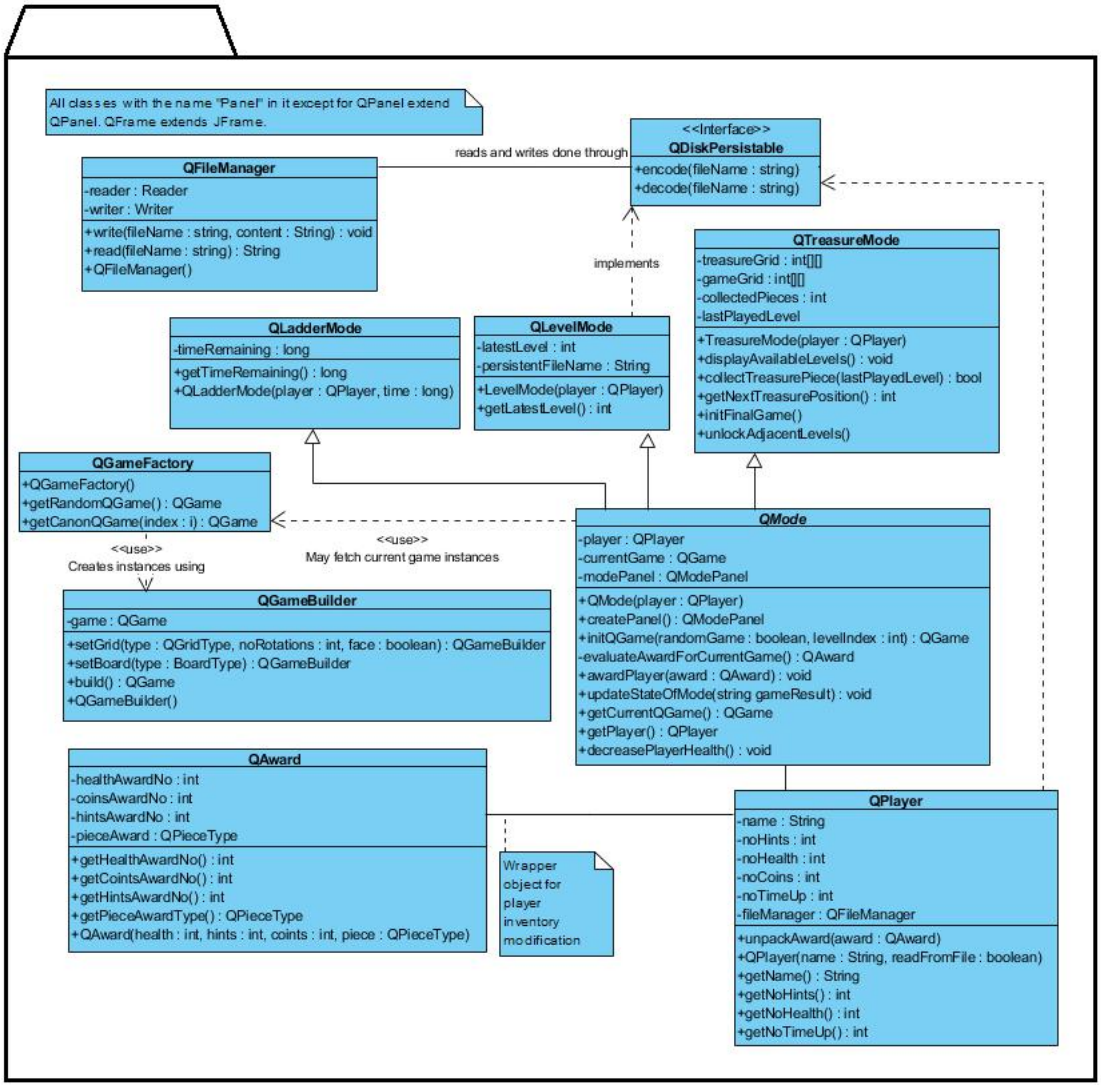


Figure 8. Game Management Package

- Menu Package

This package controls the navigation between different menu components such as settings and shop. (Figure 9)

- GUI Package

This package includes the classes which are part of the User Interface Subsystem such as: QShopPanel, QSettingsPanel, QThemePanel etc. (Figure 10)

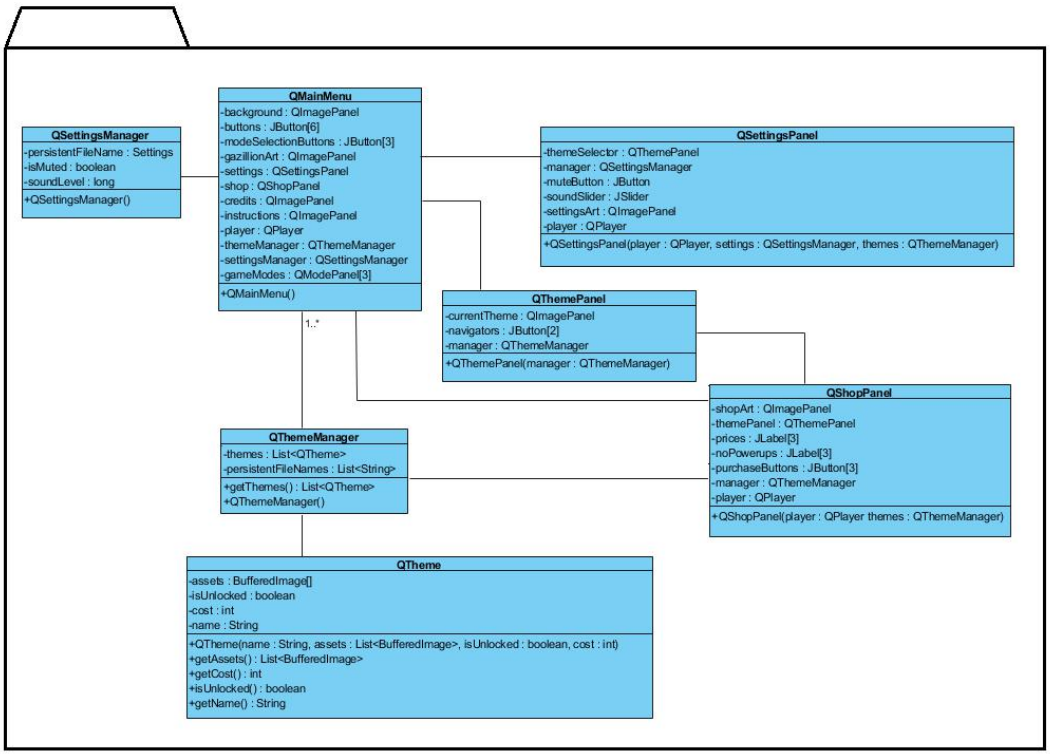


Figure 9. Menu Package

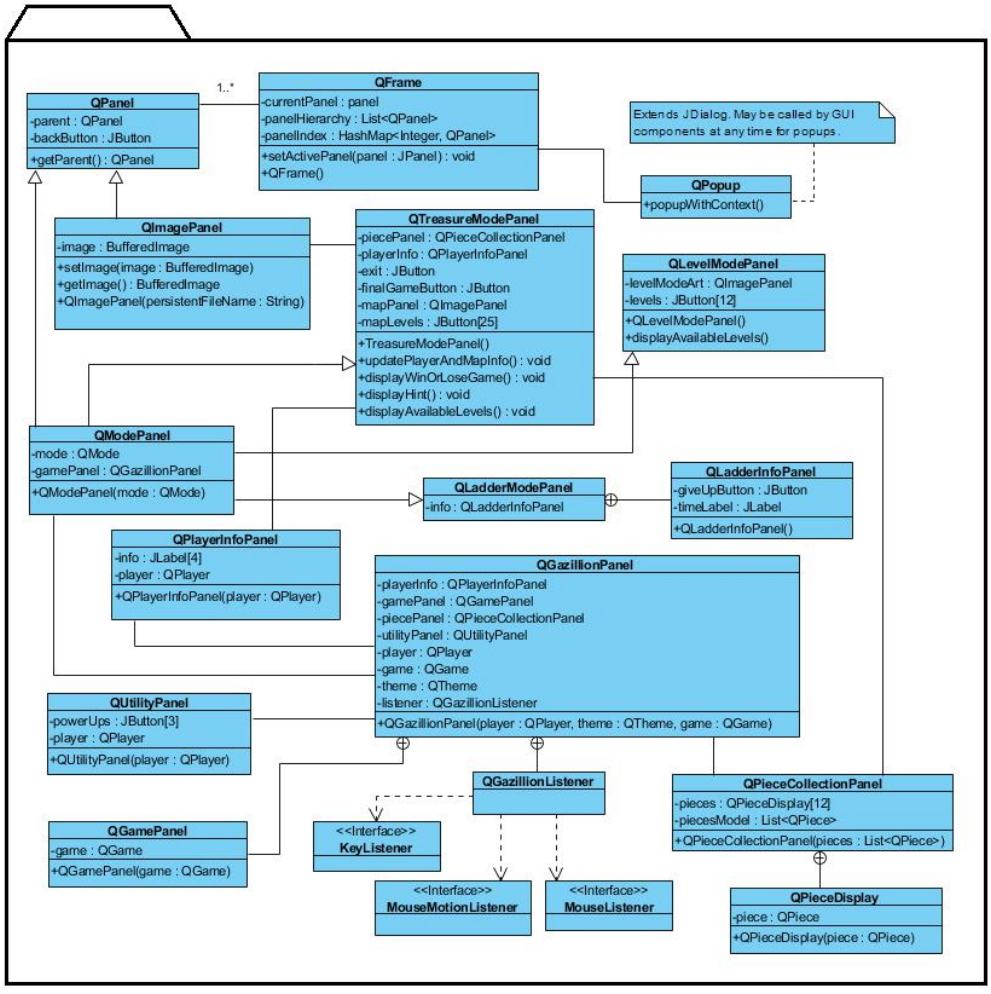


Figure 10.GUI Package

4.4.2 Java Library Packages

- java.util

This package includes ArrayList which is needed for attributes that are a collection of items, such as QCoordinates on the QGrid, QCoordinates on QPieces, QGrids on QBoard, QPieces collected on QTreasureMode, list of QThemes etc. The package is also used for the Enumeration interface to generate the series of elements (grid types, piece types and board types). Furthermore, Map is used to map QPieces to QCoordinates on the board and EventListeners are implemented as part of our controller system.

- java.swing

This package provides the user interface components of the game. Our user interface classes such as QImagePanel, QTreasureModePanel, QLevelModePanel, QLadderModePanel, QFrame etc. extend JFrame or JPanel.

4.5 Class Interfaces

This section is dedicated to explaining the interface exposed by the classes, as present in the class diagram.

4.5.1 Game Management Subsystem

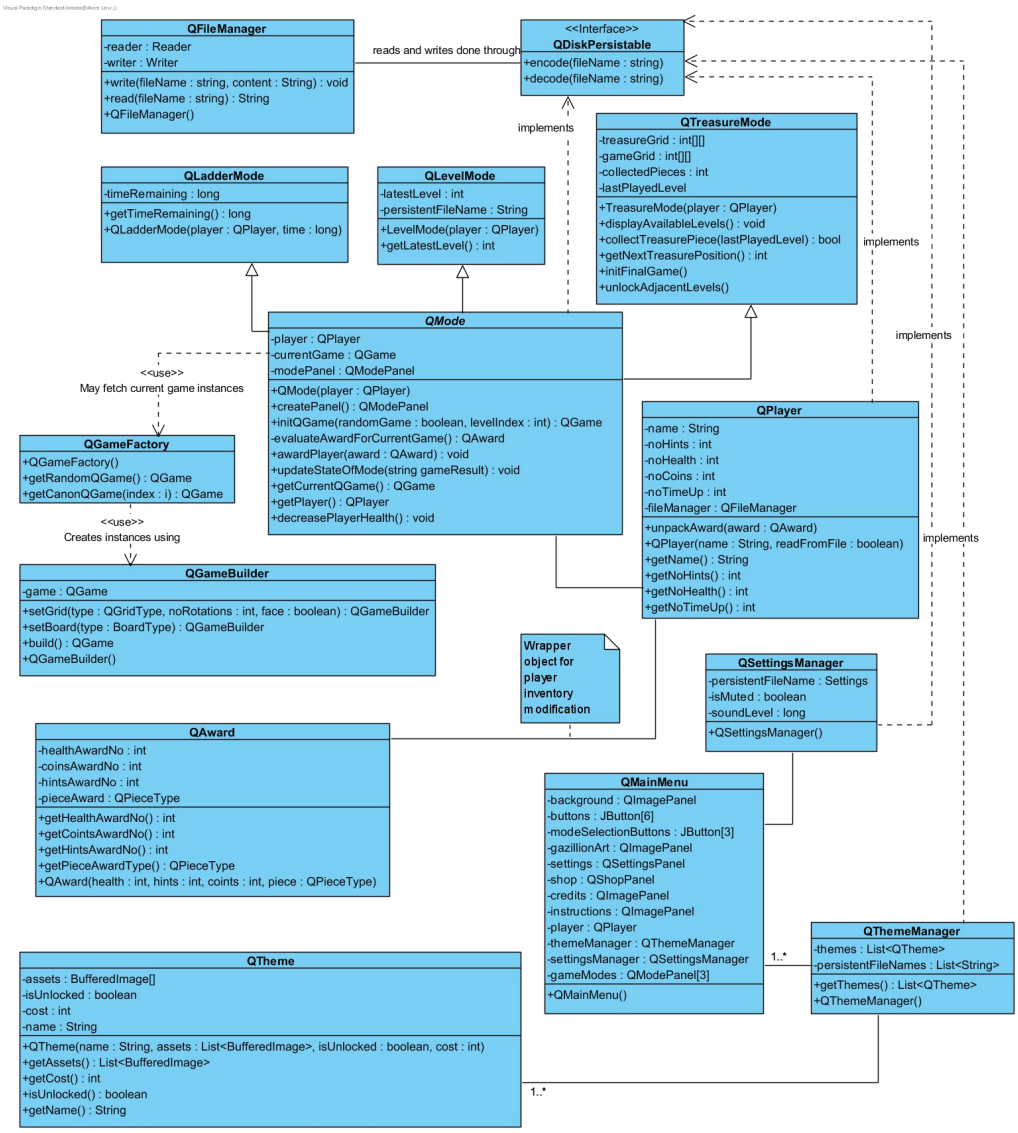
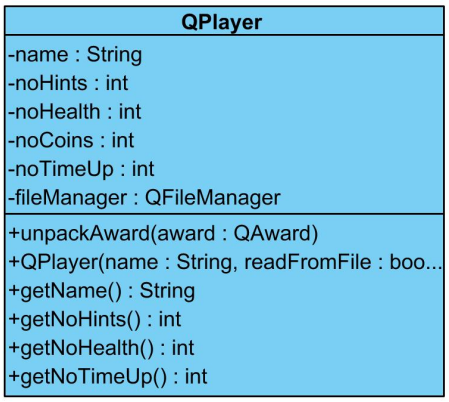


Figure 11. Classes for Game Management Subsystem

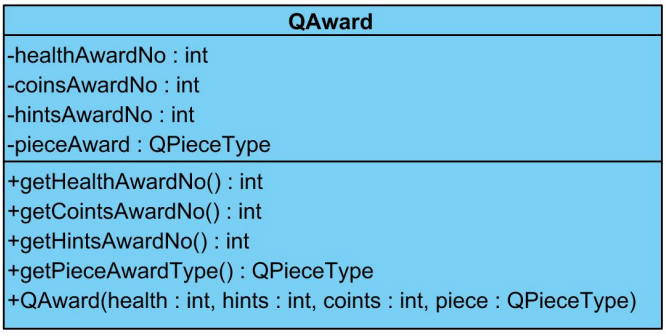
QPlayer



Defines a class representing the attributes of the player playing the game.

- private String name: Defines name for the player
- private int noHints: Keeps track of the number of the number of hint powerups.
- private int noCoins: Keeps track of the number of the number of coins.
- private int noTimeUp: Keeps track of the number of the number of time powerups.
- private int noHealth: Keeps track of the number of health powerups of the player. Note that this does not keep track of the health of the player, since health may not be present in all game modes.
- private QFileManager fileManager: Persists the player information to disk or reads the player information from disk on startup.
- public void unpackAward(Award award): Unwraps the award object passed to the method and changes the noHints, noCoins, noTimeUp, noHealth fields accordingly.
- public QPlayer(String name, boolean readFromFile): Constructs a QPlayer. Reads from the file if it is the first time the game is launched.
- public String getName(): Returns the name of the player.
- public int getNoHints(): Returns the noHints field.
- public int getNoCoins(): Returns the noCoins field.
- public int getNoTimeUp(): Returns the noTimeUp field.

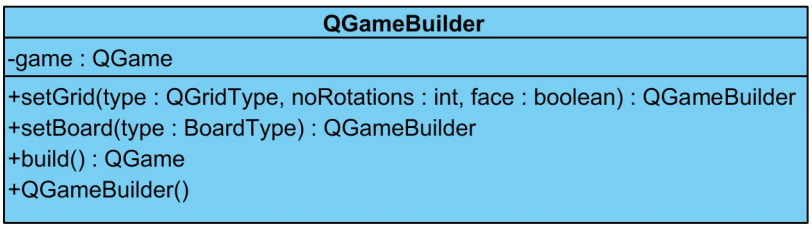
QAward



Defines a wrapper object for rewarding and punishing the player. Negative values for the field mean punishment for the player.

- private int healthAwardNo: Number of health power ups in the award.
- private int coinsAwardNo: Number of coins in the award.
- private int hintsAwardNo: Number of hint power ups in the award.
- private QPieceType pieceAward: Type of the piece awarded to the player.
- public int getHealthAwardNo(): Returns the healthAwardNo field.
- public int getCoinsAwardNo(): Returns the coinsAwardNo field.
- public int getHintsAwardNo(): Returns the hintsAwardNo field.
- public QPieceType getPieceAwardType(): Returns the type of the piece to be awarded.
- public QAward(int health, int hints, int piece, QPieceType piece): Constructs an award object with the given fields.

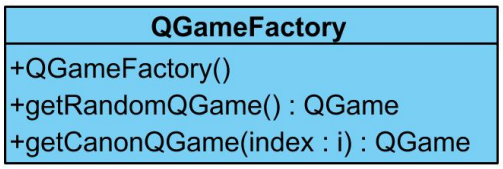
QGameBuilder



Defines a class using the builder pattern that might be used to construct a QGame.

- private QGame game: Reference to an incomplete QGame object.
- public QGameBuilder setGrid(QGridType type, int noRotations, Boolean face) : Sets one grid of the QGame object according to the parameters. Has to be called four times before the board is built.
- public QGameBuilder setBoard(QBoardType type): Sets the type of the board.
- public QGame build(): Finalizes and returns the QGame object.
- public QGameBuilder(): Constructs a builder object.

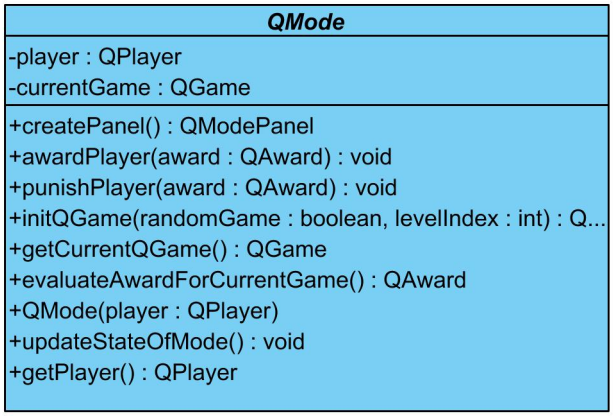
QGameFactory



Class that constructs QGame at a higher level than of the QGameBuilder class.

- public QGameFactory(): Returns an instance of this class.
- public QGame getRandomQGame(): Returns a random QGame.
- public QGame getCanonQGame(int index): Returns a QGame configuration as present in the Quadrillion game manual, indexed by the order of their appearance.

QMode



Abstract class that encapsulates and captures all necessary information for a game mode.

- private QPlayer player: Reference to the QPlayer object.
- private QGame currentGame: Reference to the current QGame object being played.
- public QModePanel createPanel(): Creates a corresponding QPanel for this mode. Enforces the mode implementation to have its own panel.
- public void awardPlayer(QAward award): Awards the player with the QAward object.

- public void punishPlayer(QAward award): Punishes the player with a QAward object.

- public QGame initQGame(boolean randomGame, int levelIndex): Initializes a QGame and returns a reference to the QGame constructed. Sets the current QGame instance.

- public QGame getCurrentGame(): Returns a reference to the current QGame instance.

- public QAward evaluateAwardForCurrentGame() : Evaluates award for current game and returns a reference to the corresponding QGame object.

- public void updateStateOfMode(): Evaluates and updates the state of the QMode object in accordance to the QGame instance.

- public QPlayer getPlayer(): Returns the QPlayer instance.

- public QMode(QPlayer player): Constructs a QMode instance given a QPlayer instance.

QLevelMode

A class that extends QMode and contains the implementation for the Level Mode. Also implements the QPersistable interface to store progress in a file persisted on the disk.

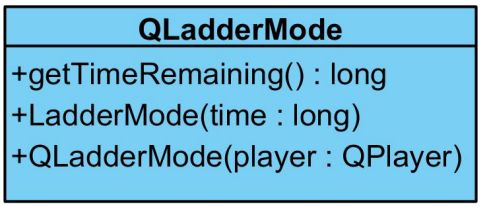
- private int latestLevel: Stores the latest level attained by the user.

- private static String persistentFileName: Persistent file name for the mode. This determines which file to look for, for information about the state.

- public int getLatestLevel(): Returns the index of the latest completed level.

- public LevelMode(QPlayer player): Constructs a LevelMode instance given a player instance.

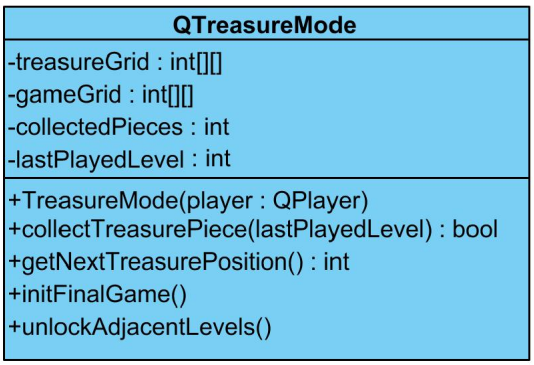
QLadderMode



A class that extends QMode and implements the Ladder Mode.

- private long timeRemaining: Stores current time remaining in the ladder mode.
- public long getTimeRemaining(): Returns the timeRemaining field of this class.
- public QLadderMode(QPlayer player): Constructs a LadderMode instance.

QTreasureMode



A class that extends QMode and contains the implementation for the Treasure Mode.

- private int[][] gameGrid: Stores information regarding the game grid via indexing a 2D matrix with predetermined values. I.e. 0 denotes a locked level, 1 denotes an unlocked level, 2 denotes a completed level.
- private int[][] treasureGrid: Stores information regarding the location of the 12 treasure pieces. 0 denotes an empty level, 1 denotes an acquired treasure, 2 denotes a revealed treasure that is not acquired yet.
- public int collectedPieces: stores the number of treasure pieces that the Player has collected so far.
- public int lastPlayedLevel: stores the number of the level that the Player played last, in order to unlock the adjacent levels in case the game was won.

- QTreasureMode(QPlayer player): Constructs a treasure mode instance with the supplied player instance.

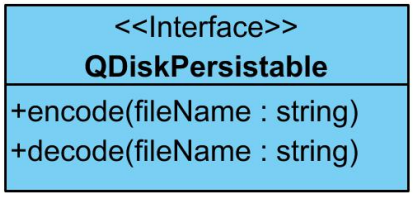
- public int collectTreasurePiece(lastPlayedLevel): after a game was won, checks if the lastPlayedLevel contains a treasure piece and awards it to Player if so.

- public int getNextTreasurePosition(): calculates the next level that contains an unrevealed and unacquired treasure and returns its index. Called when Player presses the Hint button in QTreasureModePanel.

- public void initFinalGame(): initiates the Final Game after Player has collected all the treasure pieces.

- public void unlockAdjacentLevels(): calls the appropriate methods of the QTreasureModePanel class to indicate the newly unlocked levels.

QDiskPersistable Interface

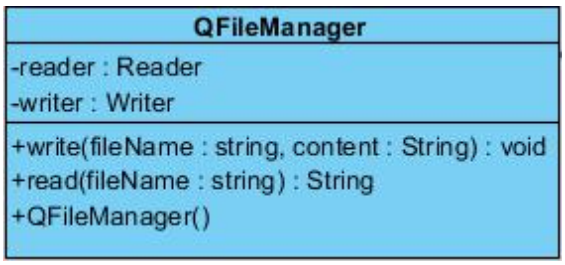


An interface that has to be implemented by classes that need to persist information to disk.

- public void encode(String fileName): The implementation should write a file to disk with an encoding decided by the application programmer.

- public void decode(String fileName): The implementation should read file from a disk with a decoding decided by the application programmer.

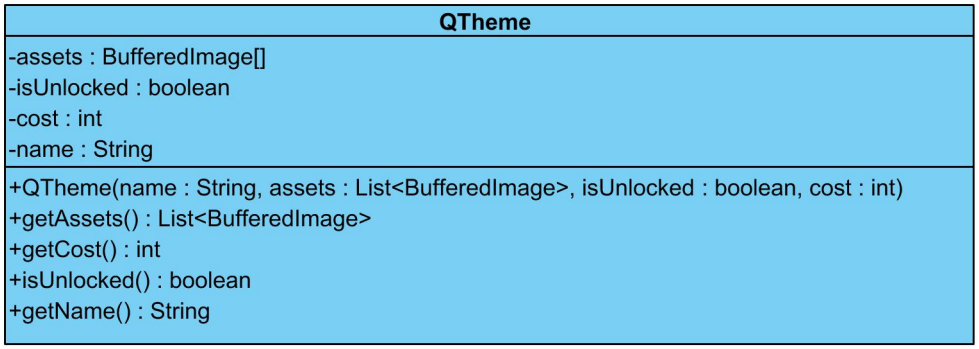
QFileManager



A class that can be used by classes that implement the QPersistable interface for disk I/O.

- private Reader reader: A reader for read(1) operation.
- private Writer writer: A writer for write(1) operation.
- public void write(String filename, String content): Writes content to the current directory with the specified file name.
- public String read(String fileName): Reads from the file and returns the contents read.
- public QFileManager(): Instantiates a QFileManager object.

QTheme



A class instance of a theme.

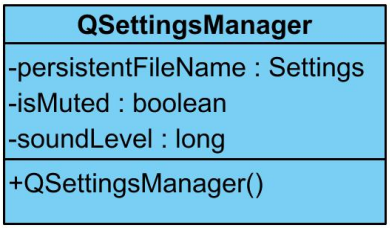
- private BufferedImage[] assets: array of image cropped in tiles read from file.
- private boolean isUnlocked: Checks whether the theme is unlocked or not.
- private int cost: The number of coins needed to buy the theme.
- private String name: The name of the theme.
- public QTheme(String name, List<BufferedImage> assets, boolean isUnlocked, int cost): Constructs an instance of this object and initializes its properties according to the parameters.
- public List<BufferedImage> getAssets(): reads the image from the file by putting all the tiles in a list.

- public int getCost(): returns the cost of the theme.

- public boolean isUnlocked(): returns whether a theme is unlocked or not.

- publicString getName(): returns the name of the theme.

QSettingsManager



This class manages the settings.

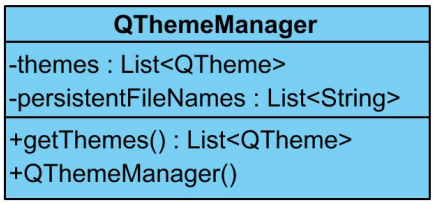
- private Settings persistentFileName: current settings configurations stored in a file

- private boolean isMuted: attribute that checks whether the sound is muted or not.

- private long soundLevel: the sound level stored in long integer value.

- public QSettingsManager: Constructs an instance of this object.

QThemeManager



This class manages the themes.

- private List<QTheme> themes: collection of all the available themes

- private List<String> persistentFileName: a list of the name of the files the images are stored.

- public QThemeManager(): Constructs an instance of this object.

- public List<QTheme> getThemes: Returns all the available themes.

QGazillionListener

Main controller class. Implements listeners for keys and mouse interactions. Implements the KeyListener, MouseListener, and the MouseMotionListener interfaces. Provides an adapter to these interfaces.

4.5.2 User Interface Subsystem

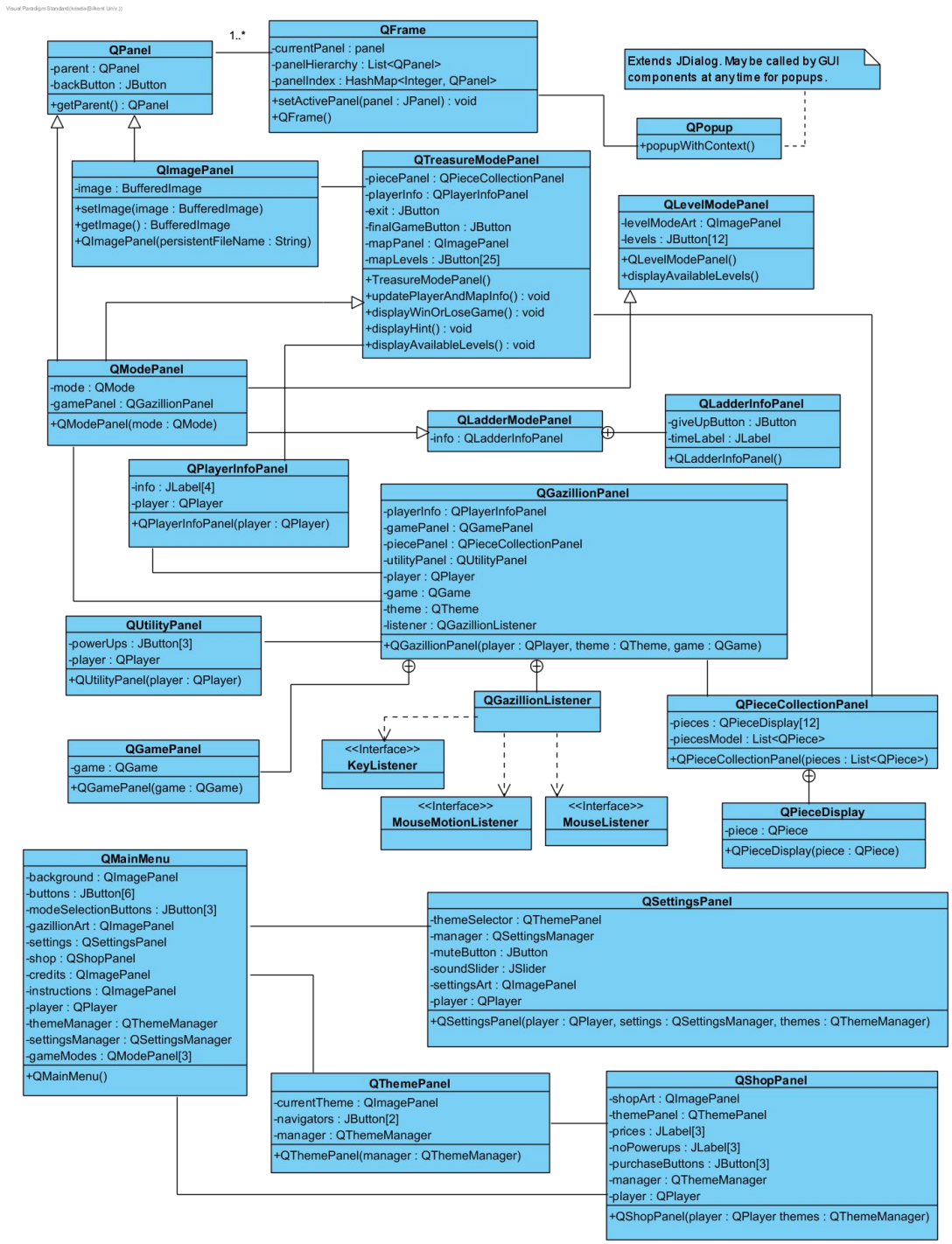
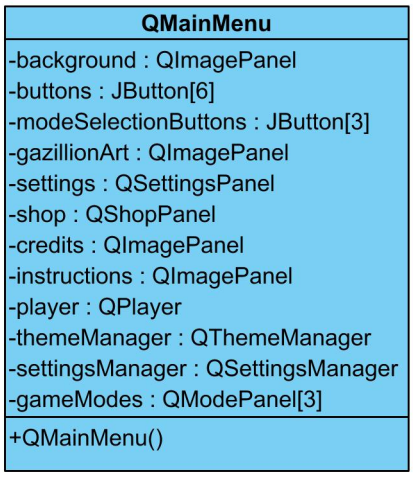


Figure 12. Classes for the User Interface Subsystem

QMainMenu



Provides the UI for the main menu.

- private QImagePanel background: The background image displayed on the main menu

- private JButton[6] buttons: 6 buttons to choose between the menu options: play, shop, instuctions, settings, credits, exit.

- private JButton[3] modeSelectionButtons: 3 buttons to choose between modes

- private QImagePanel gazillionArt: A panel that contains an image with ‘Gazillion’ written on it in an aesthetic font.

- private QSettingsPanel settings: The panel containing settings options.

- private QShopPanel shop: The panel containing shop options.

- private QImagePanel credits: The panel with the credits contained in an image

- private QImagePanel instructions: The panel with instructions contained in an image.

- private QPlayer player: An instance to the player.

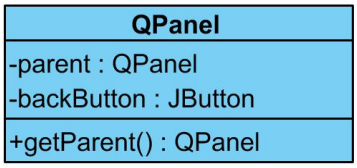
- private QThemeManager themeManager: An instance to the theme manager.

- private QSettingsManager settingsManager: An instance to the settings manager.

- private QModePanel[3] gameModes: Instances of the three game mode panels. According to the player's choice one of them is called at a time.

- public QMainMenu: Constructs an instance of the Main Menu panel.

QPanel



Custom panel for the game UI. Extends JPanel.

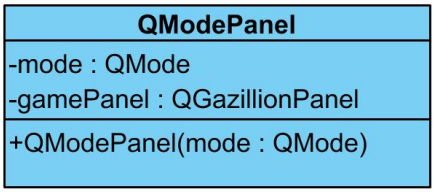
- private QPanel parent: Stores a reference to the parent QPanel that spawned this QPanel.

- private JButton backButton: A button that signals to the main frame to set the current panel to the parent panel.

- public QPanel getParent(): Returns the parent panel.

- public QPanel(QPanel parent): Constructs a QPanel. Parent is null if it is the initial panel.

QModePanel



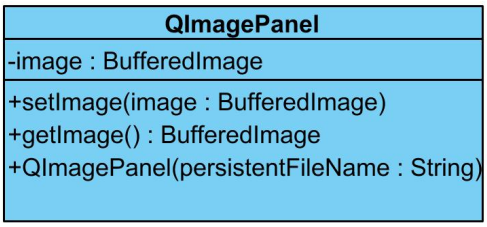
Class extending the QPanel class. Should implement the UI for a given game mode.

- private QMode mode: A reference to the mode of the panel.

- private QGazillionPanel gamePanel: A reference to the panel on which the QGame is drawn.

- public QModePanel(QMode mode): Constructs a QModePanel instance with the specified mode. QMode should call this method with itself as the parameter in its createPanel() method.

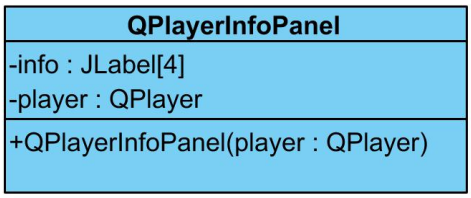
QImagePanel



Class that extends QPanel and implements QPersistable. It provides a wrapper for embedding images in the panel.

- private BufferedImage image: Image to be displayed in the panel.
- public void setImage(BufferedImage image): Sets the image of this panel.
- public BufferedImage getImage(): Returns the image field of this object.
- public QImagePanel(String persistentFileName): Creates an instance of this object by reading an image from a file.

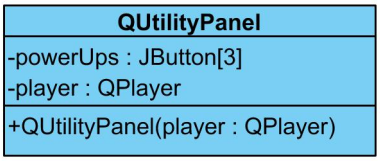
QPlayerInfoPanel



Class that displays the player information on labels. The labels display health, coins, current level, and player name.

- private JLabel[] info: Labels for information display.
- private QPlayer player: Reference to the QPlayer object.
- public QPlayerInfoPanel(QPlayer player): Constructs a player info panel for the player instance.

QUtilityPanel



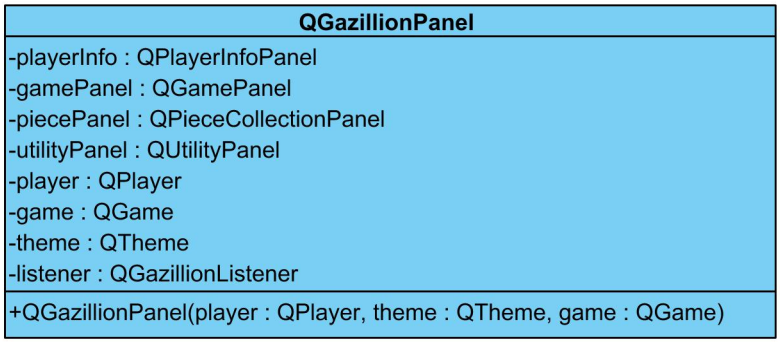
Class that allows the player to use powerups.

- private JButton[] powerUps: Buttons to use powerups. Buttons 1, 2, 3 correspond to health, time, and hints. Also displays the time left for the player through the text of the button.

- private QPlayer player: Reference to player object.

- public QUtilityPanel(QPlayer player): Constructs an instance of this class.

QGazillionPanel



Class that draws a full game of Gazillion.

- private QPlayerInfoPanel playerInfo: Panel for player info.

- private QGamePanel gamePanel: Panel for drawing the game board.

- private QPieceCollectionPanel piecePanel: Panel for displaying and interacting with the game pieces.

- private QUtilityPanel utilityPanel: Panel for using powerups and seeing the time remaining.

- private QPlayer player: Reference to the player.

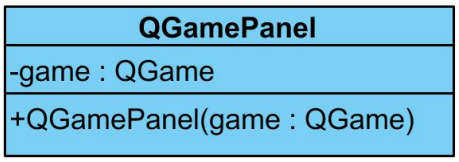
- private QGame game: Reference to the QGame object, which will be drawn.

- private QTheme theme: Reference to the QTheme object. This object determines how the board is drawn on the panel.

- private QGazillionListener listener: Listener for the Panel.

- public QGazillionPlayer(QPlayer player, QTheme theme, QGame, game): Constructs an instance of the panel with the given parameters.

QGamePanel



Inner class of QGazillionPanel. Draws the QGame instance (the game board). Overrides paintComponent(1).

- public QGamePanel(QGame game): Constructs an instance of this class

QPieceCollectionPanel

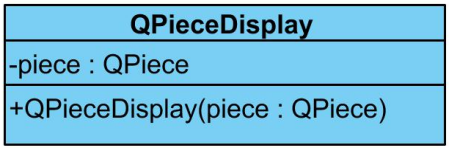
A panel that draws twelve game pieces as present in the game. Allows interaction with the piece through the QGazillionListener class.

- private QPieceDisplay[] pieces: Panels on which individuals pieces will be drawn.

- private List<QPiece> piecesModel: List logical representation of the pieces present in the game.

- public List<QPieces> QPieceCollectionPanel(List<QPieces> pieces): Constructs an instance of this object.

QPieceDisplay

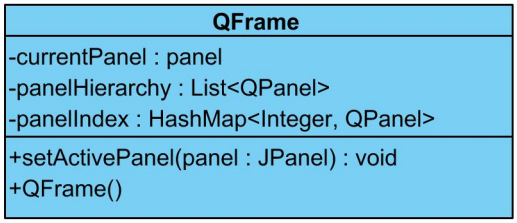


Draws the current piece in its current state, on a neat grid. Inner class of QPieceCollectionPanel.

- private QPiece piece: Reference to the QPiece to be drawn on the panel.

- public QPieceDisplay(QPiece piece): Constructs an instance of this panel.

QFrame



Class defining a hierarchy of panels, one of which at a time is drawn on the frame. Extends JFrame. Swaps panels when necessary.

- private QPanel currentPanel: Stores a reference to the current panel.

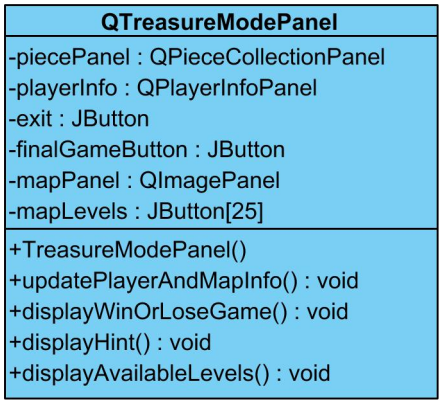
- private List<QPanel> panelHierarchy: Stores the panels available for display. This is a hierarchy since all QPanels store references to their parent.

- private HashMap<Integer, QPanel> panelIndex: Stores an index for the panels for quick access.

- public void setActivePanel(QPanel panel): Sets the active panel of the QFrame. The panel is added to the panel hierarchy and indexed.

- public QFrame(): Constructs an instance of this class.

QTreasureModePanel



Class that provides UI for the treasure mode.

- private QPieceCollectionPanel piecePanel: Draws the pieces the player has collected thus far.

- private QPlayerInfoPanel playerInfo: A panel that displays the player information as discussed in the QPlayerInfoPanel class.

- private JButton exit: JButton that finalizes the treasure mode and exits the mode.

- private JButton finalGameButton: Button that allows the player to play a final game of Quadrillion upon collection of all pieces.

- private QImagePanel mapPanel: A panel which displays the map art.

- private JButton[] mapLevels: An array of 25 buttons, each representing a node in the map in a 5x5 grid.

- public QTreasureModePanel(QTreasureMode mode): Instantiates an instance of the class given a treasure mode.

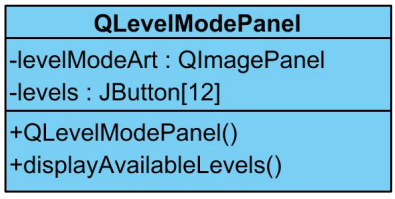
- public displayAvailableLevels(): Displays unlocked levels that the Player can select and play.

- public updatePlayerAndMapInfo(): updates player health and coins and the state of the levels on the map as played and available, after a game has been played.

- public displayWinOrLoseGame(): redirects to the appropriate window after the final game has been played, to show the outcome of the game; and if player health decreases to 0 and the game is over.

- public displayHint(): if Player has enough Hints in his inventory, an unplayed level node which holds a treasure is indicated (the node changes color).

QLevelModePanel



Implements the UI for the level mode.

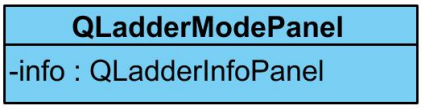
- private QImagePanel levelModeArt: A panel that displays the picture are for level mode.

- private JButton[] levels: JButtons which have images of the canonical levels that one can choose. They are arranged in a 3x4 grid.

- public QLevelModePanel(QLevelMode mode): Constructs an instance of this object.

- public displayAvailableLevels(): Displays unlocked levels that the Player can select and play.

QLadderModePanel

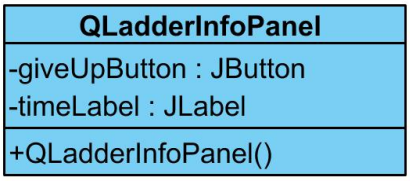


Provides the UI for the ladder mode.

- private QLadderInfoPanel info: Displays information about the state of the player in ladder mode.

- public QLadderModePanel(QLadderMode mode): Constructs an instance of this object.

QLadderInfoPanel



Provides a panel to display the information regarding the state of the ladder mode.

- private JButton giveUpButton: Button to give up.

- private JLabel timeLabel: Displays the remaining time for the player.

- public QLadderInfoPanel(): Constructs an instance of this object.

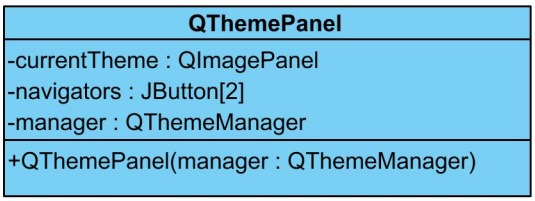
QPopup

A class that defines popup panels to be used for alerting the player. May be instantiated and used anywhere necessary. Extends JDialog.

- public void popupWithContext(): Displays a popup message with the appropriate context.

- public QPopup(Context context): Constructs a popup message with a given context.

QThemePanel



Defines a panel to display a preview of the theme that will be used to render the game.

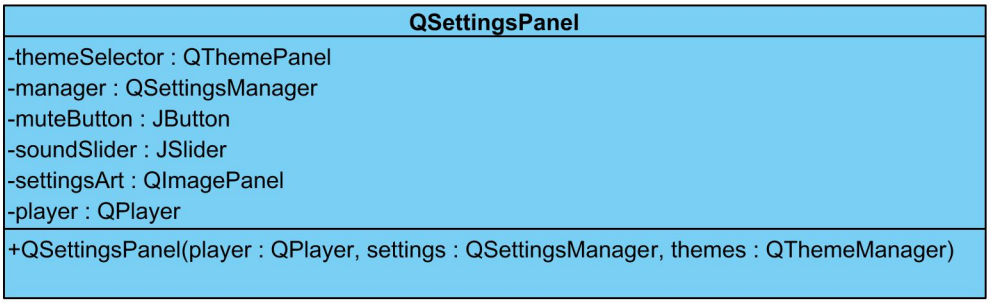
- private QImagePanel currentTheme: An image panel that has the preview picture of the theme.

- private JButton[] navigators: An array of two buttons. One for next theme, one for the previous theme.

- private QThemeManager manager: Reference to the theme manager.

- public QThemePanel(QThemeManager manager): Constructs an instance of this object.

QSettingsPanel



Defines a panel from which several settings of the game may be adjusted.

- private QThemePanel themeSelector: Panel to choose and display the current theme.

- private QSettingsManager manager: Reference to the QSettingsManager to alter the settings.

- private JButton muteButton: Button to mute the game.

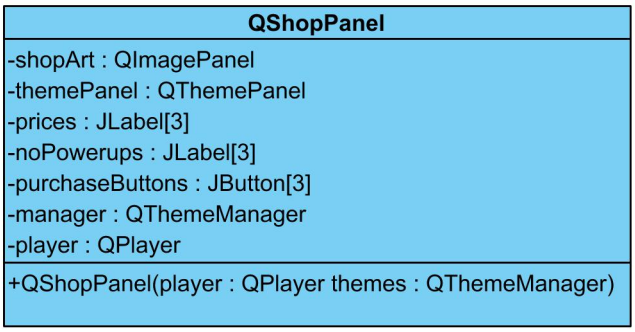
- private JSlider soundSlider: JSlider to adjust the sound level.

- private QImagePanel settingsArt: Panel to display the “Settings” picture.

- private QPlayer player: Reference to the player.

- public QSettingsPanel(QPlayer player, QSettingsManager settings, QThemeManager themes): Constructs an instance of this object.

QShopPanel



Defines a panel from which the player may buy themes and powerups with coins.

- private QImagePanel shopArt: QImagePanel to display a picture that says “shop”

- private QThemePanel themePanel: A panel to display the theme that will be purchased.

- private JLabel[] prices: Labels on which prices of the items are written.

- private JLabel[] noPowerups: Labels on which the number of player powerups are written.

- private JButton[] purchaseButtons: Buttons which allow the purchasing of powerups.

- private QThemeManager manager: Reference to the theme manager.

- private QPlayer player: Reference to the player.

- public QShopPanel(QPlayer player, QThemeManager themes): Constructs an instance of this object.

4.5.3 Game Logic Subsystem

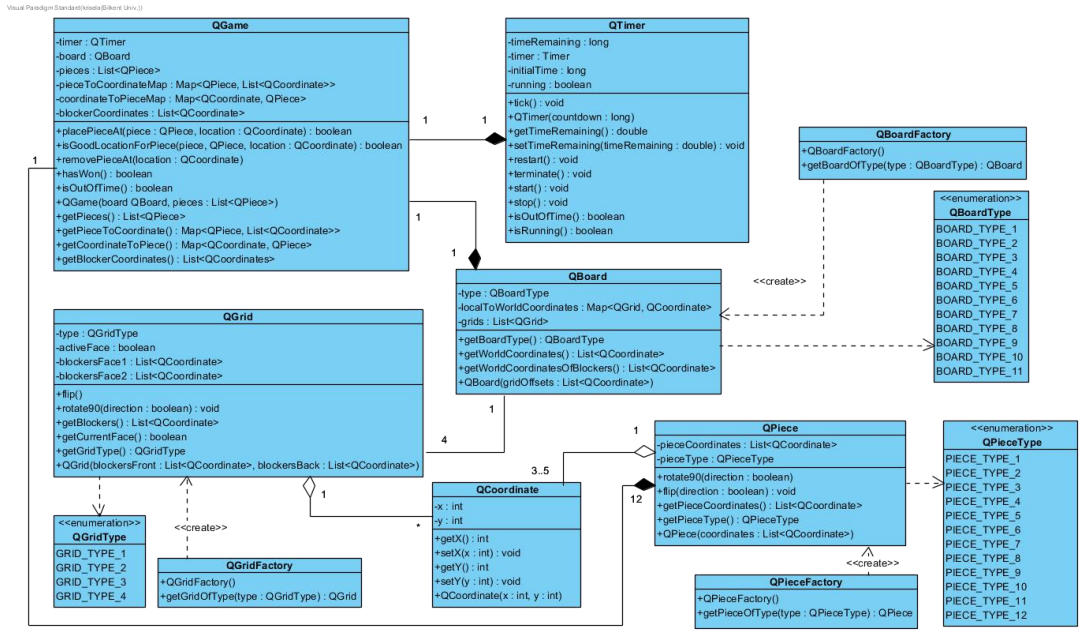
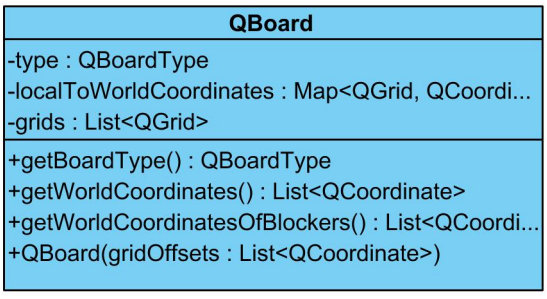


Figure 13. Classes for the Game Logic Subsystem

QBoard



Defines a board class, composed of four QGrid objects, and maps the necessary geometric transformations from the local coordinate system of the grids to the "world" coordinate system of the board.

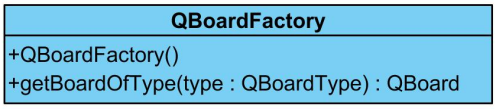
- public QBoard(List<QGrid> grids, List<QCoordinate> offsets, QBoardType type) : Constructs an instance of the object.

- public QBoardType getBoardType(): Returns the type of the board.

- public List<QCoordinate> getWorldCoordinateOfBlockers(): Returns the coordinates of the blocking coordinates in the coordinate system of the board.

- public List<QCoordinate> getWorldCoordinates(): Returns the explicit list of coordinates of the board.

QBoardFactory

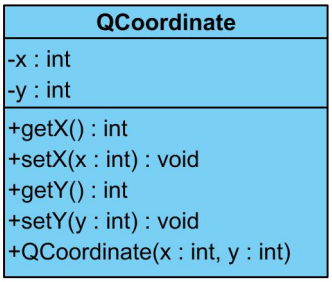


Defines a factory class for board construction. As of this version, it is empty, because the developers do not have a physical copy of the game.

- public QBoardFactory(): Constructs an instance of this object.

- public QBoard getBoardOfType(QBoardType type): Returns a board of the specified type.

QCoordinate



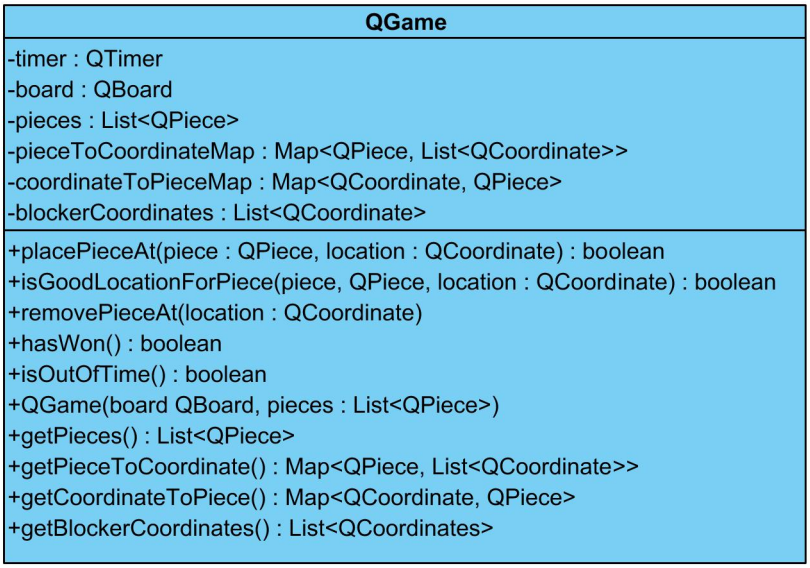
Defines a simple integer two-tuple for the representation of 2D coordinates, for modelling purposes of Quadrillion.

- public boolean equals(java.lang.Object o): Provides an equality check for two-tuples.

- public void set(int x, int y): Sets both the x and y coordinate of the two-tuple to desired values.

- public void setX(int x): Sets the x coordinate of the two-tuple to a desired value.
- public void setY(int y): Sets the y coordinate of the two-tuple to a desired value.
- public int x(): Returns the x coordinate of the two-tuple.
- public int y(): Returns the y coordinate of the two-tuple.

QGame



Defines a game of Quadrillion in the style of Gazillion, using QPiece objects, a QBoard object, a QTimer object, and provides the necessary functionality for piece placement and removal to and from the board.

- public QGame(QBoard board, long timeRemaining): Constructor for the game object.
- public boolean hasWon(): Checks if the game has been won or not.
- public void incrementTimeRemaining(long increment): Increments the time remaining by a specified amount.
- public boolean isOutOfTime(): Returns if the user has run out of time.
- public boolean placePieceAt(QPiece piece, QCoordinate location): Places a piece at a specified location on the board, such that the (0,0) coordinate of the piece fits that particular location.

- public boolean removePieceAt(QCoordinate location): Removes a piece occupying a particular coordinate on the game board.

- public void startTimer(): Starts the timer object.

- public void stopTimer(): Stops the timer object.

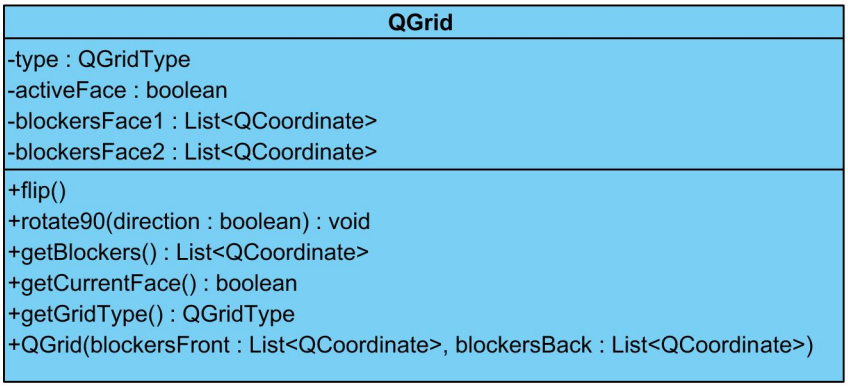
- public List<QPiece> getPieces(): Returns the pieces in the game.

- public Map<QPiece, List<QCoordinate>> getPieceToCoordinate(): Returns the piece to coordinate map.

- public Map<QCoordinate, QPiece> getCoordinateToPiece(): Returns the coordinate to piece mapping.

- public List<QCoordinate> getBlockerCoordinates(): Returns the blocking coordinates.

QGrid



Defines a Grid class, which implicitly has a bounding box of (0,0) to (3,-3), i.e. composed of a 4x4 QCoordinate grid whose top left corner is situated at (0,0), and bottom right corner is situated at (3,-3). Only the blocking tiles are explicitly stored. Has two faces.

- public void flip(): Flips the face of the grid to the other side.

- public List<QCoordinate> getBlockers(): Returns the coordinates of the blocking tiles of the active face of the grid.

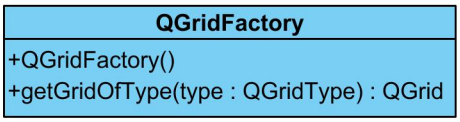
- public boolean getCurrentFace(): Returns the current face of the grid.

- public QGridType getGridType(): Returns the grid type of the grid, one among four grids as present in the original game.

- public void rotate90(boolean direction): Rotates the grid 90 degrees clockwise or counter-clockwise.

- public Qgrid(List<QCoordinate> blockersFront, List<QCoordinate> blockersBack, QGridType type): Constructs a grid given the blocking tiles on each face.

QGridFactory

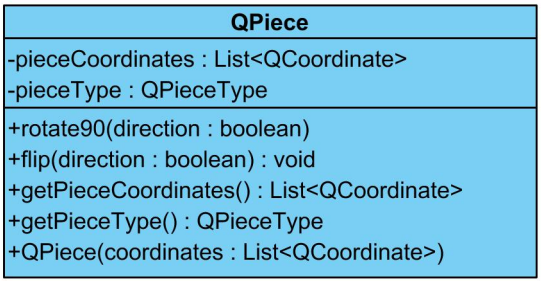


Defines a factory class for grid construction. As of this version, it is empty, because the developers do not have a physical copy of the game.

- public QGridFactory(): Constructs an instance of this object.

- public QGrid getGridOfType(QGridType type): Constructs a grid given its type.

QPiece



Defines a piece by storing the coordinates occupied by the piece explicitly in its local coordinate system. Provides functionality for rotation and the flipping of the pieces.

- public void flip(boolean direction): Flips the piece along the x or y axis depending on the parameter supplied.

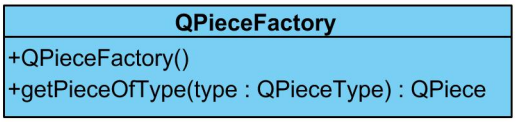
- public List<QCoordinate> getPieceCoordinates(): Returns a list of the coordinates which the piece occupies, in local coordinates.

- public QPieceType getPieceType(): Returns the type of the piece as present in the original game, one among the 12 types.

- public void rotate90(boolean direction): Rotates the piece 90 degrees, clockwise or counter-clockwise according to the parameter supplied.

- public QPiece(List<QCoordinate> coords, QPieceType type): Constructs a QPiece instance given its coordinates and piece type.

QPieceFactory

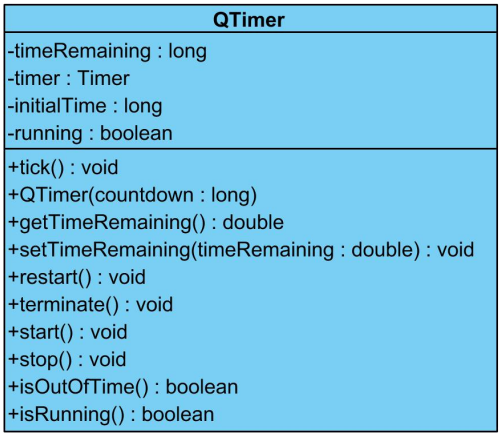


Defines a factory class for pieces as present in the original Quadrillion game.

- public QPiece getPieceOfType(QPieceType type): The type of the QPiece to be constructed.

- public QPieceFactory(): Constructs an instance of this class.

QTimer



Defines a timer for the game that runs on a separate thread. Provides functionality for starting and stopping the timer.

- public QTimer(long timeRemaining): Constructs a QTimer instance.

- public long getTimeRemaining(): Returns the remaining time in milliseconds.

- public boolean isOutOfTime(): Returns if time has run out.

- public boolean isRunning(): Returns the running state of the timer.

- public void restart(): Restarts the timer.

- public void setTimeRemaining(long timeRemaining): Sets the remaining time to a desired value.
- public void start(): Starts the timer.
- public void stop(): Stops the timer.
- public void terminate(): Terminates the timer.
- public void tick(): Ticks the timer by 100 milliseconds.

QPieceFactory

Defines a factory class for pieces as present in the original Quadrillion game.

public QPiece getPieceOfType(QPieceType type): The type of the QPiece to be constructed.

public QPieceFactory(): Constructs an instance of this class.

4.6 Class Interface Abstraction

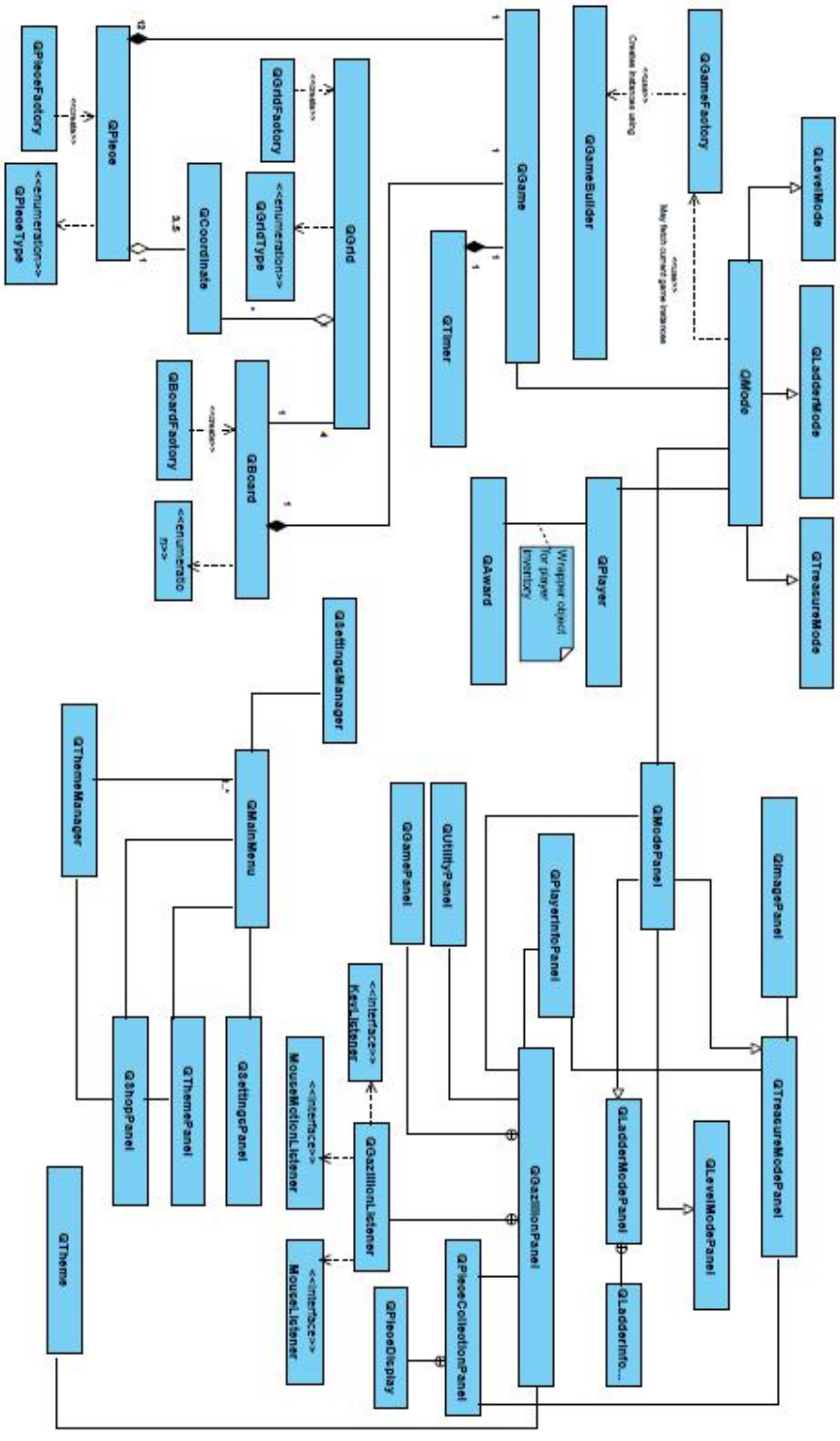


Figure 14. Abstract Class Diagram