Table of Contents

[MVC Pattern 1](#_Toc151215684)

[Model Classes 1](#_Toc151215685)

[View Classes 1](#_Toc151215686)

[Controller Classes 2](#_Toc151215687)

[Additional Considerations 2](#_Toc151215688)

[Utility Classes 2](#_Toc151215689)

[Qt Specific Classes and Features 3](#_Toc151215690)

[Implementation Considerations 3](#_Toc151215691)

[QtMainWindow Classes Construction 3](#_Toc151215692)

# **MVC Pattern**

Based on a typical game project that adheres to the Model-View-Controller (MVC) pattern, and considering common subtasks that might be mentioned in a final project for advanced programming techniques, here's a general idea of what classes you might need for each component of the MVC pattern:

## **Model Classes**

The Model classes represent the data and the business logic of your application. They are responsible for managing the game's state, rules, and behaviors.

**World:** Manages the overall game world, including dimensions, tiles, and possibly a collection of all game objects.

Tile: Represents individual tiles within the world, which could include information like terrain type, movement cost, or whether it's occupied.

**Tiles:** provided by library code.

**Enemy:** is a library class provided by the Professor. Derived class from Tile. Represents adversaries within the game, which could include stats like health, position, and behavior patterns.

**PEnemy**: is a library class provided by the Professor. Derived class from Tile. Represents the poison enemy.

**Protagonist:** is a library class provided by the Professor. Derived class from Tile. Represents the player character with stats like health, position, and inventory.

**Entity:** The abstract and parent class of EnemyModel and ProtagonistMdoel

**EnemyModel:** is a class that aggregates Enemy class and can have additional methods than Enemy for combats and future extendibility. Special Enemies such as PEnemyModel that could poison and XEnemyModel that could do custom attack can derive from this model class.

**PEnemyModel:** PEnemy is an enemy which is poisoned. It will lose its poison when attacked and finally it will die

**XEnemyModel:** A customizable class for an additional type of enemy with unique behaviors.

**ProtagonistModel:** is a class that aggregates Protagonist class and can have additional methods than Protagonist for combats and future extendibility.

**Healthpacks:** It is a special class derived from Tile class as each healthpack can only be on a tile having the same functions but performs different actions (overriding functions) and might have additional methods for the protagonist model to use.

**PTiles:** is a derived class of Tile that is able to poison both the enemy model and the protagonist model. This derived PTile would be a specialized type of Tile that has additional attributes and behavior related to being poisoned—such as affecting the health of the protagonist or an enemy when they move over it.

**Pathfinder:** Contains algorithms for pathfinding, such as A\* or Dijkstra's algorithm.

## **View Classes**

View classes are responsible for presenting the game state to the user in a visual form.

**MainWindow (Derived from QMainWindow):** This will be your primary window class, managing the main application window.

**ViewControllerFactory:** ViewControllerFactory is a factory that creates ViewController instances, which are also part of the view layer. Since WindowController is already responsible for managing UI components, it would be logical for it to also manage the creation of these components via the factory.The factory is usually used by a part of the application responsible for initializing the system or responding to a change in state that requires a new type of view controller to be created, such as the WindowController.

**GameView(abstract class)**

GameView interface or abstract class in C++ using pure virtual functions. This abstract class would define a common interface for different types of game views. GameViewText, GameView2D, and GameOverlay would then inherit from this abstract class and implement its virtual functions.

**Game2DView (extends QGraphicsView):**

A view that provides a 2D graphical representation of the game world using the QGraphicsScene and QGraphicsItem classes. Inherits from GameView and uses QGraphicsView and QGraphicsScene for rendering the 2D representation of the game. It would manage a scene containing QGraphicsItem subclasses like QGraphicsRectItem or QGraphicsPixmapItem.

**GameTextView (extends QWidget or QTextEdit):**

A view that provides a text-based representation of the game world, which could be implemented as a grid of characters or a series of descriptive texts. Also inherits from GameView. This class would handle the textual representation of the game, potentially using Qt’s text rendering classes like QTextEdit or QTextBrowser.

**GameOverlayView:** a different image as a top layer with all tiles “behind” it as a data layer. For example, a map of a medieval town with some roads, houses and gardens with stone walls surrounding them. The data layer underneath this could have tiles with low values for the roads and infinite values for the stone walls, this way a visualization can be different from the data layer. Entering the door of a house loads a new map. There could be potholes or walking through several types of roads could be more tiresome (muddy roads). Think of a way to incorporate this into your design, you should have only one image for the complete map of one level. When zooming in into such a map the correlation between the top layer and the data layer should be maintained, your solution should be able to handle this. This class would also inherit from GameView and implement their specific rendering mechanisms.

**Future Views (3DView, VRView):** These classes would also inherit from AbstractView and implement their specific rendering mechanisms, such as using QOpenGLWidget for 3D rendering or specialized libraries for VR.

**EntityGraphicsItem (extends QGraphicsItem):**

For the graphical view, a class that represents the visual aspect of dynamic entities like the protagonist and enemies.

**EntityTextItem**

For the textual view, a class that represents the text-based aspect of dynamic entities like the protagonist and enemies.

**TileGraphicsItem (extends QGraphicsItem): inlcuded in Tile.h and Tile.cpp**

For the graphical view, a class that represents the visual aspect of each tile.

## **Controller Classes**

Controller classes handle user input and translate it into actions in the model and updates to the view.

**WindowController:** Responsibilities of WindowContorller is to handle user input from MainWindow class and window-level events, delegate user actions to GameController. It has a direct association with GameController to relay user interactions.

**GameController:**

To handle the logic and rules of the game and To integrate the model (World, Tiles, Enemies) with the **view** (Graphical and Text-based representations). This remains necessary to handle user input and game logic, translating them into actions within the model. It will likely need to interact with ProtagonistModel, EnemyModel, PEnemyModel and XEnemyModel instances through their public methods and signals. This class will serve as the central point of game logic control, processing user inputs and managing the game state.

**WorldController:** The WorldController should handle interactions within the world, such as moving entities, resolving combat, and managing the state of the world.

**CombatController:** Since the EnemyModel or the ProtagonistModel class has a signal dead that is emitted when defeated, a controller class that manages combat scenarios is likely needed. This class would handle combat logic, such as determining if an attack hits and managing the health of the protagonist and enemies.

**ViewController:** This class would handle the switching between the graphical view (TileGraphicsItem) and a potential text-based view. It would manage which view is currently active and update the views when the model changes.

## **Additional Considerations**

**Signal-Slot Mechanism: Utilizing Qt's signal-slot mechanism for event-driven programming. This involves defining signals and slots in your classes to handle asynchronous events like user actions.** Qt's Signal & Slot: Leverage Qt's signal and slot mechanism for communication between different parts of your application, especially between the view and the controller.

## **Utility Classes**

**File Management:** For handling file read/write operations, especially if your application involves saving/loading data.

**Validation Classes:** To validate user input or game state changes.

**Custom Signal and Slot Classes:** For managing custom signals and slots that extend beyond Qt's default capabilities.

**Helper Classes:** These would include classes for common tasks and utilities used across the application.

## **Qt Specific Classes and Features**

**QGraphicsItem Classes:** For rendering graphical components.

**QPixmap for Image Handling:** If your application includes image rendering.

**QAction and QMenu Classes:** For handling actions and menus within the application.

## **Implementation Considerations**

**Integration with Qt:** Ensure that these classes are well integrated with Qt's event handling and rendering systems.

**Signal & Slot Mechanism:** Extensively use Qt's signal and slot mechanism for effective event handling and communication between different parts of your application.

**MVC Adherence:** Make sure that each class strictly adheres to its role within the MVC pattern to maintain a clear separation of concerns.

**Enemy, and Protagonist Model**

Certainly! Here's an example of what the attributes and methods might look like for the World, Enemy, and Protagonist classes in a game application that follows the MVC pattern:

##### **Enemy Model**

Enemy represents opponents in the game that the protagonist might interact with or fight against.

###### **Attributes**

* position: The current position of the enemy on the world grid.
* health: The health points of the enemy.
* strength: The attack power or strength of the enemy.
* speed: How fast the enemy can move or how often it can act.

###### **Methods**

* move(): Logic for moving the enemy on the world grid.
* attack(target): Logic for the enemy to attack another object.
* takeDamage(amount): Reduce the enemy's health by the damage amount.
* isDefeated(): Check if the enemy's health is depleted.
* updateState(): Update the enemy's internal state, potentially including AI behavior.

##### **Protagonist Model**

Protagonist represents the player character within the game world.

###### **Attributes:**

* position: The current position of the protagonist on the world grid.
* health: The health points of the protagonist.
* energy: An additional attribute representing stamina or mana.
* inventory: A collection of Item objects that the protagonist is carrying.

###### **Methods:**

* move(direction): Logic for moving the protagonist in the world.
* attack(): Logic for the protagonist to attack, potentially affecting an Enemy.
* useItem(Item): Use an item from the inventory.
* takeDamage(amount): Reduce the protagonist's health by the damage amount.
* isAlive(): Check if the protagonist is still alive.
* updateState(): Update the protagonist's state, such as applying ongoing effects.

These classes could be fleshed out with more attributes and methods depending on the complexity of your game. For example, the World class might need methods for serialization to save and load game state, and Enemy and Protagonist classes might have methods for specific abilities or status effects.

# **QtMainWindow Classes Construction**

Each application runs in a window, typically represented by an instance of the QMainWindow class. When making a new Qt Widgets application, a subclass of QMainwindow is automatically added to the project, and initialized in the main.cpp file

Inside this window you can add one or more QGraphicsView objects. These are essentially ‘windows into your scene’. For example, you could have a huge scene of 1000x1000 tiles, of which your QGraphicsView would only show a small fraction

Inside this view you can add a number of things, for this project one of the most interesting options is a QGraphicsScene object. This is a 2D canvas that can contain thousands of 2D objects. This would be what you add your tiles, enemies etc to

Inside the scene you can add objects. You can add anything, as long as it is a subclass of QGraphicsItem. There’s a number of predefined subclasses, such as QGraphicsRectItem (a rectangle) and QGraphicsPixmapItem (an image). If you want, you can of course also create your own subclasses

# **UML diagram**

**View**

* **ViewController - GameView:**
  + **Inheritance:** GameView is an interface that ViewController implements.
  + Each ViewController is responsible for a specific GameView.
* **GraphicsViewController – Game2DView:**
  + **Inheritance**: GraphicsViewController is a specific type of ViewController that manages a GraphicsView.
  + GraphicsViewController controls how the graphical elements are presented and interacted with.
* **TextViewController - TextView:**
  + **Inheritance**: TextViewController is a specific type of ViewController that manages a TextView.
  + TextViewController controls how text elements are presented and interacted with.
* **ViewControllerFactory - ViewController:**
  + **Dependency**: ViewControllerFactory creates instances of ViewController.
  + When a new view controller is needed, the factory is called upon to provide it.
* **WindowController - ViewControllerFactory:**
  + **Association**: WindowController has a ViewControllerFactory.
  + WindowController uses the factory to create and switch between view controllers.
* **WindowController - MainWindow:**
  + **Association**: WindowController has a MainWindow (or the otherway around?).
  + WindowController to handle userinput from MainWindow.

**Controller**

* **WindowController - GameController:**
  + **Association**: WindowController has a GameController.
  + WindowController may direct the GameController to update the combat status, indirectly update view controlled by GameController, or handle user input.
* **GameController - ViewController:**
  + **Aggregation**: GameController holds a collection of ViewController objects.
  + GameController may switch between different ViewController instances based on game state.
* **GameController - WorldController:**
  + **Association**: GameController has a WorldController.
  + GameController calls methods on WorldController to manipulate the game world.
* **WorldController - Entity (ProtagonistModel/EnemyModel):**
  + **Aggregation**: WorldController manages multiple Entity objects, including Protagonist and Enemy.
  + WorldController is responsible for updating and managing these entities within the game world.

**Model**

* **EntityGraphicsItem - Entity:**
  + **Dependency**: EntityGraphicsItem depends on Entity for its data.
  + EntityGraphicsItem is the graphical representation of an Entity.
* **EntityTextItem - Entity**
  + **Dependency**: EntityTexticsItem depends on Entity for its data.
  + EntityTexticsItem is the textual representation of an Entity.
* **ProtagonistModel/EnemyModel – Entity:**
  + **Inheritance**: Protagonist and Enemy are specific types of Entity.
  + They extend Entity with additional properties and behaviors specific to protagonists and enemies.
* Please note that in UML, **inheritance** is typically represented by a line with a **closed** **arrowhead**, **aggregation** by a line with a **hollow** **diamond** **at the aggregate (containing)** end, **association** by a **plain line**, and **dependency** by a **dashed line**.

# **Model Class Construction**

* **Classes**
  + **Attributes**
  + **Constructor**
  + **DeConstructor**
  + **Methods**

# **View Class Construction**

* **Classes**
  + **Attributes**
  + **Constructor**
  + **DeConstructor**
  + **Methods**

# **Controller Class Construction**

* **Classes**
  + **Attributes**
  + **Constructor**
  + **DeConstructor**
  + **Methods**