BOMBERMAN

HIGH DISTINCTION CUSTOM PROGRAM

MAY 10, 2022

QUANG NGUYEN
103493836

Table of Contents

Table of Figures

Introduction of Custom Program Design

The custom program design is based on the childhood game ‘Bomber man’, in which the player becomes a bomber who can place bombs in squares of the map to kill enemies. When the player places a bomb on the floor, the bomb will wait about 2-3 seconds to explode. The explosion occurs in the ‘cross’ direction, in which either player or enemies or breakable blocks will be destroyed immediately. Once the breakable blocks are destroyed by the bombs, it is likely to produce some special items to power up the player, such as bomb radius enhancer, bomb counter enhancer or speed the player up. These items will help the player to battle with the enemies more easily because those enemies have a special skill of creating the ultrasonic waves that can stun the player at one location for couple seconds. Once all the enemies are killed by the player, the player will win the game. However, once the character is dead due to losing all health (touch an enemy or bomb explosion corresponds to -1 HP), the player will lose the game

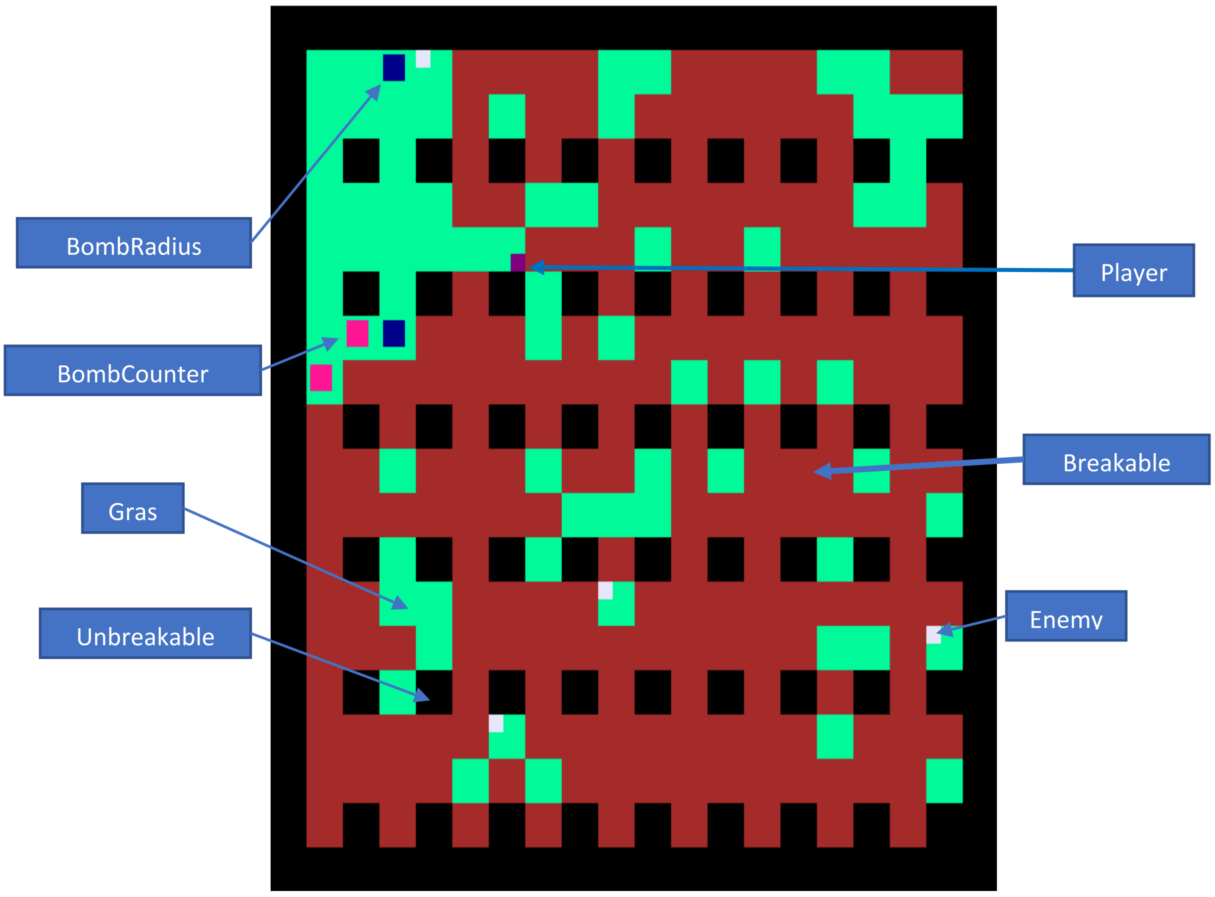

Figure 1 My program with simple shapes representing for objects

UML Diagram

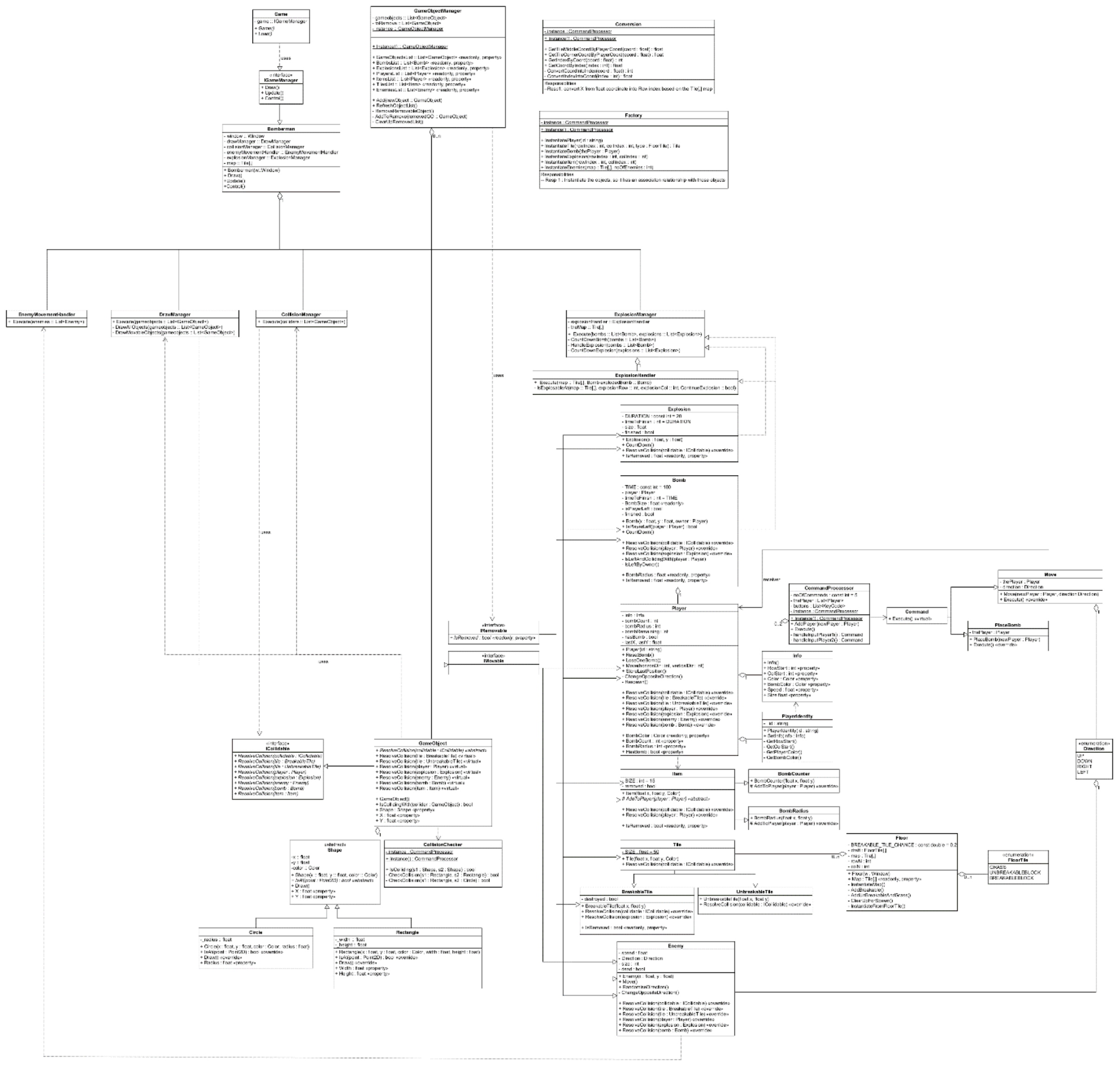

Figure 2 UML diagram for the custom program

Describe the design decisions

1. GameObject class and Its Inheritance

Start with game object in my Bomber man game, its shape is extremely basic, such as rectangle or circle. Therefore, the creation of **Shape class** as an abstract shape will store the basic information of the shape and allow its children, such as **rectangle class or circle class**, to extend its property. In our program, the relationship between **Game Object class** and Shape class is aggregation as Shape is one of the GameObject's fields and only instantiated in its constructor. In this case, the **polymorphism** is applied, where the Shape can have different forms, such as Rectangle or Circle. During the custom program, various kinds of game objects are needed. Therefore, making GameObject an abstract class is an appropriate way. Whenever the GameObject is instantiated, the object will be added into the list of game objects in the **GameObjectManager class**. This class is responsible for adding in or removing objects out of the game. Moreover, the GameObject class also implements ICollidable interface, in which this is an intended use of **Visitor Pattern**. The fact that the GameObject class has abstract ResolveCollision(ICollidable collidable) and virtual methods for ResolveCollision(with other game objects) allows the child classes to respond differently when it visits other game objects. The use of this design pattern helps the code look cleaner and easy to maintain or extend. One of the evidences is lying in the CollisionManager class, in which only one invoked method collidedObject.ResolveCollision(colliding) can solve all problems of the collisions.

2. Player class and Its Abstraction

Player class is responsible for creating the player that can be controlled by the user's input. Player contains an Info class as one of its fields, in which the Info class is used to store the basic information of a player set by PlayerIdentity class. The reason for this approach is that the Info class is a private field and whenever the Player wants to expose its default information to the outer world, we can use the property, such as public Color BombColor { get { return info.BombColor; } }. Furthermore, in the Bomber man game, it is possible to have 2 players playing together on the same keyboard. Therefore, each player will have a special ID which is passed into the PlayerIdentity class to set the corresponding default information for the player. Moreover, the Player class has an ability to ResetBomb() and LoseOneBomb(), in which these methods will increment or decrement BombRemaining by one during the game play. Additionally, the player is also responsible for knowing its position, by having function such as StoreLastPosition() (to store LastX = X and LastY = Y) and ChangeOppositeDirection() which is invoked when the player collides with other game objects. Throughout the Player class, it does not hold strongly dependent relationship with other classes in GameObject, which makes itself low coupling and leads a good object-oriented design.

3. Command Pattern for Player action

To control the Player action, I applied the **Command Pattern**, in which there is an abstract class Command and two other specific commands are Move and PlaceBomb. This is where the polymorphism is again applied. In the CommandProcessor class which is responsible for processing commands, each required input from the keyboard will instantiate a specific Command, such as new PlaceBomb(thePlayer[0]) if Space key is pressed or new Move(thePlayer[0], Direction.UP) if W key is pressed. The two commands here will be returned under the form of Command class. This method named handleInput() is set to be private to encapsulate all information needed to perform the action, which leads no other classes to invoke this method. In the method Execute(), this method is invoked and then returned a command, in which the defined command will be executed. In this case, the CommandProcessor class is used with a **Singleton Pattern**, as the player is immediately added to the List while constructing the Player object. By using Command Pattern, the Player class becomes more cohesive because now, the movement or place bomb command is responsible by CommandProcessor. This sets the independent relationship between commands and player class. Moreover, using Command Pattern allows a great extensibility, especially offering the other types of devices, such as PS4 or mouse or TV remote controller.

4. Bomb class and Its Collision Mechanism

In addition to Player class, **Bomb class** is crucial as it is a signature for the Bomber man game. This class holds a dependent relationship on the Player when containing the Player class as one of its fields. This is because Bomb's property, such as Bomb Counter or Bomb Radius of Explosion will be dependent on its owner. Therefore, the Player is one of the parameters for Bomb constructor. Furthermore, the bomb has an ability to count down, which allows itself to be displayed in a longer time. Furthermore, the collision handler in bomb is quite complicated because it requires the bomb placer to leave itself first before enabling the collision algorithm. Therefore, the two private methods for doing this are IsLeftByOwner() and IsLeftAndCollidingWith(Player player) are implemented. Next is the explosion class, in which it also has an ability to count down similar to Bomb but does not hold dependent relationship with Player class.

5. Item class and Strategy Pattern

The **Item class** is also fundamental to the Bomber man game. In this case, the **Strategy Pattern** is applied, in which Item abstract class represents for the other two items, including BombCounter and BombRadius. Every item must have a method AddToPlayer(Player player). BombCounter will increment the bomb count for the collector, while BombRadius will extend the explosion radius. The Item class hold a dependent class with Player class because Player is used as a parameter for AddToPlayer method as well as ResolveCollision(Player player) for checking the collision. If the Player and Item collides with each other, the method AddToPlayer will be invoked. Depending on which item the player collects, the method AddToPlayer in BombCounter or BombRadius will be chosen to be executed. By using Strategy Pattern, this allows the Item class to be more extensible, which means the game can have a lot more items without modifying the code too much.

6. Tile class: How to create the game map

The next class is to create the map for the game. In the game, there are three typical types of blocks, one is grass (player can walk on), one is breakable (player cannot go through but can use bomb to explode it) and one is unbreakable (player neither goes through nor use bomb to destroy it). The idea starts with creating the **Tile class**, which represents for the grass tile and does not have an ability to be destroyed or resolve collision with Player. The **unbreakable tile** inherits from the tile, but also has an ability to resolve collision with Player or Enemy. Finally, the breakable tile not only can resolve collision with Player or Enemy but implements the interface IRemovable. This interface helps to remove the Tile when it collides with the Explosion. I will discuss the interface IRemovable later. What is the class used to create the tile? It is the Floor class. This class is responsible for creating the map for the game. However, the process is quite complicated. The draft will be created first by using the FloorTile enum. This is because after finishing to add breakable, unbreakable and grass tiles, the array of FloorTile will be modified to make spots for Player Respawn. After the draft is completed, this will be passed into the

InstantiateTilesFromFloorTile() method to instantiate the Tile. Each instantiated tile will be stored in the Tile[,] 2D array.

7. Enemy class: challenging game

To make the game more challenging, the Enemy class is used to add enemies to the game. This enemy class has an ability to Move in the indicated direction, ChangeOppositeDirection when it collides with some game objects or player. It can randomise its direction by using the Random Number Generator. Also, when the enemy collides with Player, the Player will die and respawn back to the original place. When the player kills one enemies, the other enemies' speed will increase by using the static field for speed value. In the game, all enemies movement will be handled in the Execute() method of EnemyMovementHandler class which is then invoked in the Bomberman class, the largest game manager in the program.

8. Game Manager Classes

a. Game Object Manager

In the Bomber man game, there are many classes that are responsible for handling game objects. Firstly, the GameObjectManager class is a class that can store, add or remove the game objects out of the game. Therefore, a list of GameObjects is a private field and whenever a new gameobject is instantiated, it will be added into this list. Besides, the class can remove any game objects that implements the interface IRemovable. When the property IsRemoved becomes true, the private method RemoveRemovableObject() will add the removed object into ToRemove list which will be then removed through the method ClearUpRemovedList(). The removable object in this game includes bombs, explosions, items and breakable tiles. In addition, this GameObjectManager class is a singleton because it helps to add new gameobject into the list easily. Besides, the GameObjectManager has many other properties such as BombsList, ExplosionsList, PlayersList, ItemsList, TilesList or EnemiesList which will be passed into the other methods as parameters.

b. Draw Manager

The next manager class is DrawManager which is responsible for drawing game objects. This mechanism is to prioritise drawing movable objects and then the other game objects. This is because when the new game objects are added to game (example is grass tile after breakable tile is destroyed), they are always on the top layer of the game and the player will be drawn under these layers and become invisible. To draw all game objects, the DrawManager uses the List of GameObjects from GameObjectManager classes and is called in the Bomberman class which is the game manager.

c. Explosion Manager and Explosion Handler

One of the important manager classes in the Bomberman game is explosion manager. This class is designed not to be highly cohesive as it is responsible for bomb countdown, handling explosion, and explosion countdown. However, this helps the maintain step become easier because these events are placed in order. For the bomb and explosion count down, these have the same method CountDown(), which will be called in this ExplosionManager class. When the countdown is finished, the property IsRemoved is set to be true inside bomb and explosion class. As discussed earlier, these bombs and explosions will be added to the removed list in GameObjectManager class. Moreover, the ExplosionManager holds a dependent relationship with ExplosionHandler class which is responsible for checking whether the explosion should occur or not. To achieve this, the Tile[,] map 2D array is used to give the knowledge about surrounding tiles. The mechanism of the game explosion is that it can explode in four directions (UP, DOWN, LEFT, and RIGHT). The explosion will continue to go if the tile is grass, while stopping at breakable or unbreakable tile. Whenever the breakable tile is collided with explosion, the property IsRemoved is set to be true and therefore the breakable tile is removed out of the game. That spot is then replaced with a grass tile that is instantiated within the IsExplosableAt method. This method is not quite cohesive, a bad object-oriented design, as it has too many responsibilities, including creating tile grass, explosion and modify the Tile[,] map 2D.

d. Bomberman Manager: The most powerful class

Finally, the most powerful manager class in this program is BomberMan class. This class holds the information of the other manager classes, such as drawManager, collisionManager, enemyMovementHandler, explosionManager, and game map under the data type Tile [,]. As this class implements the IGameManger interface, it signs a contract of compulsorily having Draw(), Update() and Control() methods in its class. For Draw() method, the DrawManager will be executed. For Update() method, the CollisionManager, EnemyMovementHandler and ExplosionManager will be executed along with Tile[,] map as a parameter. Besides, the Update() method also includes the GameObjectManager.Instance().RefreshObjectList() to remove the required objects. In the Control() method, the command processor will be executed by calling its method CommandProcessor.Instance().Execute().

9. Factory class:

Furthermore, you may wonder how the object is instantiated in the program. This is executed by using **Factory class** which is also a factory pattern. By applying this class in the program, it increases the cohesion for each other classes because the factory class now is only responsible for creating objects. As there is only one factory in the game, a single pattern is applied.

Summary:

- **Used Design Pattern:**
  - Singleton Pattern
  - Strategy Pattern
  - Factory Pattern
  - Visitor Pattern
  - Command Pattern
- **Polymorphism:**
  - Shape
  - Item

    - Command
- **Inheritance:**
    - GameObject: parent class
    - Tile: parent class
    - Item: parent class
- **Interface:**
    - IMovable
    - IRemovable
    - ICollidable
    - IGameManager
- **A good object-oriented design:**
    - Player class: is independent of other game objects.
    - Collision Manager system: use visitor pattern, high extensibility, and maintainability.
    - Item class: use strategy pattern, can easily add more items to the game without modifying the program much.
    - Use Factory class to specialise for creating objects.
    - Implement IRemovable class for the object that can be removed. This follows open for extension and close for modification.
- **A bad object-oriented design:**
    - ExplosionHandler class has too many responsibilities → low cohesion