Design Overview for <<Adventure Time>>

Name: Huynh Trung Chien
Student ID: 104848770

Summary of Program

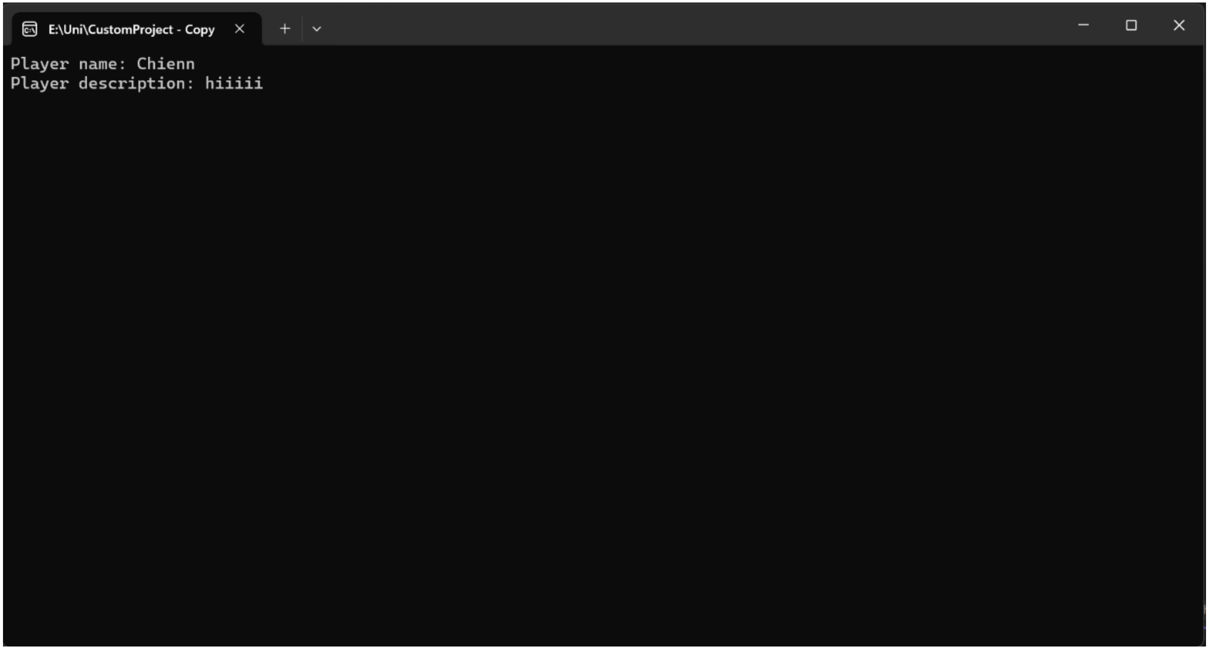
For my HD custom program, I aimed to create an adventure game using the SplashKit library in C#. The first part of this HD Custom Program Initial Design is about the features for my D Custom Program, and the extended features to get HD are shown below this part.

First the users will be asked for their personal information before playing the game, after that the user will be directed to the game interface. In this game, the player will use the Up, Down, Right, and Left keys on their keyboard to move the character around the map. When moving around the player can collected the objects on the map such as golds, swords. There are also bombs in the maps, and if the players touch these bombs, they will lose the game. Then the players can restart the games by pressing the R key or quit the game by pressing the Q key. Moreover, the players also have their inventory, and they can check the inventory by using the Tab key.

This is just the initial plan for my custom program, there will be changes when I develop this game in the future

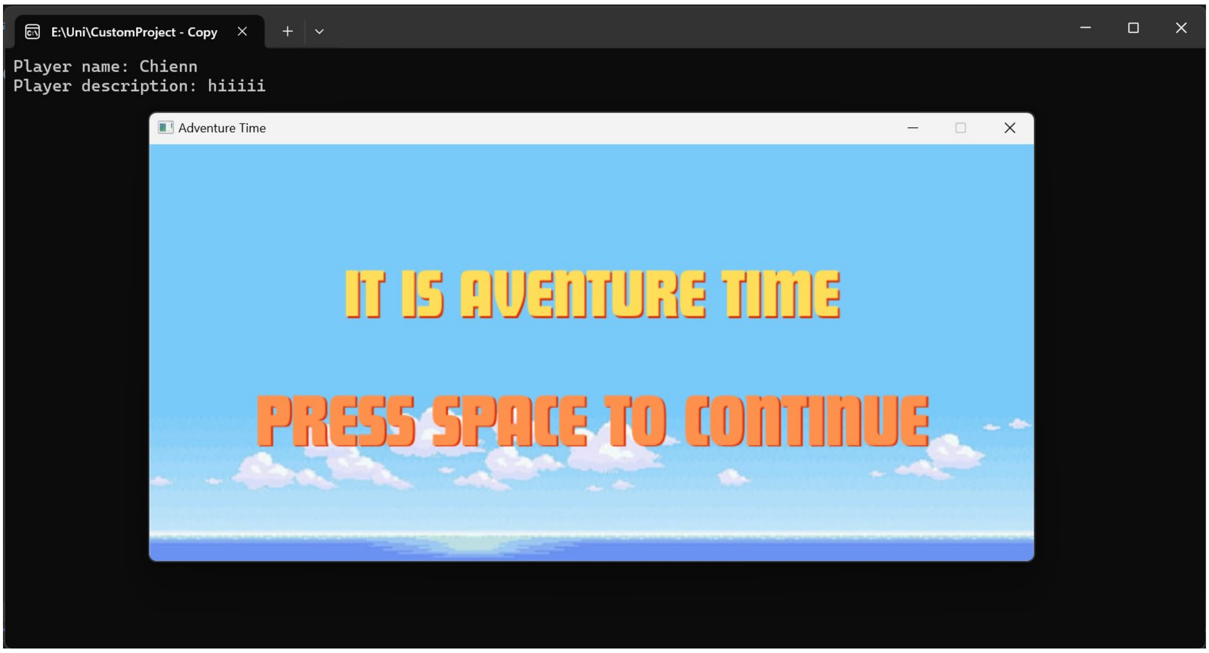
Here are some first pictures of the game:

- This is the game requiring the player to type in their name before starting, the player will be asked until they type in the acceptable name and description.



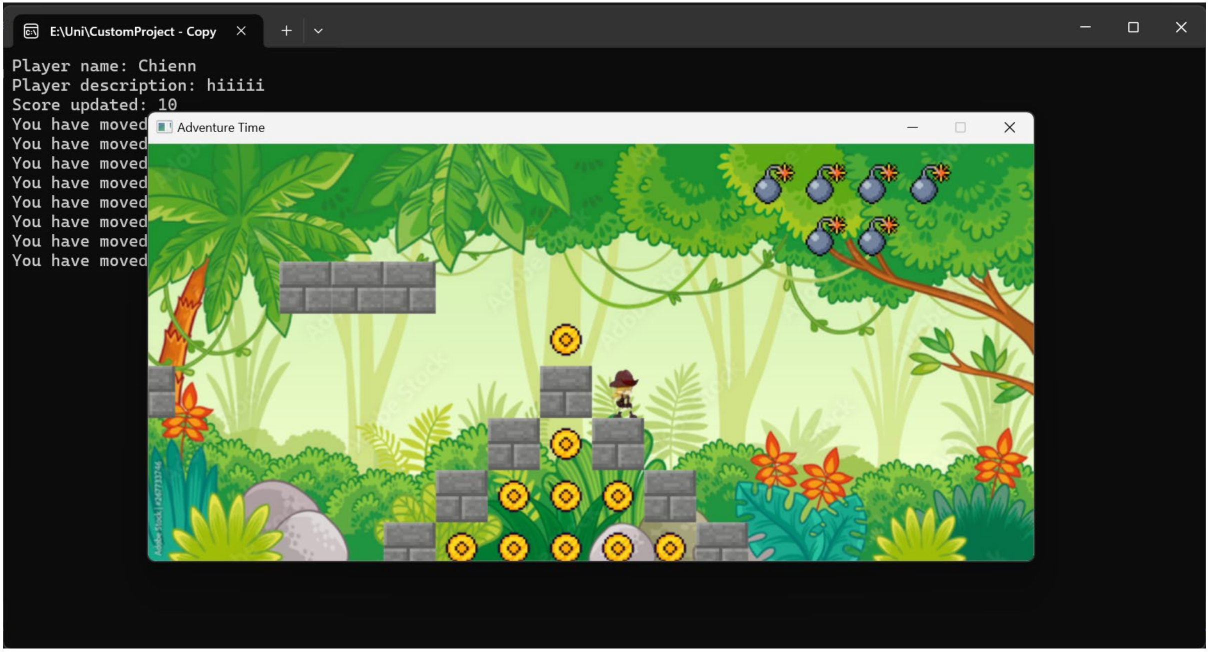
```
Player name: Chienn
Player description: hiiiii
```

- Here is the game interface:



```
Player name: Chienn
Player description: hiiiii
```

- Here is how the map and the player look like in the game:



```
Player name: Chienn
Player description: hiiiii
Score updated: 10
You have moved
You have moved
You have moved
You have moved
You have moved
You have moved
You have moved
You have moved
```

D CUSTOM PROGRAM DESIGN

Required Roles

Describe each of the classes, interfaces, and any enumerations you will create. Use a different table to describe each role you will have, using the following table templates.

Table 1: <<IdentifiableObject>> details – Public class

Responsibility	Type Details	Notes

Fields	- _identifiers : List<string>	Identifiers to locate objects in game
Constructor	+ IdentifiableObject(string[] idents)	
Methods	+ AreYou(string id) : bool	Check if the object is itself
	+ FirstId() : string	Return the first id of the objects

Table 2: <<GameObject : IdentifiableObject>> details – abstract class

Responsibility	**Type Details**	**Notes**
Fields	- _description : string	The object's description
	- _name : string	The object's name
Constructor	+ GameObject(string[] ids, string name, string description)	
Properties	+ Name : string <<readonly property>>	
	+ ShortDescription : string <<readonly property>>	Provide short description
	+ FullDescription : string <<readonly property>>	Provide the full description

Table 3: <<DrawableObject : GameObject>> details – abstract class

Responsibility	**Type Details**	**Notes**
Fields	# _xLocation : double # _yLocation : double # _image : Bimap	
Constructor	+ DrawableObject(double xLocation, double yLocation, string[] ids, string name, string description, string imagePath)	
Methods	+ Draw() : void <<virtual>>	Draw the objects on the map
Properties	+ X: double <<properties>> + Y: double <<properties>> + Image : Bitmap <<readonly properties>>	

Table 4: <<CollectibleBomb : DrawableObject>> details - public class

Responsibility	**Type Details**
Constructor	+ CollectibleBomb(double xLocation, double yLocation) : base(xLocation, yLocation, new string[] { "bomb" }, "Bombs!!", "Do not touch the bombs!", "Bomb.png")

Table 5: <<Gold : DrawableObject>> details – public class

Responsibility	**Type Details**
Constructor	+ CollectibleGold(double xLocation, double yLocation) : base(xLocation, yLocation, new string[] { "gold" }, "Golds!!", "Collect golds to win the game!!", "Gold.png")

Table 6: <<Land : DrawableObject>> details – public class

Responsibility	Type Details
Constructor	+ Land(double xLocation, double yLocation) : base(xLocation, yLocation, new string[] { "" }, "", "", "land.png")

Table 7: <<Sword : DrawableObject>> details – public class

Responsibility	Type Details
Constructor	+ Sword(double xLocation, double yLocation) : base(xLocation, yLocation, new string[] { "sword" }, "Metal Sword!", "Meelee weapons!", "sword.png")

Table 8: <<GameMap : GameObject>> details – public class

Responsibility	Type Details	Notes
Fields	- _mapWidth : int - _mapHeight : int - _background : Bitmap - _jungle : Bitmap - _sky : Bitmap - _sea : Bitmap - _lands : List<Land> - _bombs : List<CollectibleBomb> - _golds : List<CollectibleGold> - _sword : List<Sword> - _txtMap : string[]	
Constructor	+ GameMap(string[] ids, string name, string description)	
Methods	- ReadMapFromFile(string filename) : void	Reads the map from a text file and initializes _txtMap.
	- LoadBackgrounds() : void	Loads the background images.
	+ SetUpGameMap() : void	Sets up the game map based on the text map representation.
	+ Draw() : void	Draws the game map and its objects.
Properties	+ Width : int <<readonly property>>	Returns the width of the game map in pixels.
	+ Height : int <<readonly property>>	Returns the height of the game map in pixels.
	+ Lands : List<Land> <<readonly property>>	Returns the list of land objects.
	+ Bombs : List<CollectibleBomb> <<readonly property>>	Returns the list of collectible bombs.
	+ Golds : List<CollectibleGold> <<readonly property>>	Returns the list of collectible golds.
	+ Swords : List<Sword> <<readonly property>>	Returns the list of swords.
	+ Background : Bitmap <<property>>	Gets or sets the current background.

	+ Sea : Bitmap <<readonly property>>	Returns the sea background bitmap.
	+ Jungle : Bitmap <<readonly property>>	Returns the jungle background bitmap.
	+ Sky : Bitmap <<readonly property>>	Returns the sky background bitmap.

Table 9: <<Inventory>> details – public class

Responsibility	Type Details	Notes
Fields	- _items : List<Item>	Stores the list of items in the inventory.
Constructor	+ Inventory() : void	Initializes the inventory with an empty list of items.
Methods	+ HasItem(string id) : bool	Checks if an item with the given identifier is in the inventory.
	+ Put(Item itm) : void	Adds an item to the inventory.
	+ Fetch(string id) : Item	Retrieves an item with the given identifier from the inventory.
Properties	+ ItemList : string <<readonly property>>	Returns a string listing the short descriptions of all items in the inventory.

Table 10: <<IHaveInventory>> details - Interface

Responsibility	Type Details	Notes
Methods	Locate(string id) : GameObject	Locates and returns a GameObject with the given identifier from the inventory.
Properties	+ Name : string <<property>>	Gets the name of the inventory holder.

Table 11: <<Item : GameObject>> details – public class

Responsibility	Type Details	Notes
Constructor	+ Item(string[] idents, string name, string description)	Initializes the Item with identifiers, name, and description.

Table 12: <<CollisionHandler>> details – public class

Responsibility	Type Details	Notes
Fields	- _gameMap : GameMap	Stores the reference to the game map.
	- _player : Player	Stores the reference to the player.

Constructor	+ CollisionHandler(GameMap gamemap, Player player)	Initializes the collision handler with references to the game map and player.
Methods	+ CheckBombCollision() : void	Checks for collisions between the player and bombs, handles the collision action, and removes the bombs.
	+ IsPlayerOnGround() : bool	Checks if the player is on the ground by verifying collision with land.
	+ IsCollideWithLand(double x, double y) : bool	Checks for collisions between the player and land at the given coordinates.

Table 13: <<CameraHandler>> details – public class

Responsibility	**Type Details**	**Notes**
Fields	- _gameMap : GameMap	Stores the reference to the game map.
	- _player : Player	Stores the reference to the player.
Constructor	+ CameraHandler(GameMap gamemap, Player player)	Initializes the camera handler with references to the game map and player.
Methods	+ HandleCamera() : void	Adjusts the camera position based on the player's location and the boundaries of the game map.

Table 14: <<AnimationHandler>> details – public class

Responsibility	**Type Details**	**Notes**
Fields	- _player : Player	Stores the reference to the player.
Constructor	+ AnimationHandler(Player player)	Initializes the animation handler with a reference to the player.
Methods	+ HandleCamera() : void	Adjusts the camera position based on the player's location and the boundaries of the game map.
	- UpdateStandingAnimation() : void	Updates the player's standing animation.
	- UpdateRunningAnimation() : void	Updates the player's running animation based on key presses.
	- StartAnimationIfNotRunning(string animationName) : void	Starts the specified animation if it is not already running.
	+ UpdateDeadAnimation() : void	Updates the player's dead animation.

	+ Die() : void	Handles the player's death animation.

Table 15: <<Command : IdentifiableObject>> details – abstract class

Responsibility	Type Details	Notes
Fields	Inherits fields from IdentifiableObject	Stores the reference to the player.
Constructor	+ Command(string[] ids)	Initializes the command with identifiers.
Methods	+ Execute(Player p, string[] text) : string <<abstract>>	Abstract method to execute the command.

Table 16: <<LookCommand : Command>> details : public class

Responsibility	Type Details	Notes
Fields	Inherits fields from Command	No additional fields.
Constructor	+ LookCommand()	Initializes the LookCommand with the identifier "look".
Methods	+ Execute(Player p, string[] text) : string <<override>>	Executes the look command, returning the description of the specified object.
	- LookAtIn(string thingId, IHaveInventory container) : string	Looks for an item in the given inventory and returns its description.

Table 17: <<GameProcessor>> details : public class

Responsibility	Type Details	Notes
Fields	- _window : Window	Stores the game window.
	- _map : GameMap	Stores the game map.
	- _player : Player	Stores the player.
	- _playerName : string	Stores the player's name.
	- _interface : Bitmap	Stores the interface bitmap.
	- _playerName : string	Stores the player's name.
	- _playerDescription : string	Stores the player's description.
	- _gameStarted : bool	Indicates if the game has started.
	- _gameOver : bool	Indicates if the game is over.
	- _showingInstructions : bool	Indicates if the instructions are being shown.
Constructor	+ GameProcessor()	Initializes the game processor, setting up the game state.
Methods	+ Run() : void	Runs the main game loop.
	- GetUserName() : void	Looks for an item in the given inventory and returns its description.
	- GetUserDescription() : void	Prompts the user for their description.
	- InitializeMap() : void	Initializes the game map.

	- InitializePlayer() : void	Initializes the player.
	- HandleGameStart() : void	Handles the game start sequence, including showing instructions.
	- HandleGameOver() : void	Handles the game over sequence.
	- HandleGamePlay() : void	Handles the main gameplay sequence.
	- CheckDeathState() : void	Checks if the player is dead and handles game over state.
	- UpdateInventory() : void	Updates the player's inventory based on collected items.
	- OpenInventory() : void	Handles opening and displaying the player's inventory.
	- UpdateGameMap() : void	Updates the game map based on player actions.

Table 18: <<Player : DrawableObject, IHaveInventory>> details – public class

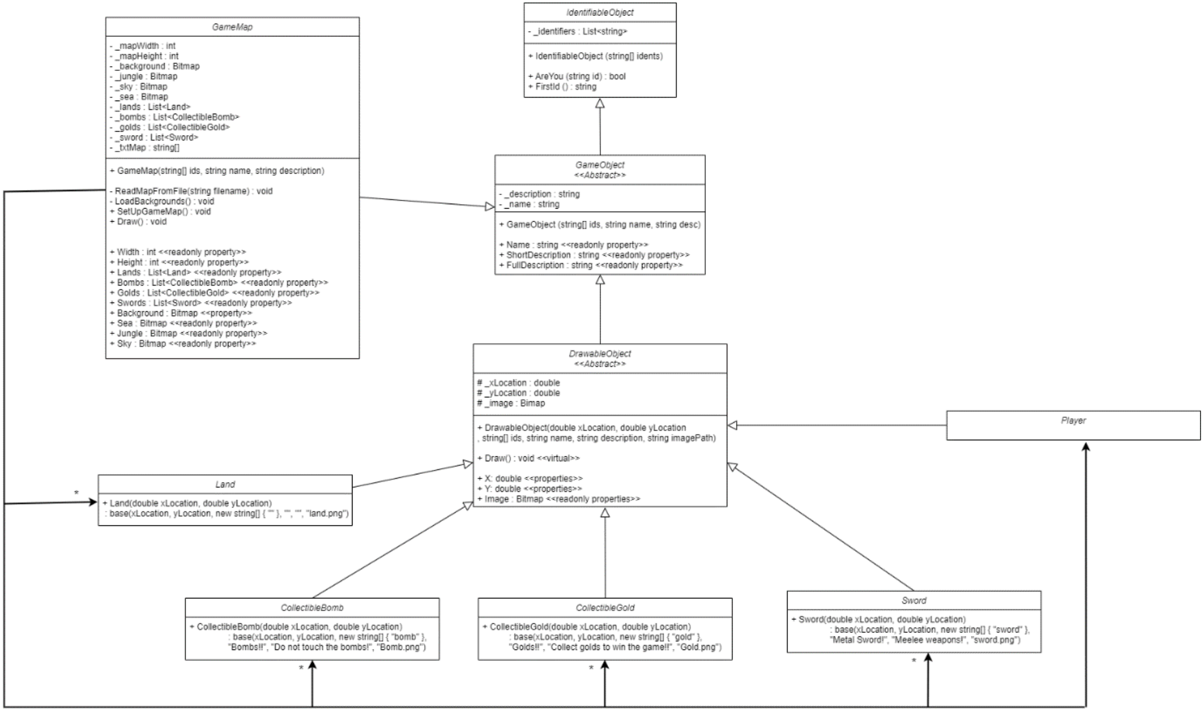
Responsibility	Type Details	Notes
Fields	- _window : Window	Stores the gravity effect on the player.
	- _playerSprite : Sprite	Stores the player's sprite.
	- _gameMap : GameMap	Stores the game map reference.
	- _isDead : bool	Indicates if the player is moving.
	- _isMoving : bool	Stores the interface bitmap.
	- _inventory : Inventory	Stores the player's inventory.
	- _collectedGolds : int	Stores the number of collected golds.
	- _collectedSwords : int	Stores the number of collected swords.
Constructor	+ Player(GameMap gameMap, string name, string description)	Initializes the player with the game map, name, and description.
Methods	+ Update() : void	Updates the player's state.
	+ Draw() : void <<override>>	Draws the player on the screen.
	- UpdateYLocation() : void	Updates the player's Y location considering gravity and collisions.
	- InitializePlayerSprite() : void	Initializes the player's sprite.
	- InitializePlayerState(GameMap gameMap) : void	Initializes the player's state.
	+ Locate(string id) : GameObject <<interface implementation>>	Locates an item in the player's inventory or the player itself.
Properties	+ Inventory : Inventory <<property>>	Gets or sets the player's inventory.

	+ Gravity : double <<property>>	Gets or sets the player's gravity.
	+ IsMoving : bool <<property>>	Gets or sets the player's dead state.
	+ IsDead : bool <<property>>	Updates the player's inventory based on collected items.
	+ GameMap : GameMap <<readonly property>>	Gets the game map reference.
	+ CollectedGolds : int <<property>>	Gets or sets the number of collected golds.
	+ CollectedSwords : int <<property>>	Gets or sets the number of collected swords.
	+ PlayerSprite : Sprite <<readonly property>>	Gets the player's sprite.

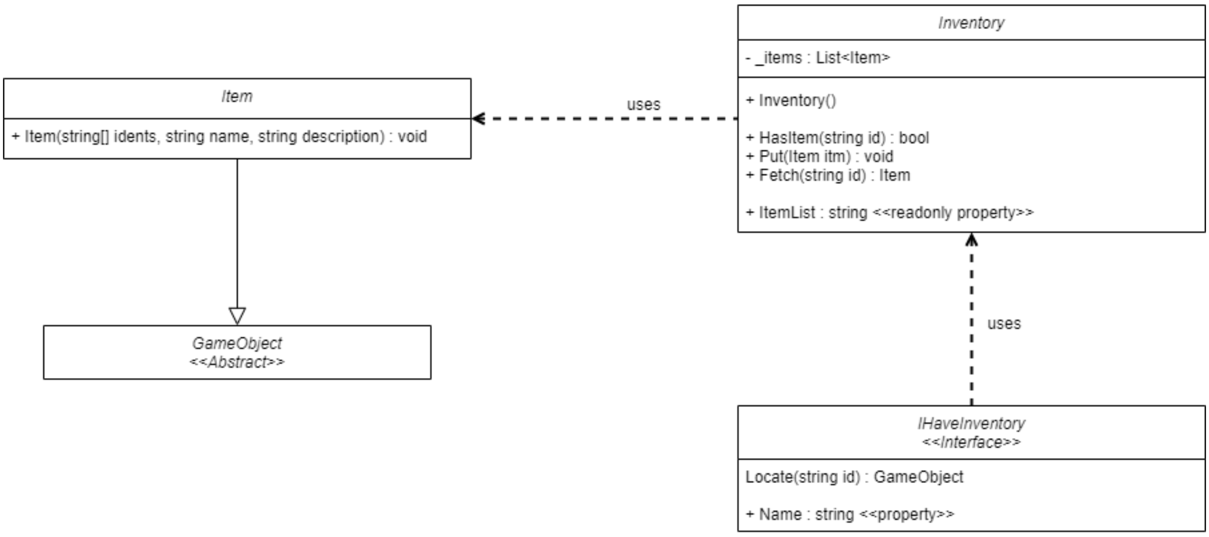
Class Diagram

Here is my initial design for my D custom program.

- **GameMap:**

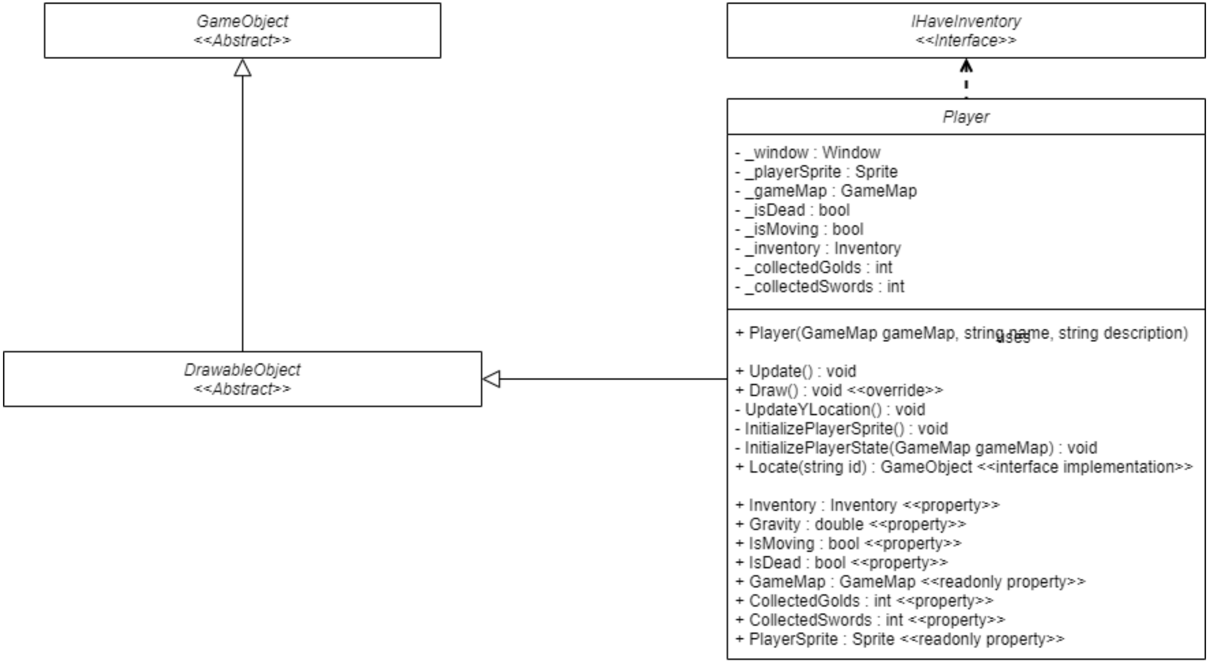

GameMap
- _mapWidth : int
- _mapHeight : int
- _background : Bitmap
- _jungle : Bitmap
- _sky : Bitmap
- _sea : Bitmap
- _lands : List<Land>
- _bombs : List<CollectibleBomb>
- _golds : List<CollectibleGold>
- _sword : List<Sword>
- _txtMap : string[]
+ GameMap(string[] ids, string name, string description)
- ReadMapFromFile(string filename) : void
- LoadBackgrounds() : void
+ SetUpGameMap() : void
+ Draw() : void
+ Width : int <<readonly property>>
+ Height : int <<readonly property>>
+ Lands : List<Land> <<readonly property>>
+ Bombs : List<CollectibleBomb> <<readonly property>>
+ Golds : List<CollectibleGold> <<readonly property>>
+ Swords : List<Sword> <<readonly property>>
+ Background : Bitmap <<property>>
+ Sea : Bitmap <<readonly property>>
+ Jungle : Bitmap <<readonly property>>
+ Sky : Bitmap <<readonly property>>
IdentifiableObject
- _identifiers : List<string>
+ IdentifiableObject (string[] idents)
+ AreYou (string id) : bool
+ FirstId () : string
GameObject
<<Abstract>>
- _description : string
- _name : string
+ GameObject (string[] ids, string name, string desc)
+ Name : string <<readonly property>>
+ ShortDescription : string <<readonly property>>
+ FullDescription : string <<readonly property>>
DrawableObject
<<Abstract>>
# _xLocation : double
# _yLocation : double
# _image : Bimap
+ DrawableObject(double xLocation, double yLocation
, string[] ids, string name, string description, string imagePath)
+ Draw() : void <<virtual>>
+ X: double <<properties>>
+ Y: double <<properties>>
+ Image : Bitmap <<readonly properties>>
Player
*
Land
+ Land(double xLocation, double yLocation)
: base(xLocation, yLocation, new string[] { "" }, "", "", "land.png")
CollectibleBomb
+ CollectibleBomb(double xLocation, double yLocation)
: base(xLocation, yLocation, new string[] { "bomb" },
"Bombs!!", "Do not touch the bombs!", "Bomb.png")
*
CollectibleGold
+ CollectibleGold(double xLocation, double yLocation)
: base(xLocation, yLocation, new string[] { "gold" },
"Golds!!", "Collect golds to win the game!!", "Gold.png")
*
Sword
+ Sword(double xLocation, double yLocation)
: base(xLocation, yLocation, new string[] { "sword" },
"Metal Sword!", "Meelee weapons!", "sword.png")
*

- **Inventory:**



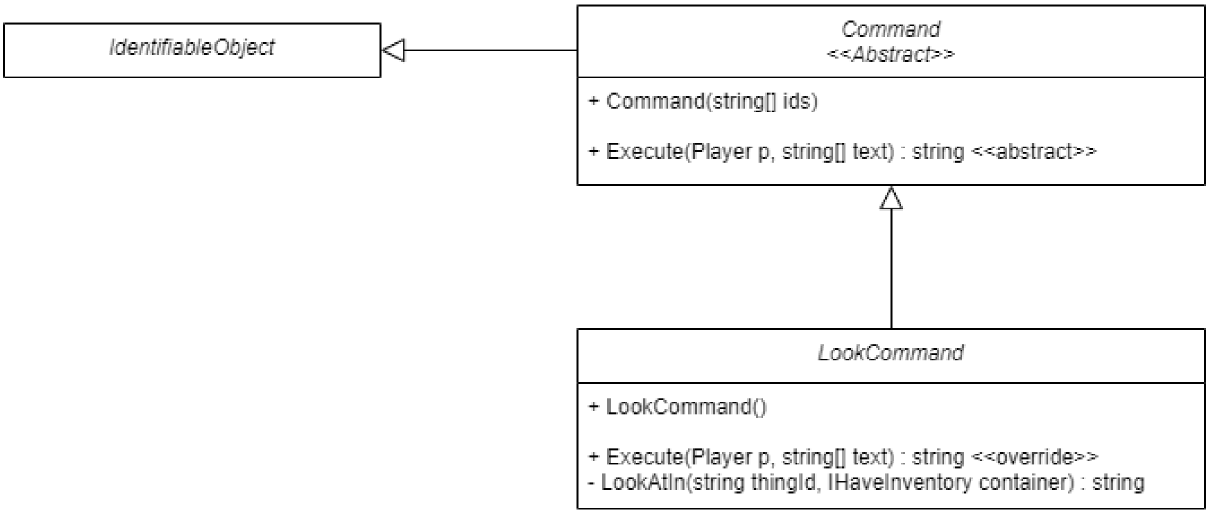
Item
+ Item(string[] idents, string name, string description) : void
GameObject
<<Abstract>>
uses
Inventory
- _items : List<Item>
+ Inventory()
+ HasItem(string id) : bool
+ Put(Item itm) : void
+ Fetch(string id) : Item
+ ItemList : string <<readonly property>>
uses
IHaveInventory
<<Interface>>
Locate(string id) : GameObject
+ Name : string <<property>>

- **Player:**



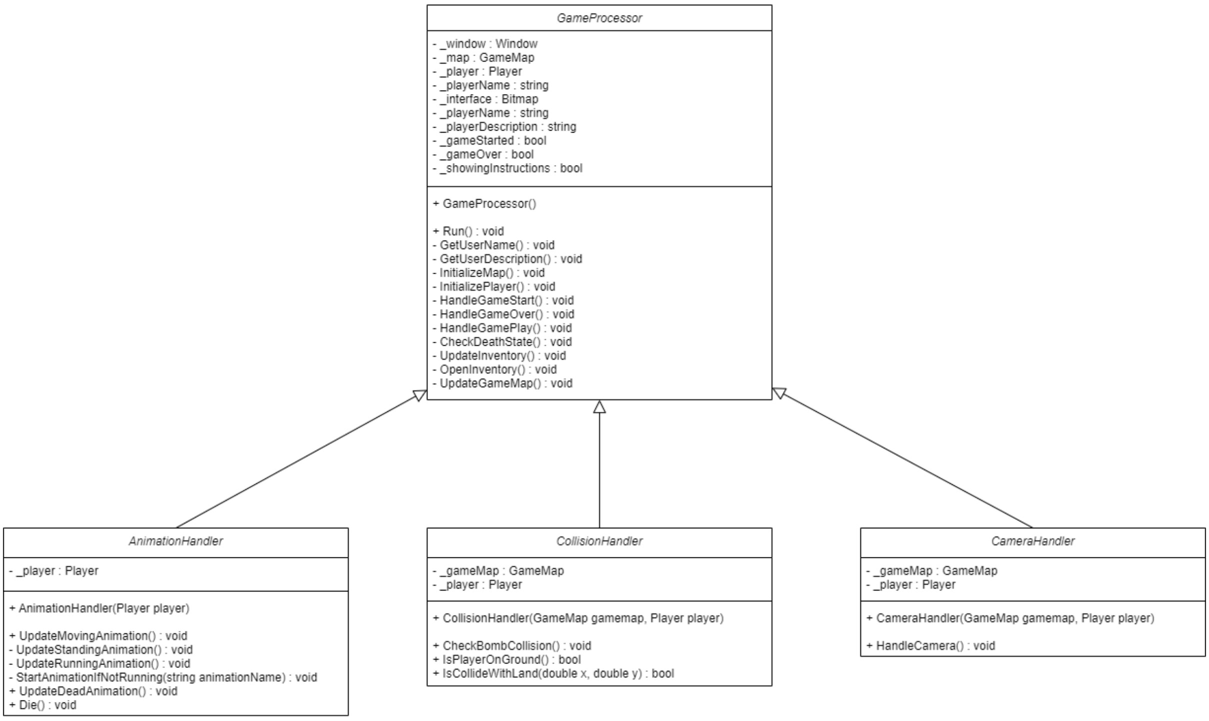
GameObject
<<Abstract>>
DrawableObject
<<Abstract>>
IHaveInventory
<<Interface>>
Player
- _window : Window
- _playerSprite : Sprite
- _gameMap : GameMap
- _isDead : bool
- _isMoving : bool
- _inventory : Inventory
- _collectedGolds : int
- _collectedSwords : int
+ Player(GameMap gameMap, string name, string description)
uses
+ Update() : void
+ Draw() : void <<override>>
- UpdateYLocation() : void
- InitializePlayerSprite() : void
- InitializePlayerState(GameMap gameMap) : void
+ Locate(string id) : GameObject <<interface implementation>>
+ Inventory : Inventory <<property>>
+ Gravity : double <<property>>
+ IsMoving : bool <<property>>
+ IsDead : bool <<property>>
+ GameMap : GameMap <<readonly property>>
+ CollectedGolds : int <<property>>
+ CollectedSwords : int <<property>>
+ PlayerSprite : Sprite <<readonly property>>

- **Command:**



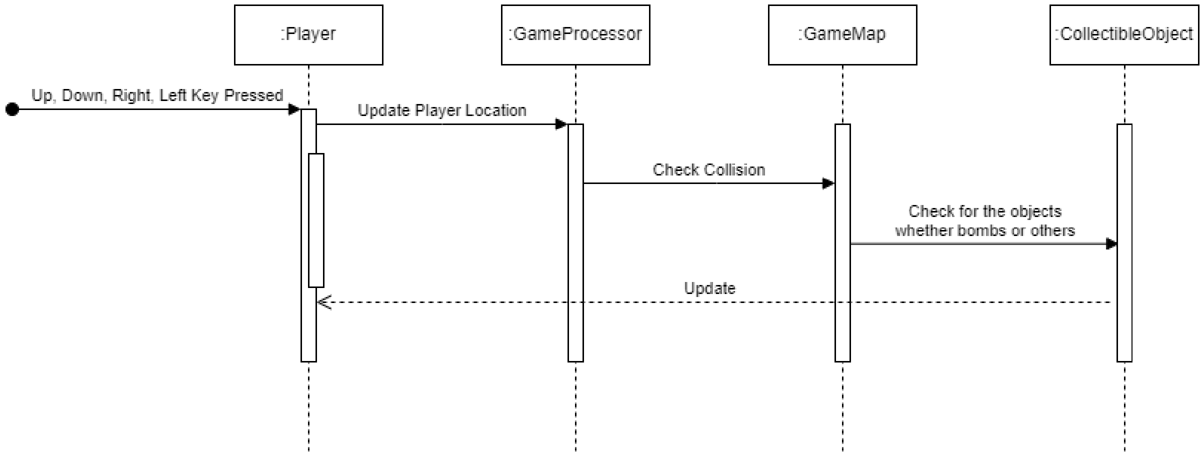
IdentifiableObject
Command
<<Abstract>>
+ Command(string[] ids)
+ Execute(Player p, string[] text) : string <<abstract>>
LookCommand
+ LookCommand()
+ Execute(Player p, string[] text) : string <<override>>
- LookAtIn(string thingId, IHaveInventory container) : string

- **Game Processor:**


GameProcessor
- _window : Window
- _map : GameMap
- _player : Player
- _playerName : string
- _interface : Bitmap
- _playerName : string
- _playerDescription : string
- _gameStarted : bool
- _gameOver : bool
- _showingInstructions : bool
+ GameProcessor()
+ Run() : void
- GetUserName() : void
- GetUserDescription() : void
- InitializeMap() : void
- InitializePlayer() : void
- HandleGameStart() : void
- HandleGameOver() : void
- HandleGamePlay() : void
- CheckDeathState() : void
- UpdateInventory() : void
- OpenInventory() : void
- UpdateGameMap() : void
AnimationHandler
- _player : Player
+ AnimationHandler(Player player)
+ UpdateMovingAnimation() : void
- UpdateStandingAnimation() : void
- UpdateRunningAnimation() : void
- StartAnimationIfNotRunning(string animationName) : void
+ UpdateDeadAnimation() : void
+ Die() : void
CollisionHandler
- _gameMap : GameMap
- _player : Player
+ CollisionHandler(GameMap gamemap, Player player)
+ CheckBombCollision() : void
+ IsPlayerOnGround() : bool
+ IsCollideWithLand(double x, double y) : bool
CameraHandler
- _gameMap : GameMap
- _player : Player
+ CameraHandler(GameMap gamemap, Player player)
+ HandleCamera() : void

Sequence Diagram

Here is an initial diagram showing how the game is working.



EXTENDED FEATURES FOR THE HD CUSTOM PROGRAM

Any additional features or complexity could be added

Because my current custom program just allows the players to move around the maps to collect the golds and avoid the bombs, it is not quite an interesting game. Therefore, I want to add some other kinds of objects into the game for the player to collect and increase the complexity of the game.

There are some objects that I want to add such as JumpPotion, SpeedPotion, Sword and maybe some other objectss. Besides that, for those objects, the player can only collect when they collect enough number of coins. Moreover, the JumpPotion and SpeedPotion only increase the speed of the player for a specific number of seconds. Finally, I want all of the notifications about those things appear on the console to let the player know about their progress in the game.

Opportunities to use design patterns and why

From those things that I want to add into my current designs, I have found some opportunities to add some design patterns as well. Here are some design patterns that I want to add and why:

- **Singleton Design Pattern:** I want to use the Singleton design pattern for managing the timers that control the temporary effects of the SpeedPotion and JumpPotion. This design pattern will ensure that there's only one instance of each timer running at any given time, providing consistency and preventing redundant or conflicting timers. Here are some benefits, which are the reasons why I choose the Singleton design pattern:
    - **Consistency:** Ensures that only one instance of each timer exists, avoiding conflicts and redundant timers.
    - **Resource Efficiency:** Helps save memory and resources by not creating multiple timer instances.
    - **Global Access:** Allows access to the timer from anywhere in the program.
    - **Maintainability:** Makes the code more maintainable and easier to debug by centralizing timer management.
- **Observer Design Pattern:** I also intend to use the Observer design pattern because it helps to notify players of various game information via the console, such as scores, when they can collect certain objects, and the timing of some effects. Moreover, it ensures efficient and flexible notifications to players. Here are some benefits, which are the reasons why I choose the Observer design pattern:
    - **Decoupling:** The Observer pattern decouples the notification logic from the game mechanics, making it easy to add, remove, or modify observers without altering the core game logic.
    - **Consistency:** It ensures that all relevant events are communicated consistently and immediately to the player, enhancing the user experience.
    - **Flexibility:** Allows for easy management and updates of the notification system independently from the game logic.
- **Strategy Design Pattern:** I also intend to use Strategy design pattern because it helps me to manage different types of jumps and movements, such as normal jump, high jump, normal movement, fast movement, normal collect, and advanced collect. Moreover, it allows for dynamic changes at runtime and promotes maintainability and flexibility. Here are some benefits, which are the reasons why I choose Strategy design pattern:
    - **Flexibility:** I can switch between different strategies (e.g., different types of jumps or movements) at runtime without changing the game's logic.
    - **Maintainability:** Each strategy is encapsulated in its own class, making it easier to manage, extend, and debug.
    - **Scalability:** Adding new strategies is straightforward and doesn't require modifying existing code, adhering to the open/closed principle.

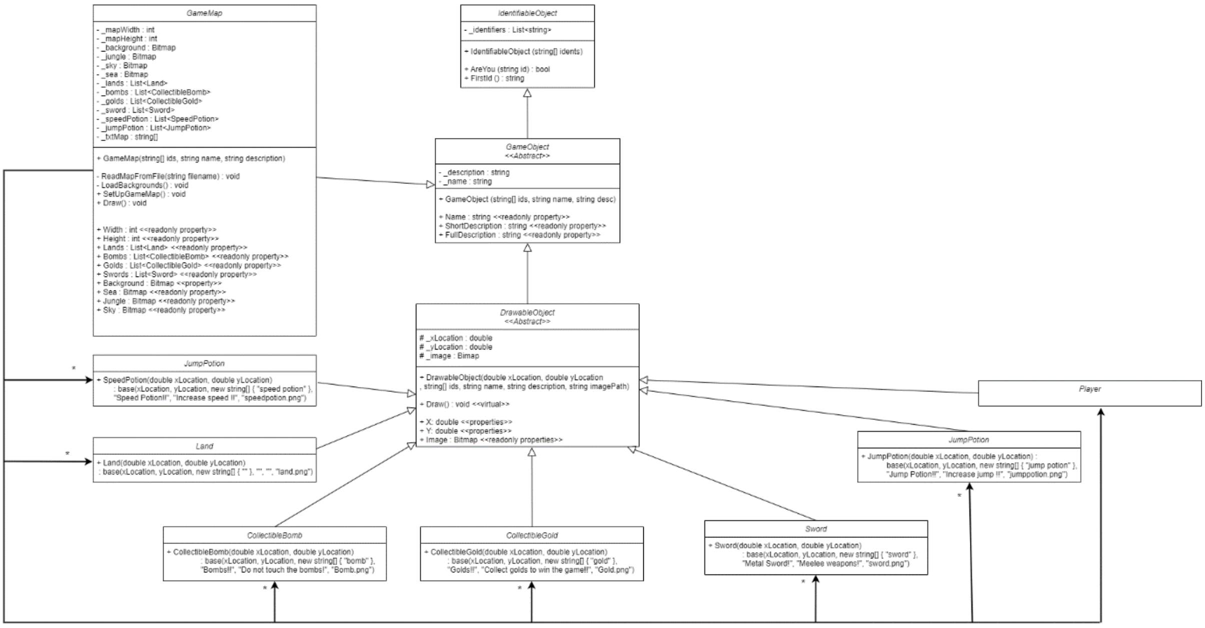
What makes my new program design more complex, and/or well done than a D level custom program design.

- My new program design for <<Adventure Time>> exhibits a higher level of complexity and sophistication compared to a D-level custom program design due to several advanced features and design principles:

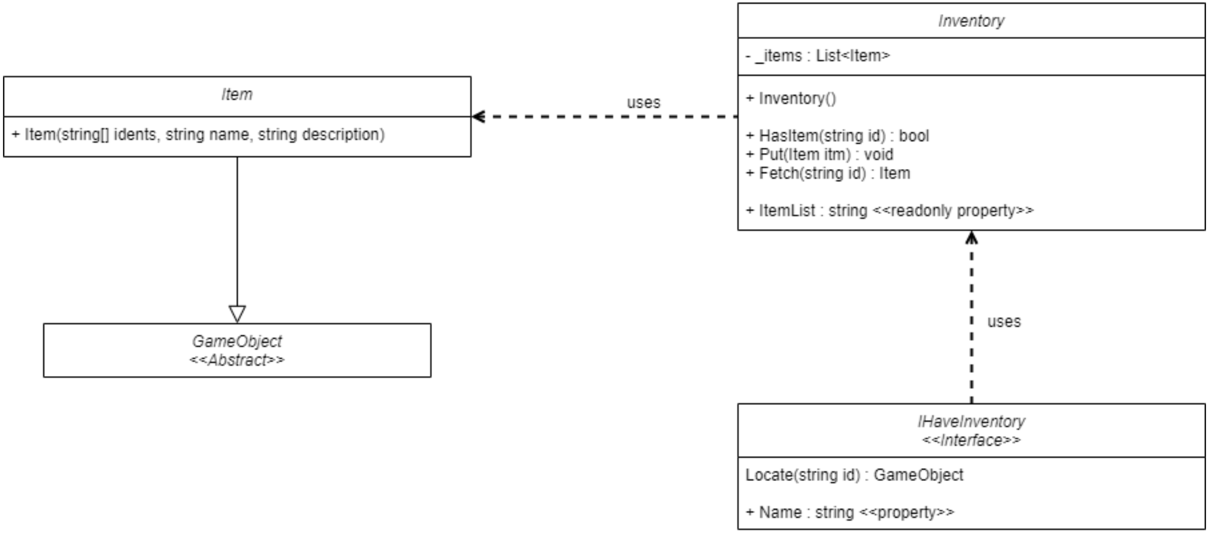
  1. **Advanced Design Patterns:**
     - **Singleton Pattern:** This pattern is used for managing timers, ensuring that only one instance of each timer exists, which provides consistency and prevents conflicts.
     - **Observer Pattern:** This pattern is used to notify players of various game events via the console, decoupling the notification logic from the game mechanics and ensuring flexible and consistent communication.
     - **Strategy Pattern:** This pattern is employed to manage different types of jumps, movements, and collection strategies, allowing dynamic changes at runtime and promoting maintainability and flexibility.
  2. **Dynamic and Interactive Gameplay:**
     - The game includes various collectible items (e.g., golds, swords, potions) and interactive elements (e.g., bombs, land), each with specific behaviors and effects. This diversity enhances gameplay and provides a richer user experience.
     - The ability to dynamically switch between different strategies (e.g., types of jumps or movements) at runtime adds a layer of complexity and keeps the game engaging.
  3. **Enhanced User Experience:**
     - The use of real-time notifications via the console improves the user experience by providing immediate feedback and keeping players informed about their progress and game events.
  4. **Scalability and Maintainability:**
     - The modular design and use of design patterns make the program highly scalable. Adding new functionalities or refactoring components can be done without affecting the overall system, adhering to the open/closed principle.
     - The separation of concerns facilitated by the Observer pattern makes the notification system easy to update independently from the core game logic, ensuring that the program can handle increased scale and complexity efficiently.

HD Updated Class Diagram

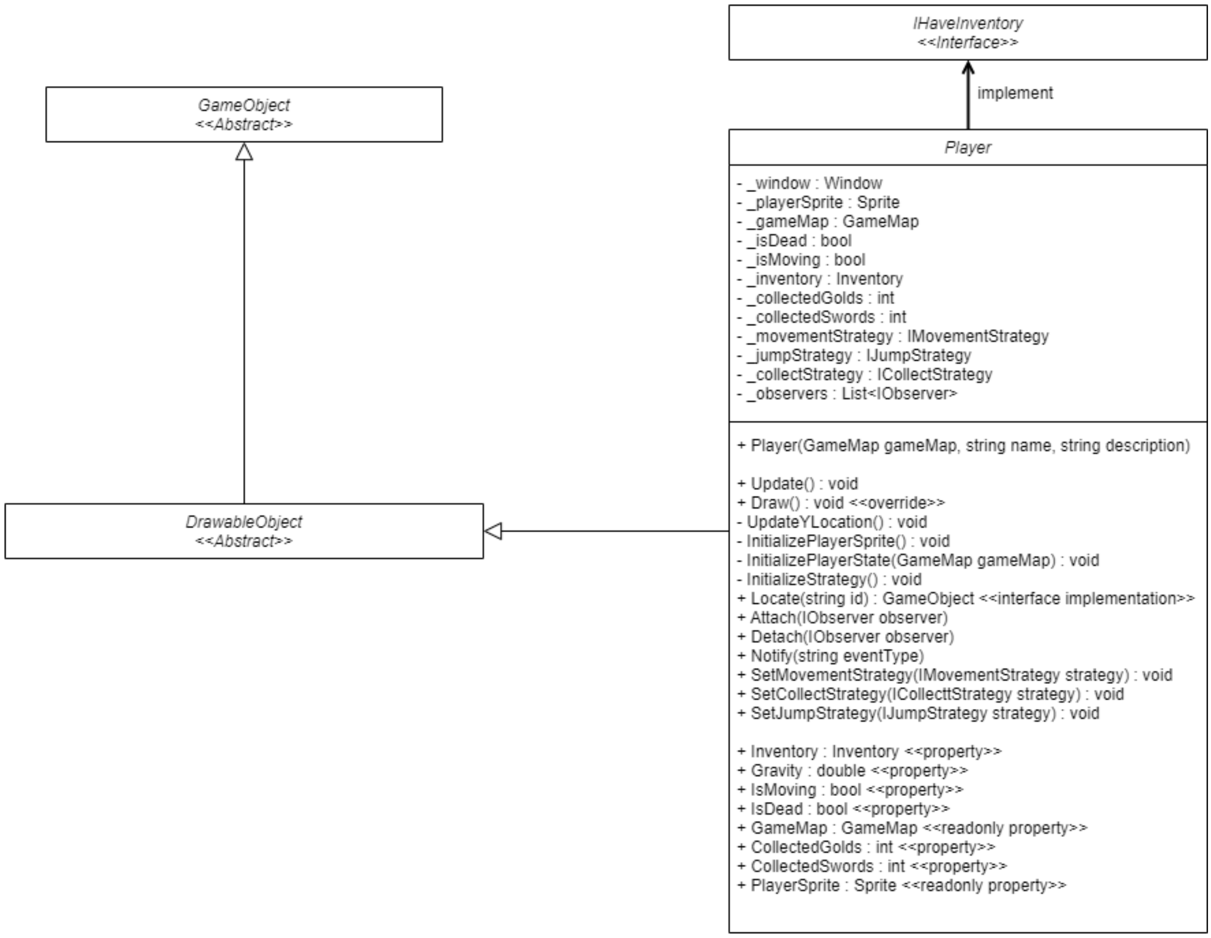
- **GameMap:**


GameMap
- _mapWidth : int
- _mapHeight : int
- _background : Bitmap
- _jungle : Bitmap
- _sky : Bitmap
- _sea : Bitmap
- _lands : List<Land>
- _bombs : List<CollectibleBomb>
- _golds : List<CollectibleGold>
- _sword : List<Sword>
- _speedPotion : List<SpeedPotion>
- _jumpPotion : List<JumpPotion>
- _txtMap : string[]
+ GameMap(string[] ids, string name, string description)
- ReadMapFromFile(string filename) : void
- LoadBackgrounds() : void
+ SetUpGameMap() : void
+ Draw() : void
+ Width : int <<readonly property>>
+ Height : int <<readonly property>>
+ Lands : List<Land> <<readonly property>>
+ Bombs : List<CollectibleBomb> <<readonly property>>
+ Golds : List<CollectibleGold> <<readonly property>>
+ Swords : List<Sword> <<readonly property>>
+ Background : Bitmap <<property>>
+ Sea : Bitmap <<readonly property>>
+ Jungle : Bitmap <<readonly property>>
+ Sky : Bitmap <<readonly property>>
IdentifiableObject
- _identifiers : List<string>
+ IdentifiableObject (string[] idents)
+ AreYou (string id) : bool
+ FirstId () : string
GameObject
<<Abstract>>
- _description : string
- _name : string
+ GameObject (string[] ids, string name, string desc)
+ Name : string <<readonly property>>
+ ShortDescription : string <<readonly property>>
+ FullDescription : string <<readonly property>>
DrawableObject
<<Abstract>>
# _xLocation : double
# _yLocation : double
# _image : Bimap
+ DrawableObject(double xLocation, double yLocation
, string[] ids, string name, string description, string imagePath)
+ Draw() : void <<virtual>>
+ X: double <<properties>>
+ Y: double <<properties>>
+ Image : Bitmap <<readonly properties>>
*
JumpPotion
+ SpeedPotion(double xLocation, double yLocation)
: base(xLocation, yLocation, new string[] { "speed potion" },
"Speed Potion!!", "Increase speed !!", "speedpotion.png")
*
Land
+ Land(double xLocation, double yLocation)
: base(xLocation, yLocation, new string[] { "" }, "", "", "land.png")
Player
JumpPotion
+ JumpPotion(double xLocation, double yLocation) :
base(xLocation, yLocation, new string[] { "jump potion" },
"Jump Potion!!", "Increase jump !!", "jumppotion.png")
*
CollectibleBomb
+ CollectibleBomb(double xLocation, double yLocation)
: base(xLocation, yLocation, new string[] { "bomb" },
"Bombs!!", "Do not touch the bombs!", "Bomb.png")
*
CollectibleGold
+ CollectibleGold(double xLocation, double yLocation)
: base(xLocation, yLocation, new string[] { "gold" },
"Golds!!", "Collect golds to win the game!!", "Gold.png")
*
Sword
+ Sword(double xLocation, double yLocation)
: base(xLocation, yLocation, new string[] { "sword" },
"Metal Sword!", "Meelee weapons!", "sword.png")
*

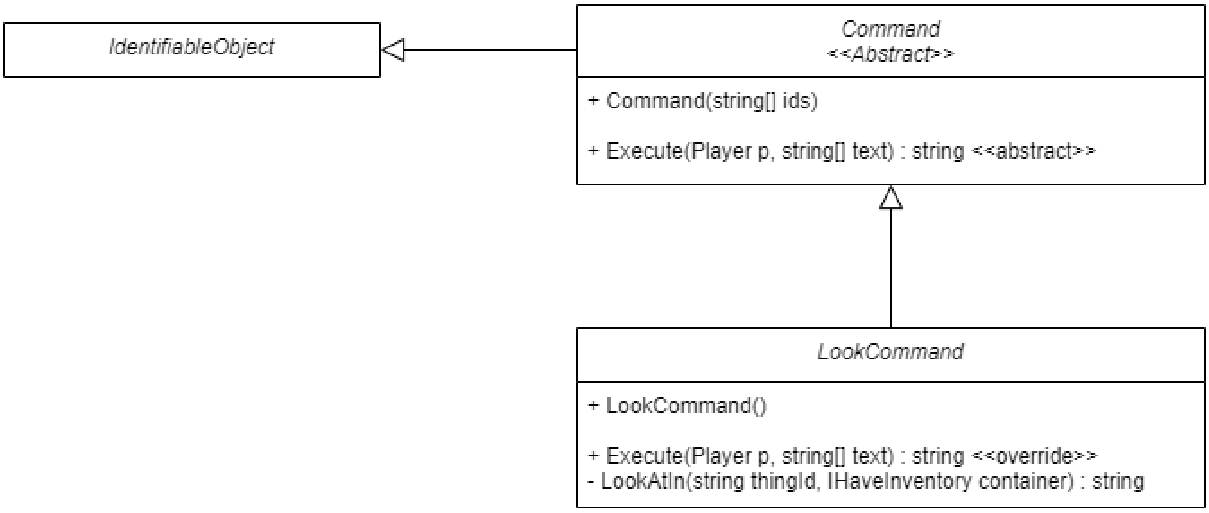
- **Inventory:**


Inventory
- _items : List<Item>
+ Inventory()
+ HasItem(string id) : bool
+ Put(Item itm) : void
+ Fetch(string id) : Item
+ ItemList : string <<readonly property>>
Item
+ Item(string[] idents, string name, string description)
uses
GameObject
<<Abstract>>
uses
IHaveInventory
<<Interface>>
Locate(string id) : GameObject
+ Name : string <<property>>

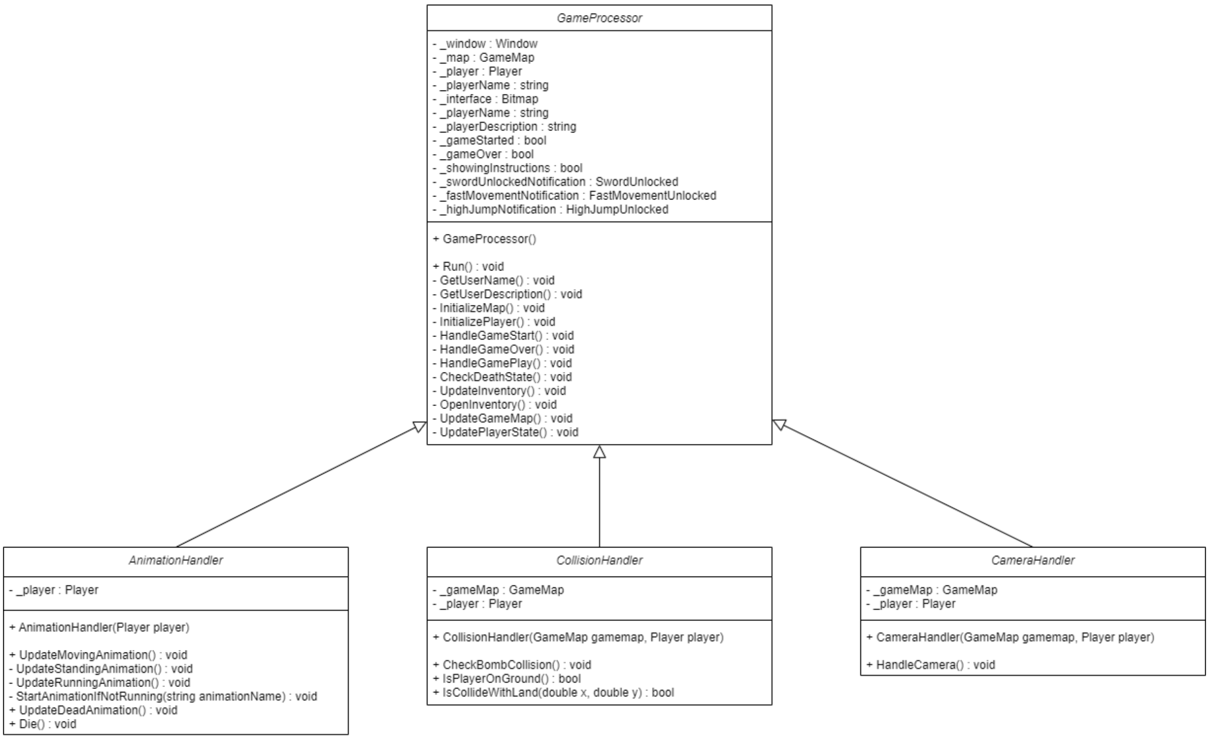
- **Player:**


IHaveInventory
<<Interface>>
implement
GameObject
<<Abstract>>
Player
- _window : Window
- _playerSprite : Sprite
- _gameMap : GameMap
- _isDead : bool
- _isMoving : bool
- _inventory : Inventory
- _collectedGolds : int
- _collectedSwords : int
- _movementStrategy : IMovementStrategy
- _jumpStrategy : IJumpStrategy
- _collectStrategy : ICollectStrategy
- _observers : List<IObserver>
+ Player(GameMap gameMap, string name, string description)
+ Update() : void
+ Draw() : void <<override>>
- UpdateYLocation() : void
- InitializePlayerSprite() : void
- InitializePlayerState(GameMap gameMap) : void
- InitializeStrategy() : void
+ Locate(string id) : GameObject <<interface implementation>>
+ Attach(IObserver observer)
+ Detach(IObserver observer)
+ Notify(string eventType)
+ SetMovementStrategy(IMovementStrategy strategy) : void
+ SetCollectStrategy(ICollecttStrategy strategy) : void
+ SetJumpStrategy(IJumpStrategy strategy) : void
+ Inventory : Inventory <<property>>
+ Gravity : double <<property>>
+ IsMoving : bool <<property>>
+ IsDead : bool <<property>>
+ GameMap : GameMap <<readonly property>>
+ CollectedGolds : int <<property>>
+ CollectedSwords : int <<property>>
+ PlayerSprite : Sprite <<readonly property>>
DrawableObject
<<Abstract>>

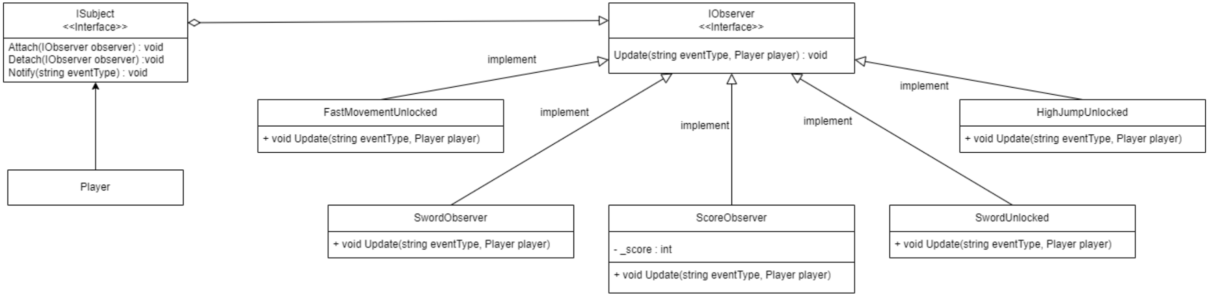
- **Command:**


IdentifiableObject
Command
<<Abstract>>
+ Command(string[] ids)
+ Execute(Player p, string[] text) : string <<abstract>>
LookCommand
+ LookCommand()
+ Execute(Player p, string[] text) : string <<override>>
- LookAtIn(string thingId, IHaveInventory container) : string

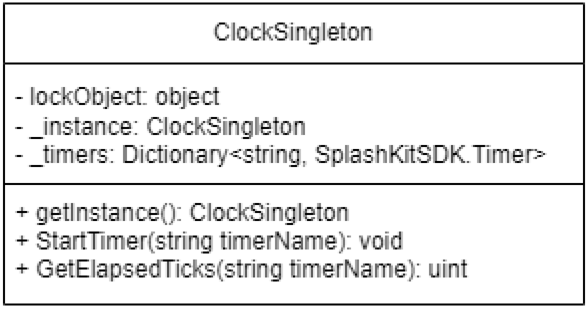
- **GameProcessor:**


GameProcessor
- _window : Window
- _map : GameMap
- _player : Player
- _playerName : string
- _interface : Bitmap
- _playerName : string
- _playerDescription : string
- _gameStarted : bool
- _gameOver : bool
- _showingInstructions : bool
- _swordUnlockedNotification : SwordUnlocked
- _fastMovementNotification : FastMovementUnlocked
- _highJumpNotification : HighJumpUnlocked
+ GameProcessor()
+ Run() : void
- GetUserName() : void
- GetUserDescription() : void
- InitializeMap() : void
- InitializePlayer() : void
- HandleGameStart() : void
- HandleGameOver() : void
- HandleGamePlay() : void
- CheckDeathState() : void
- UpdateInventory() : void
- OpenInventory() : void
- UpdateGameMap() : void
- UpdatePlayerState() : void
AnimationHandler
- _player : Player
+ AnimationHandler(Player player)
+ UpdateMovingAnimation() : void
- UpdateStandingAnimation() : void
- UpdateRunningAnimation() : void
- StartAnimationIfNotRunning(string animationName) : void
+ UpdateDeadAnimation() : void
+ Die() : void
CollisionHandler
- _gameMap : GameMap
- _player : Player
+ CollisionHandler(GameMap gamemap, Player player)
+ CheckBombCollision() : void
+ IsPlayerOnGround() : bool
+ IsCollideWithLand(double x, double y) : bool
CameraHandler
- _gameMap : GameMap
- _player : Player
+ CameraHandler(GameMap gamemap, Player player)
+ HandleCamera() : void

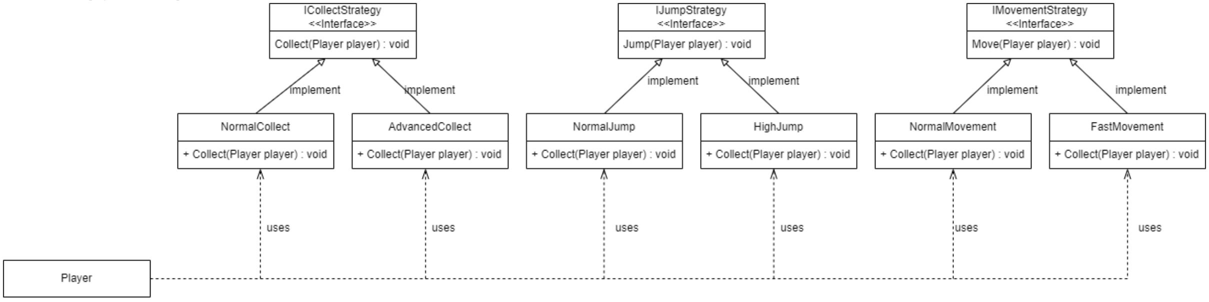
- **Observers Design:**


ISubject
<<Interface>>
Attach(IObserver observer) : void
Detach(IObserver observer) :void
Notify(string eventType) : void
Player
IObserver
<<Interface>>
Update(string eventType, Player player) : void
implement
implement
implement
implement
implement
FastMovementUnlocked
+ void Update(string eventType, Player player)
HighJumpUnlocked
+ void Update(string eventType, Player player)
SwordObserver
+ void Update(string eventType, Player player)
ScoreObserver
- _score : int
+ void Update(string eventType, Player player)
SwordUnlocked
+ void Update(string eventType, Player player)

- **ClockSingleton:**

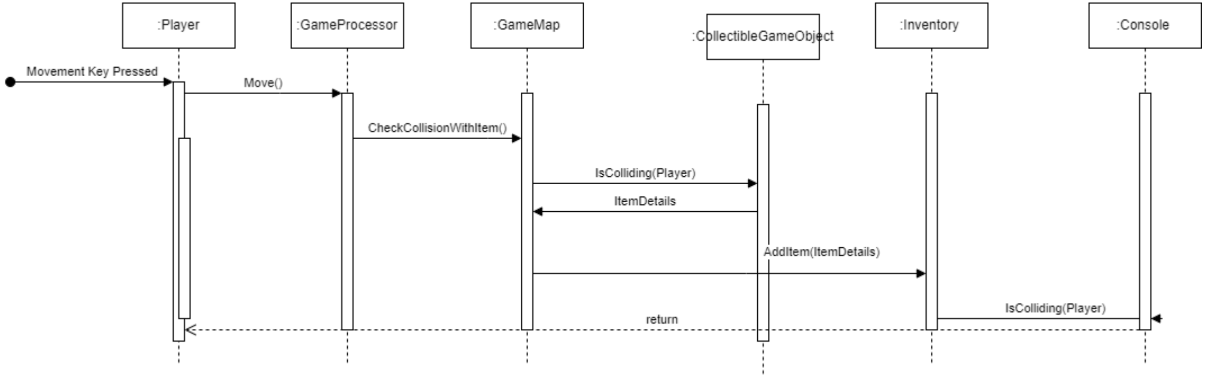

ClockSingleton
- lockObject: object
- _instance: ClockSingleton
- _timers: Dictionary<string, SplashKitSDK.Timer>
+ getInstance(): ClockSingleton
+ StartTimer(string timerName): void
+ GetElapsedTicks(string timerName): uint

- **StrategyDesign:**

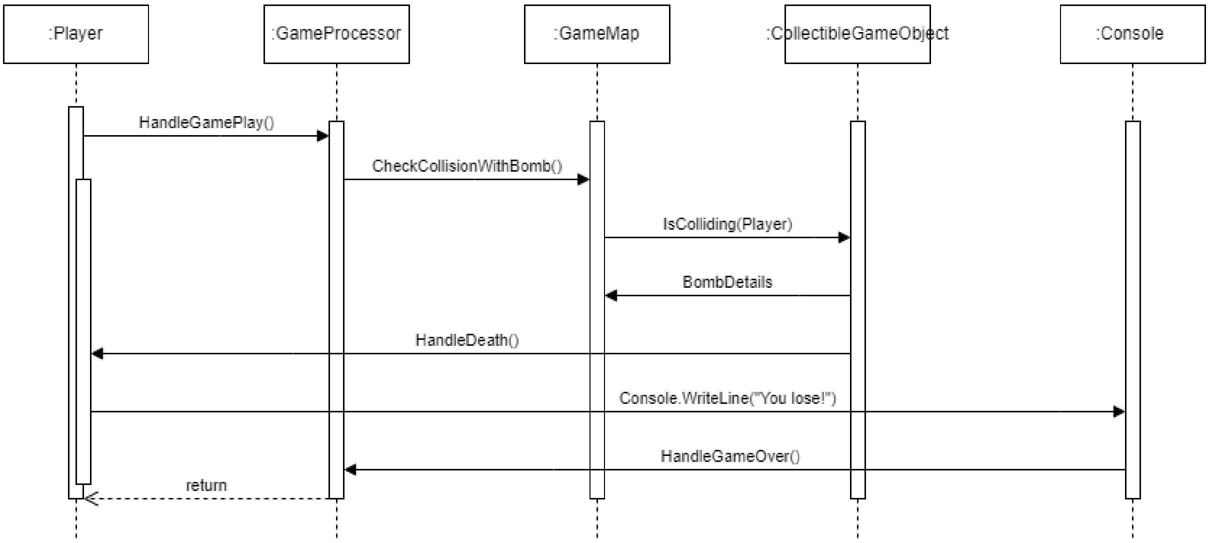

ICollectStrategy
<<Interface>>
Collect(Player player) : void
IJumpStrategy
<<Interface>>
Jump(Player player) : void
IMovementStrategy
<<Interface>>
Move(Player player) : void
implement
implement
implement
implement
implement
implement
NormalCollect
+ Collect(Player player) : void
AdvancedCollect
+ Collect(Player player) : void
NormalJump
+ Collect(Player player) : void
HighJump
+ Collect(Player player) : void
NormalMovement
+ Collect(Player player) : void
FastMovement
+ Collect(Player player) : void
uses
uses
uses
uses
uses
uses
Player

Updated Sequence Diagram

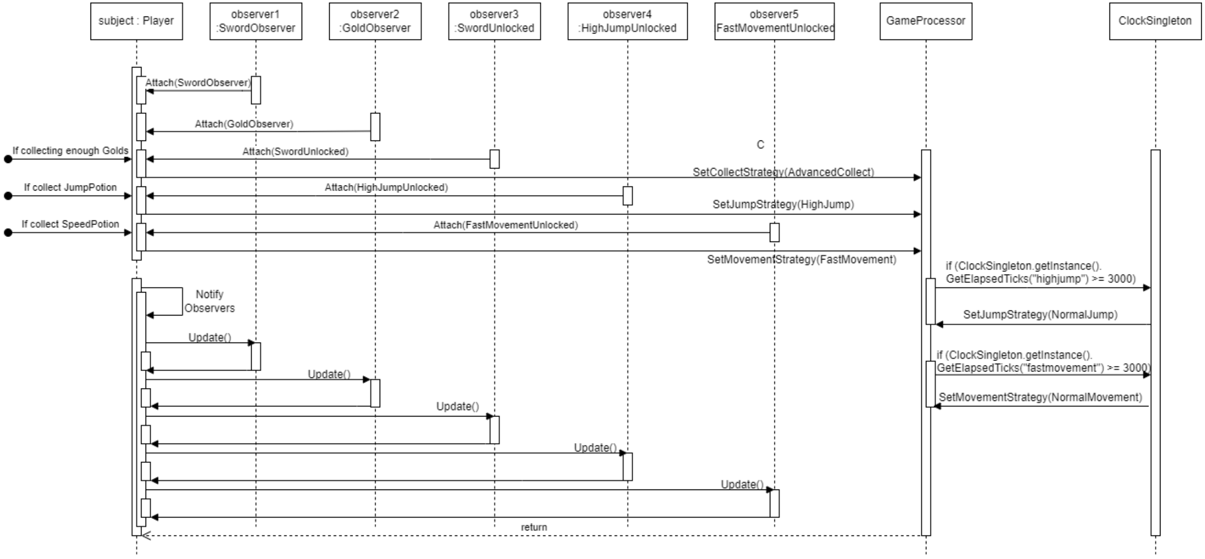
- **Collecting Items:**



- **Hit Bomb:**



- **Player State and Notifications:**


subject : Player
observer1
:SwordObserver
observer2
:GoldObserver
observer3
:SwordUnlocked
observer4
:HighJumpUnlocked
observer5
FastMovementUnlocked
GameProcessor
ClockSingleton
Attach(SwordObserver)
Attach(GoldObserver)
If collecting enough Golds
Attach(SwordUnlocked)
C
SetCollectStrategy(AdvancedCollect)
If collect JumpPotion
Attach(HighJumpUnlocked)
SetJumpStrategy(HighJump)
If collect SpeedPotion
Attach(FastMovementUnlocked)
SetMovementStrategy(FastMovement)
if (ClockSingleton.getInstance().
GetElapsedTicks("highjump") >= 3000)
SetJumpStrategy(NormalJump)
if (ClockSingleton.getInstance().
GetElapsedTicks("fastmovement") >= 3000)
SetMovementStrategy(NormalMovement)
Notify
Observers
Update()
Update()
Update()
Update()
Update()
return