Assignment 3

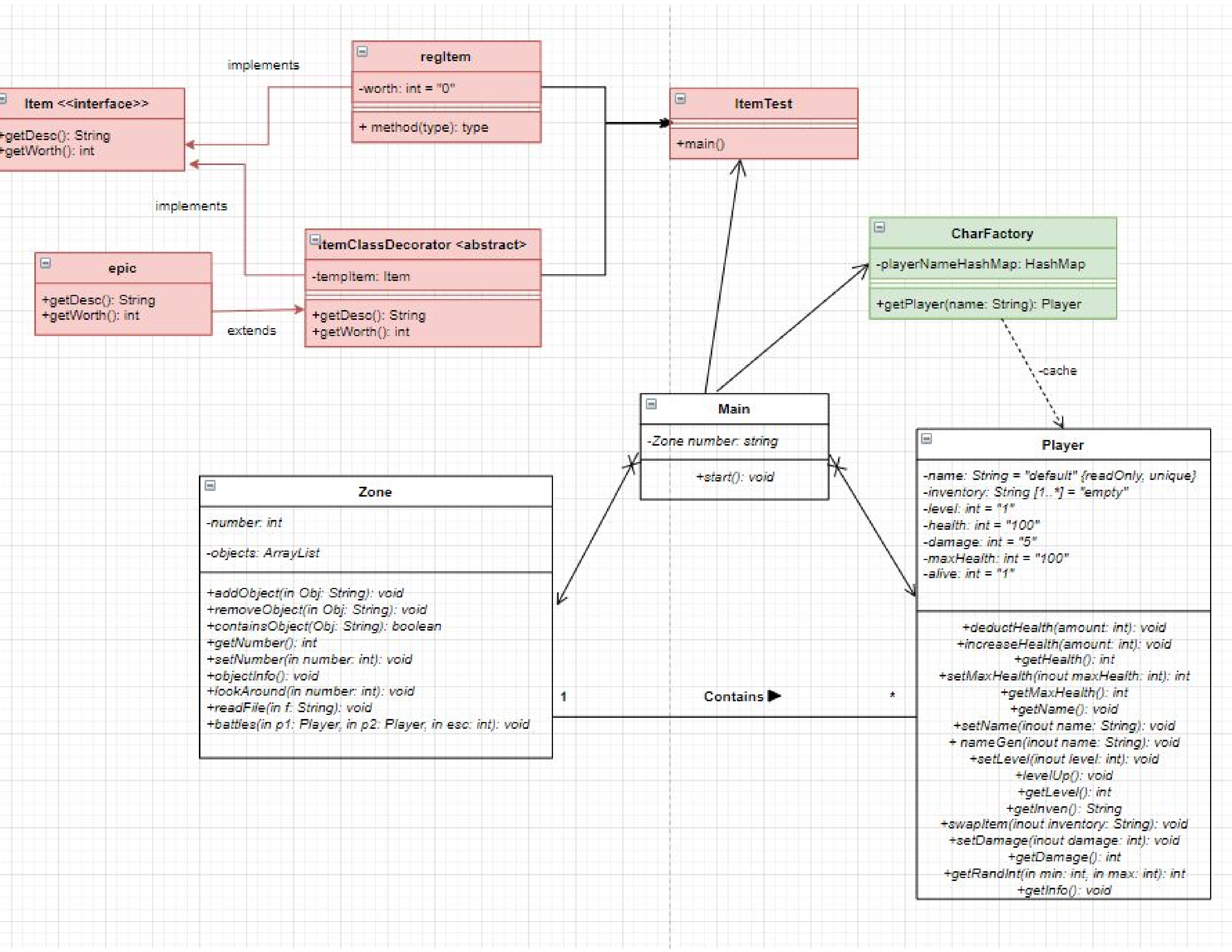
Name: Tjun Tji Oon (vuID: 2675214)

Modelling tool: draw.io

[The jar file for this program is in the classes directory, please open the terminal in that directory and enter java -jar main.jar to execute and play the game!]

Summary of changes

- Added charFactory class
    - Flyweight design pattern applied
- Added Item interface and other item classes
    - Decorator design pattern applied

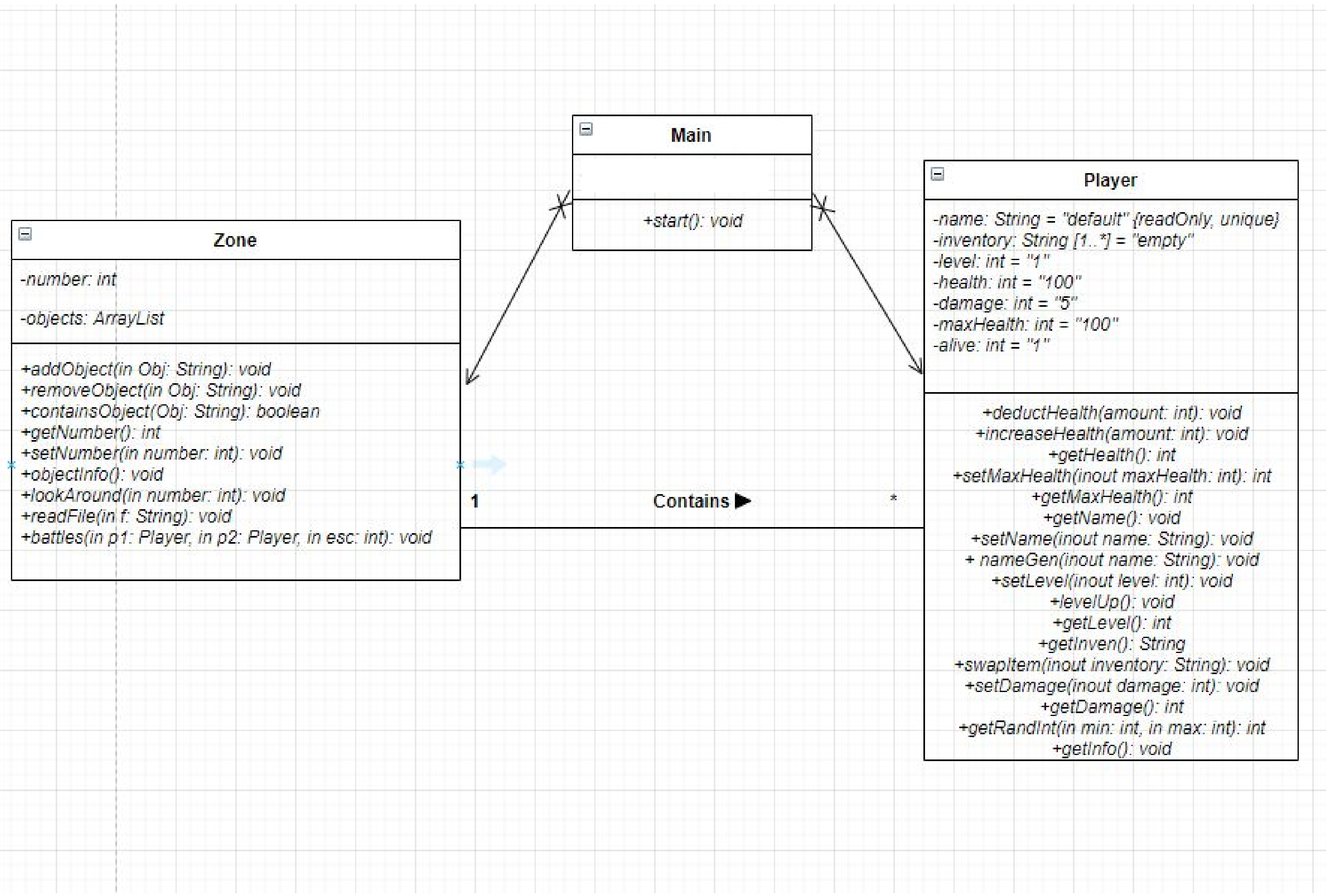


Item <<interface>>
+getDesc(): String
+getWorth(): int
implements
regItem
-worth: int = "0"
+ method(type): type
ItemTest
+main()
implements
epic
+getDesc(): String
+getWorth(): int
extends
itemClassDecorator <abstract>
-tempItem: Item
+getDesc(): String
+getWorth(): int
CharFactory
-playerNameHashMap: HashMap
+getPlayer(name: String): Player
-cache
Main
-Zone number: string
+start(): void
Zone
-number: int
-objects: ArrayList
+addObject(in Obj: String): void
+removeObject(in Obj: String): void
+containsObject(Obj: String): boolean
+getNumber(): int
+setNumber(in number: int): void
+objectInfo(): void
+lookAround(in number: int): void
+readFile(in f: String): void
+battles(in p1: Player, in p2: Player, in esc: int): void
Player
-name: String = "default" {readOnly, unique}
-inventory: String [1..*] = "empty"
-level: int = "1"
-health: int = "100"
-damage: int = "5"
-maxHealth: int = "100"
-alive: int = "1"
+deductHealth(amount: int): void
+increaseHealth(amount: int): void
+getHealth(): int
+setMaxHealth(inout maxHealth: int): int
+getMaxHealth(): int
+getName(): void
+setName(inout name: String): void
+ nameGen(inout name: String): void
+setLevel(inout level: int): void
+levelUp(): void
+getLevel(): int
+getInven(): String
+swapItem(inout inventory: String): void
+setDamage(inout damage: int): void
+getDamage(): int
+getRandInt(in min: int, in max: int): int
+getInfo(): void
1
Contains ▶
*

ID	**DP1**
Design Pattern	Flyweight
Problem	Creating new objects in large numbers is very time and memory expensive. In this project's context, if the game designer wanted to add more items/characters and the amount got extremely large, the game would create a new object for each and every one. This uses up a lot of space in memory and takes longer to do as opposed to if the design pattern was applied.
Solution	The Flyweight design pattern solves this problem by storing existing objects in a hashmap, which is used for the creation of new objects. When the creation process begins, the client asks the factory class if an object with a shared value exists (the key of the hashmap) if it does, then the class returns that same object for reuse. If it doesn't then a new object is created by the factory class and returned (and also stored in the hashmap). In the game's context, an example for a shared value that could be checked would be a character's name, so that if a game designer wanted to add 100,000 characters named 'zombie' this flyweight design pattern would apply and save memory and time.
Intended use	As of the current version, the use of this design pattern will not save much memory or time simply because there is only a small number of characters and items in the game. However, if a designer wanted to expand the game and include a situation where the player has to fight many generic characters, this design pattern could be used.
Constraints	This design pattern reduces the granularity of each object slightly, reducing flexibility and making each object immutable.
Additional remarks	

ID	**DP1**
Design Pattern	Decorator
Problem	Items in a game (including this one) are such an abstract concept as they cover such a wide variety of objects and functionality. For example, they can be usable or unusable, add damage, add resistances, be wearable etc. As such, if a game were to have many different types of items, then programming all possible functionalities into one class would make the code very confusing, messy and give objects of that class behaviour/operations that it would never need.
Solution	Implementing the decorator design pattern could add behaviour to objects without changing the behaviour of other objects of the same type. This is because behaviour is added by a separate decorator class which can be created ad hoc and tailored to the exact situation that calls for it.
Intended use	In this game for example, this also could be used to deal with the wide variety of items problem. If a designer wanted to add a specific new class of items based on rarity, say ‘epic’ items and designate a few items to this rarity without having other items have an isEpic() method, then they could use the decorator alongside a new ‘epic’ class to do so.
Constraints	Bloats the number of classes, complicates the instantiation of objects and an error in one of the extra classes could lead to a chain of errors in the others.
Additional remarks	

<u>Class diagram</u>


Main
+start(): void
Player
-name: String = "default" {readOnly, unique}
-inventory: String [1..*] = "empty"
-level: int = "1"
-health: int = "100"
-damage: int = "5"
-maxHealth: int = "100"
-alive: int = "1"
+deductHealth(amount: int): void
+increaseHealth(amount: int): void
+getHealth(): int
+setMaxHealth(inout maxHealth: int): int
+getMaxHealth(): int
+getName(): void
+setName(inout name: String): void
+ nameGen(inout name: String): void
+setLevel(inout level: int): void
+levelUp(): void
+getLevel(): int
+getInven(): String
+swapItem(inout inventory: String): void
+setDamage(inout damage: int): void
+getDamage(): int
+getRandInt(in min: int, in max: int): int
+getInfo(): void
Zone
-number: int
-objects: ArrayList
+addObject(in Obj: String): void
+removeObject(in Obj: String): void
+containsObject(Obj: String): boolean
+getNumber(): int
+setNumber(in number: int): void
+objectInfo(): void
+lookAround(in number: int): void
+readFile(in f: String): void
+battles(in p1: Player, in p2: Player, in esc: int): void
1
Contains ▶
*

The three classes represented in this class diagram are the zone, main and player classes. The zone class allows for the instantiation of each zone in the game. Depending on the numeric zone number argument passed as the constructor parameter, the zone will add the appropriate objects to an arraylist named ‘objects’ which is shown as the 2nd attribute of the zone class in the diagram above.

The main class has no attributes as its main purpose is to be the bridge between the two other classes as well as the engine. However even though it has no attributes and only one operation (method) it is still crucial to the running of the game. This is because that method is responsible for creating the world (i.e creating all the zone and player instances) and for taking and parsing user input.

The final class is the player class, which allows for the creation of one or more player objects to populate the zones. This brings us to the symbols in the class diagram, with the ‘contains’ relationship between player and zone showing that one zone contains any amount of players in it. The binary association between both zone and player with main and the non-navigability cross symbol attached indicates that the main class can access both other class’ public operations and attributes, however the relationship is not bidirectionally navigable. The primary reason for this is that there is simply no need.

A short description of each attribute and operation is attached below:

Zone class:

- *Number:* This is the zone number used by the main class to firstly populate each zone by passing an integer into the zone constructor, and secondly to distinguish each zone from each other.

- *Objects:* This arraylist is used to store information on the items/objects the player can interact with in each zone. For example, in zone 2 there is an apple tree etc. It is dynamically updated upon a sequence of events (this will be demonstrated via sequence diagram later on)

- *addObject(String object):* This method simply takes a string as an argument and adds that string to the objects attribute.

- *removeObject(String object):* This removes the argument from the objects attribute IF it contained it previously.

- *containsObject(String object):* This operation checks to see if the objects arraylist currently contains an object.

- *getNumber():* returns the number of the zone.

- *objectInfo():* prints the objects arraylist.

- *lookAround():* reads a file with a description of the zone the player is in and concatenates it with the objects arraylist and prints it out.

- *readFile(String fileName):* Scans a file and prints out the contents.

- *battles(Player one, Player two, int escape):* Launches the battle sequence between two players.

<u>Player class</u>

- *Name:* The name attribute of a character, used to distinguish one from another, also used as an argument for the battles method in the zone.

- *Inventory:* A player's inventory can only hold one item at a time due to the players being animals (they can only use their mouths to store items), so it is just a string rather than another data structure.

- *Health:* Stores the player's health, used as a loop condition in the battles method and for other health deducting/increasing activities in the game.

- *maxHealth:* Stores the maxhealth value for each player, so if they level up or get their health restored to full there is a preset value for it. It is a variable because it is based on level and levels change throughout the game.

- *Level:* Stores the player's level, used to determine their current maxHealth and damage.

- *Damage:* Damage values determine the range of damages that a player can do in 1 attack turn in the battle sequence.

- *Alive:* Alive status represents whether the player is alive and thus able to interact/be interacted with or dead.

- *deductHealth(int amount):* Deducts the amount of health specified in the parameter from a player.

- *increaseHealth(int amount):* Increases the amount of health specified in the parameter for a player.

- *getHealth():* returns the health value for the player- useful for when the user wants to check the amount of health they have before engaging in a battle for example.

- *setMaxHealth(int max):* Used to set the max health of a player for easy configuration and maintainability purposes.

- *getMaxHealth():* returns the maxHealth value for a player.

- *getName():* returns the name of a player.

- *setName(String name):* sets the name of a player to a certain string.

- *nameGen():* The opening sequence of the game, allows the player(user) to define their own name so other characters can reference it in the future. Creates immersion and personalization. This is done through a scanner within the method so no parameter is necessary.

- *setLevel(int l):* Sets the level to a specified amount.

- *levelUp():* Levels the character up by one, increasing their maxHealth and damage by a configurable amount and restores their health to full.

- *getInven():* Returns the item in the player's inventory.

- *swapItem(String item):* Overwrites the player's current inventory with the item argument, this method is used in the pick up command section in the parser located in the main class.

- *setDamage(int damage):* Sets the player's damage to a specified amount.

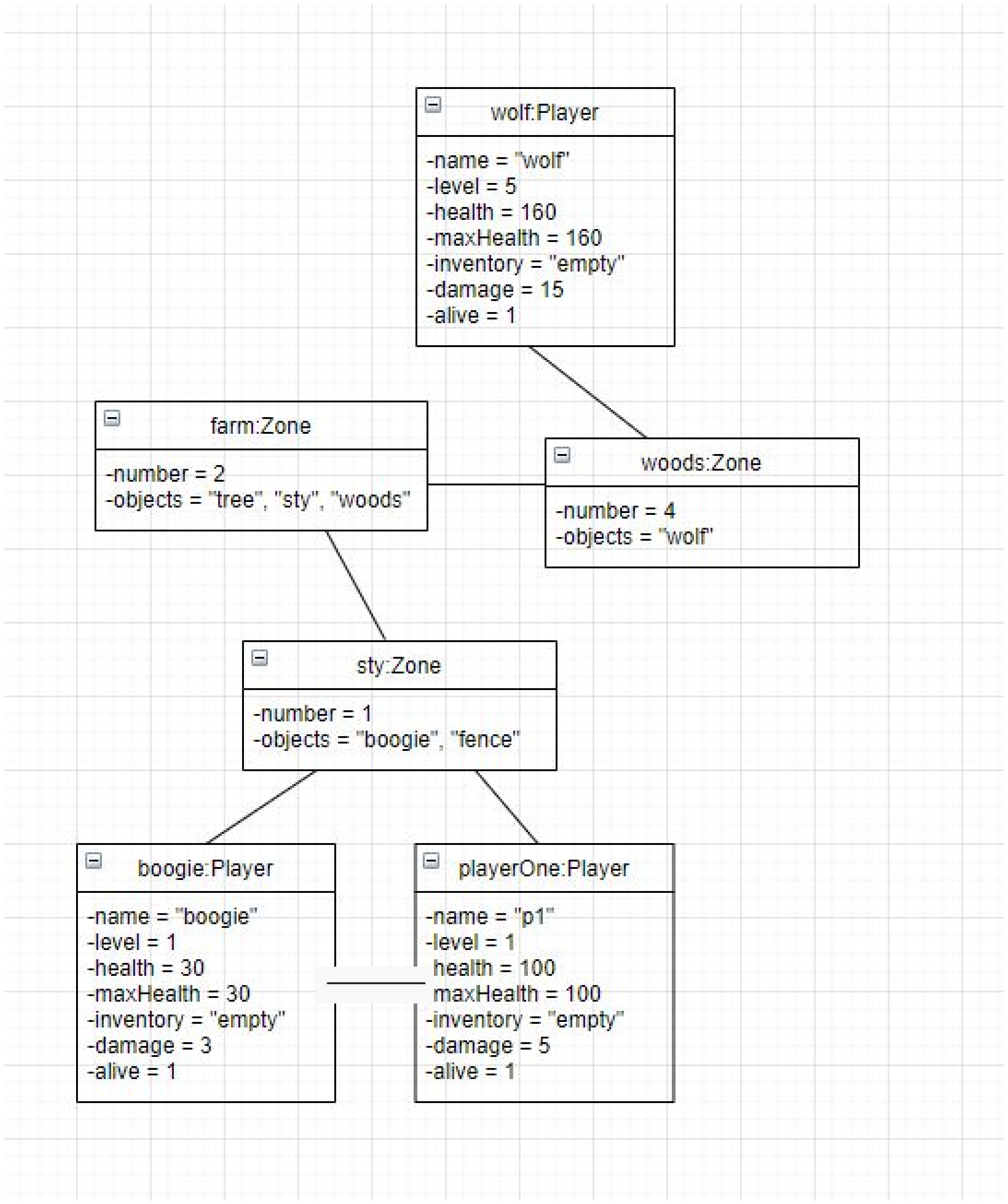
- *getDamage():* Returns the player's current damage.

- *getRandInt(int max, int min):* Generates a random integer between a maximum and minimum boundary, used to determine damage output in a given turn in a battle for both combatants.

Main class

The main class has only one private method, the start method, which is used to run the entire program until 'exit' is typed as a command (or ctrl+c is entered). This method contains the input parser, which decides what actions/operations to run after input is entered. In addition it is responsible for all the story elements for the game as well.

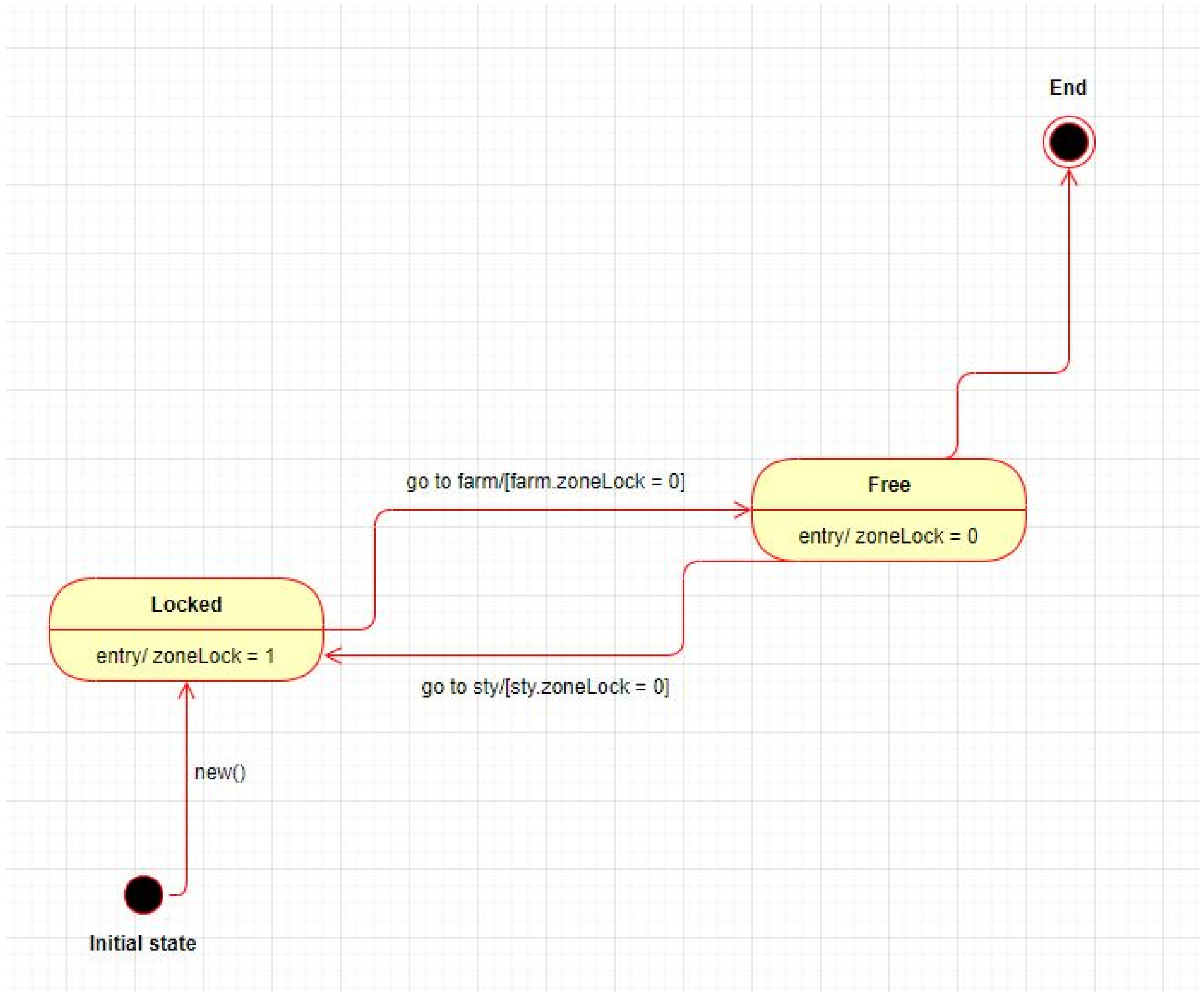
Object diagram



The object diagram above shows concrete instances of the zone and player classes in one snapshot of the game. In this case the diagram shows 3 zones and 3 players, the links between object imply associations. For example, this diagram can be interpreted that the zone ‘sty’ is associated with 2 players, boogie and playerOne. This could be explained by the fact that boogie is an npc that is located in sty, and playerOne (which is operated by the user) is currently located in sty as well. In addition, both player objects are also connected to one another, which represents the ability for one to interact with the other, perhaps in a class diagram this relationship would be called ‘talks to’. Also, multiple zones are interlinked because the design of this game involves moving from zone to zone and the object diagram illustrates that the player can move from sty to farm and farm to woods (as they are associated) but not from sty to woods for example.

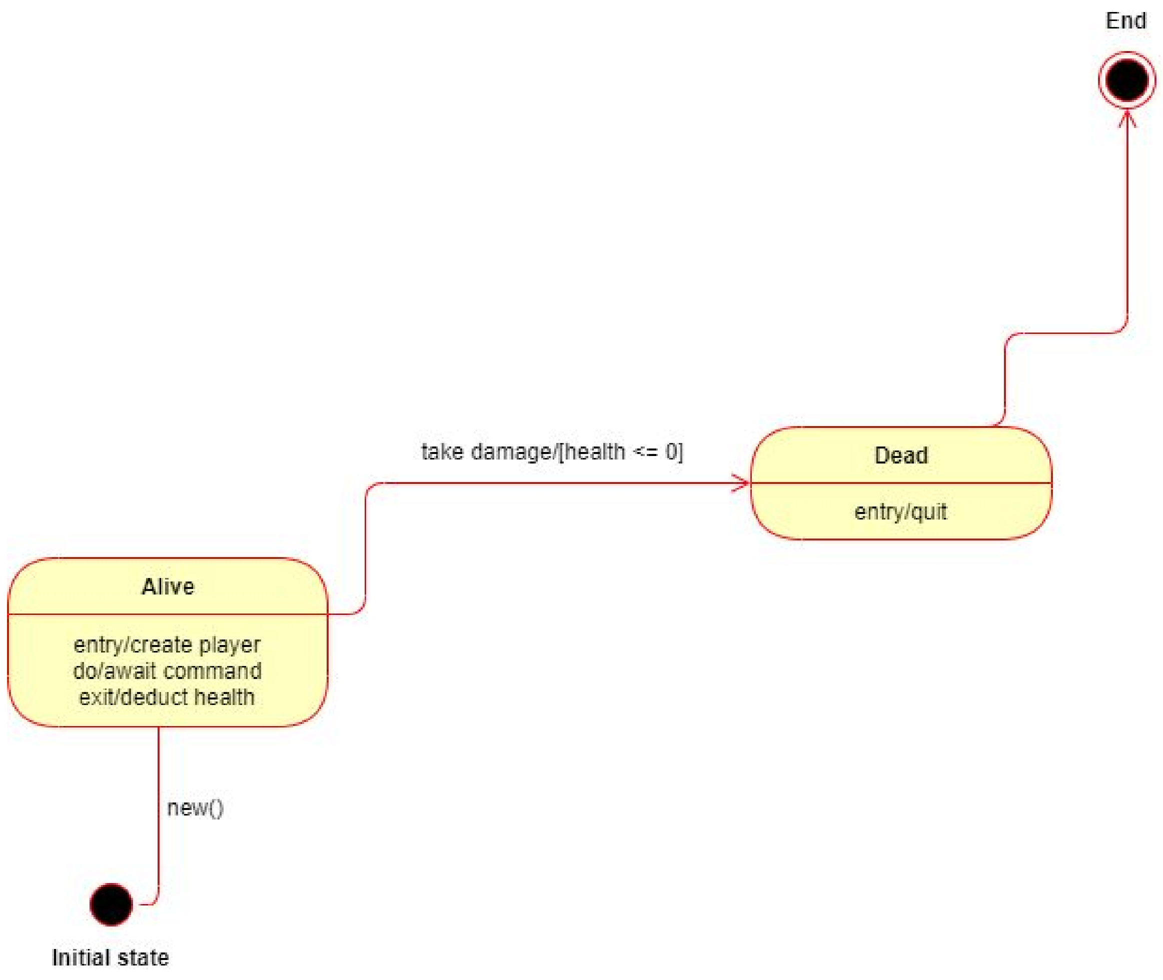
Furthermore, the object diagram shows only a snapshot of the system, which means it shows the internals of the system at one point in time. For example, this particular diagram shows the internals around the starting point of the game, where every character is still at full health and alive.

State machine diagram



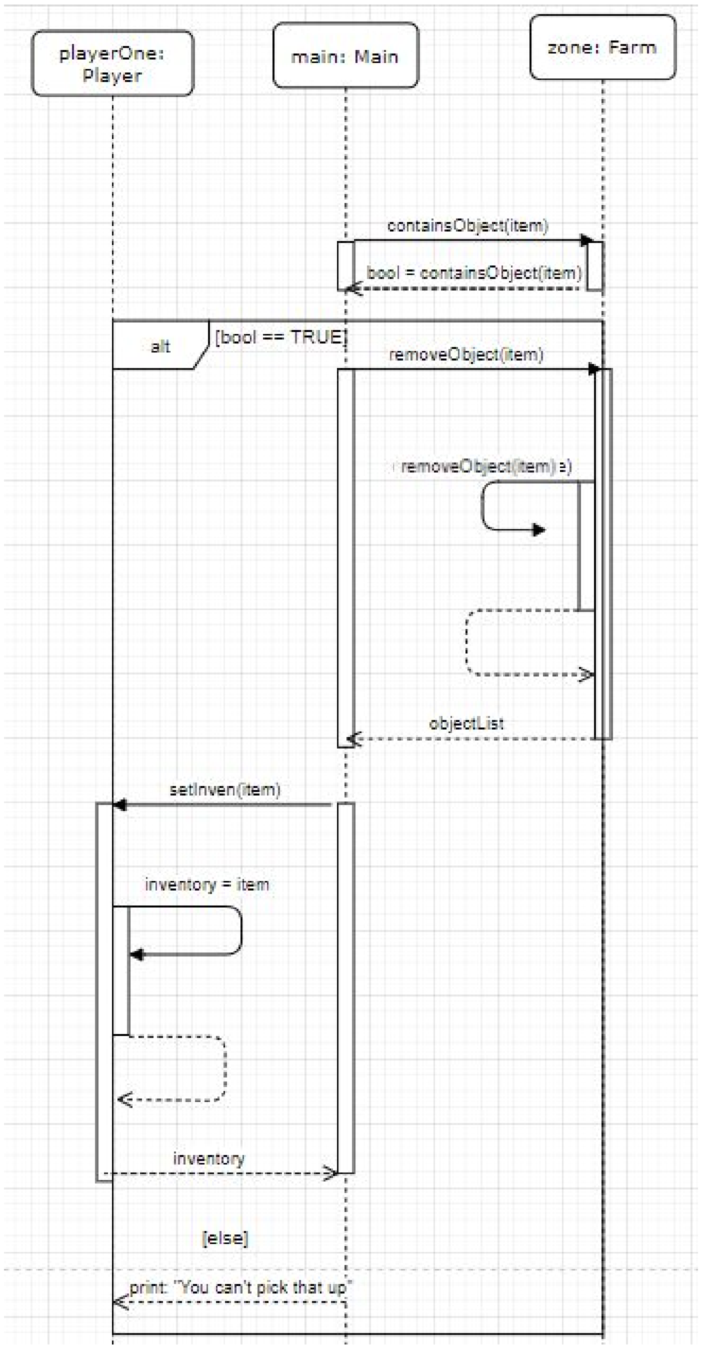
The state machine diagram above shows how the lock mechanisms for the sty and farm zones function. A zone’s lock variable stores either a 1 or a 0 which translates to it either being free or locked (as shown above). If a zone is locked, it cannot be accessed by the ‘go to’ command in the terminal, obviously with more zones in the game this state diagram would look more complicated but for now it is focused on just two. If a player goes to the sty, as the entry activity suggests, its zoneLock is changed to one, which means it is locked and the player can't go to it again (as they are already there). This state machine's initial state starts the player in the sty, so the zonelock for the sty is initially 1.

To transition between states, as described earlier, the player has to enter 'go to x' into the terminal with x being a different zone (because the same zone is locked!). This is the trigger event that causes a change in states in this state machine diagram, which is shown through the label of the transition. In addition, the guard condition is checked which in this case is whether farm's zonelock is 0 (farm being the zone that is being travelled to) and if this evaluates to true i.e the farm is free, then the player relocates to the farm (not shown in the diagram) and the sty's zonelock enters the free state which sets the zonelock to 0. This would allow the player in the farm to travel back to the sty as a similar guard condition would be used except the destination would be sty. This guard condition is shown as the label of the transition from free to locked.



This second state machine diagram shows two states, alive and dead, this refers to a character's life status (which is useful to know so that dead characters cannot be interacted with etc.). This time there is only one transition between states since there is no respawning in this game, and the event that triggers this transition is if the player takes damage at which point the guard condition is checked. If the guard evaluates to true then the transition takes place and the player enters the dead state, which quits the game. If it evaluates to false then the event is discarded and the player do/ activity continues to take place as the player is still in the alive state.

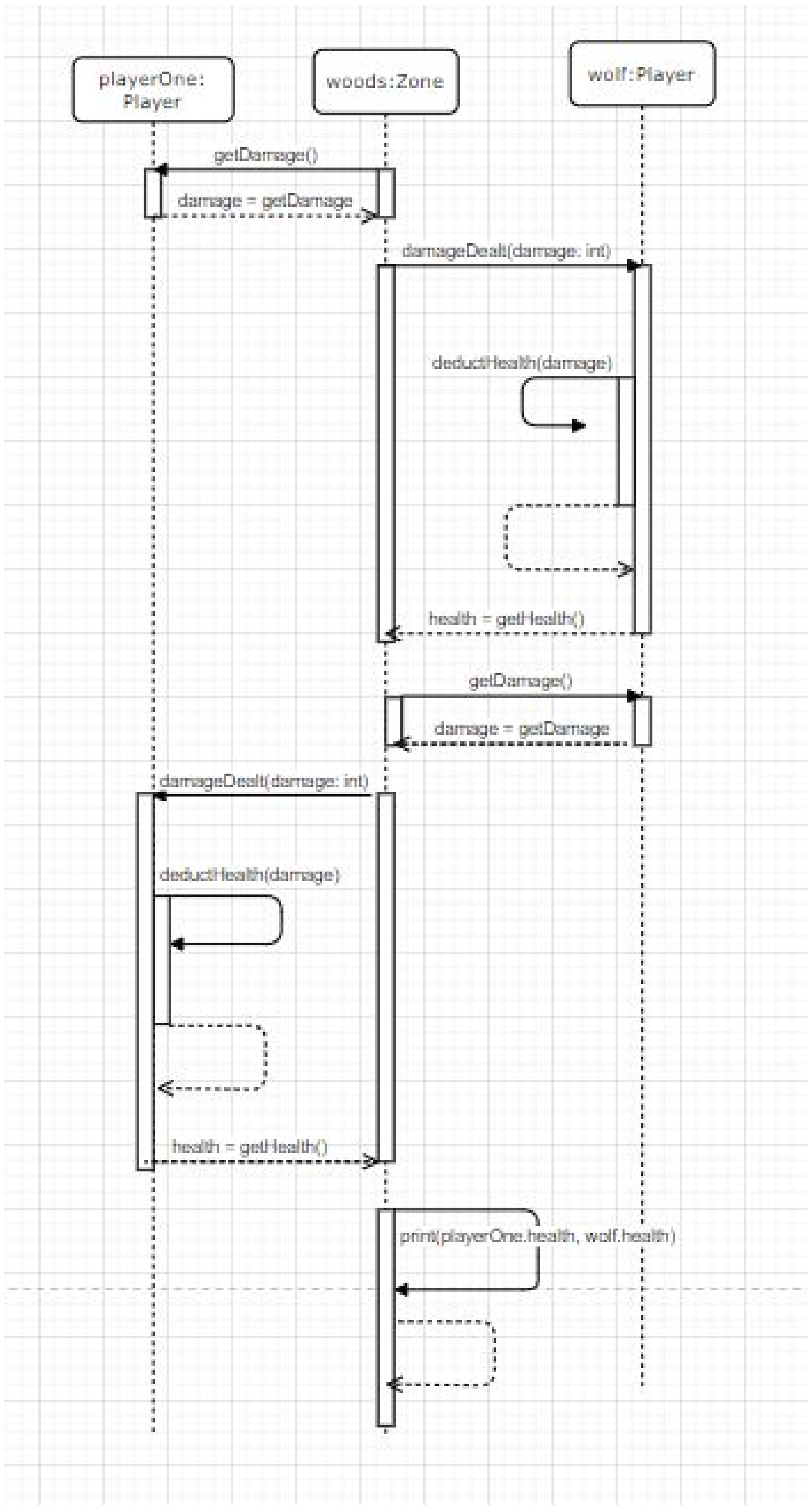
Sequence diagrams



This sequence diagram shows the interactions between all the classes when an event occurs. For this example, this sequence diagram shows how each object interacts with another when the player picks up an item.

Each node represents an object and the dashed vertical line below it is it's lifeline. When an item is picked up, the sequence's chronology starts in the main object as that is where the command 'pick up x' is parsed. From there main checks if the zone object contains the item through the containsObject method and the zone replies with either true or false, informing main whether its objects arrayList does indeed contain the item.

Next the sequence diagram shows an alt fragment, which if the reply from the zone evaluates to true, it performs the following operations of removing the item from the zone and returning the objectList to main. Then main tells the playerOne object to set its inventory to the item. If the boolean reply evaluates to false, main does not tell playerOne to set its inventory.



This sequence diagram shows what happens when the player uses an attack during a battle sequence. This time however since the battle method is in the zone class (this is so main can check if both combatants are in the zone) the coordinator/intermediary for these messages is the zone object. The sequence diagram depicts a battle between the player and the wolf.

It shows that initially the woods object gets the player's damage, sends it to the wolf object who processes it and deducts health equal to the player's damage and returns its health.

The same is done to the player with the wolf's damage and after that both combatant's health is printed. All this occurs after the player selects attack in the battle screen and it occurs once per turn.