JAdventure

Francis Côté-Tremblay (6615287)
Michal Wozniak (21941097)
Sebastian Rafique Proctor-Shah (29649727)
Illya Reutskyy (6418236)
Harrison Ianatchkov (6607403)
Simon Monière (6648568)

You must submit this milestone through **GitHub** and have uploaded the project's source code (-2 marks if you don't use GitHub) (https://github.com/francisct/JAdventure.git)

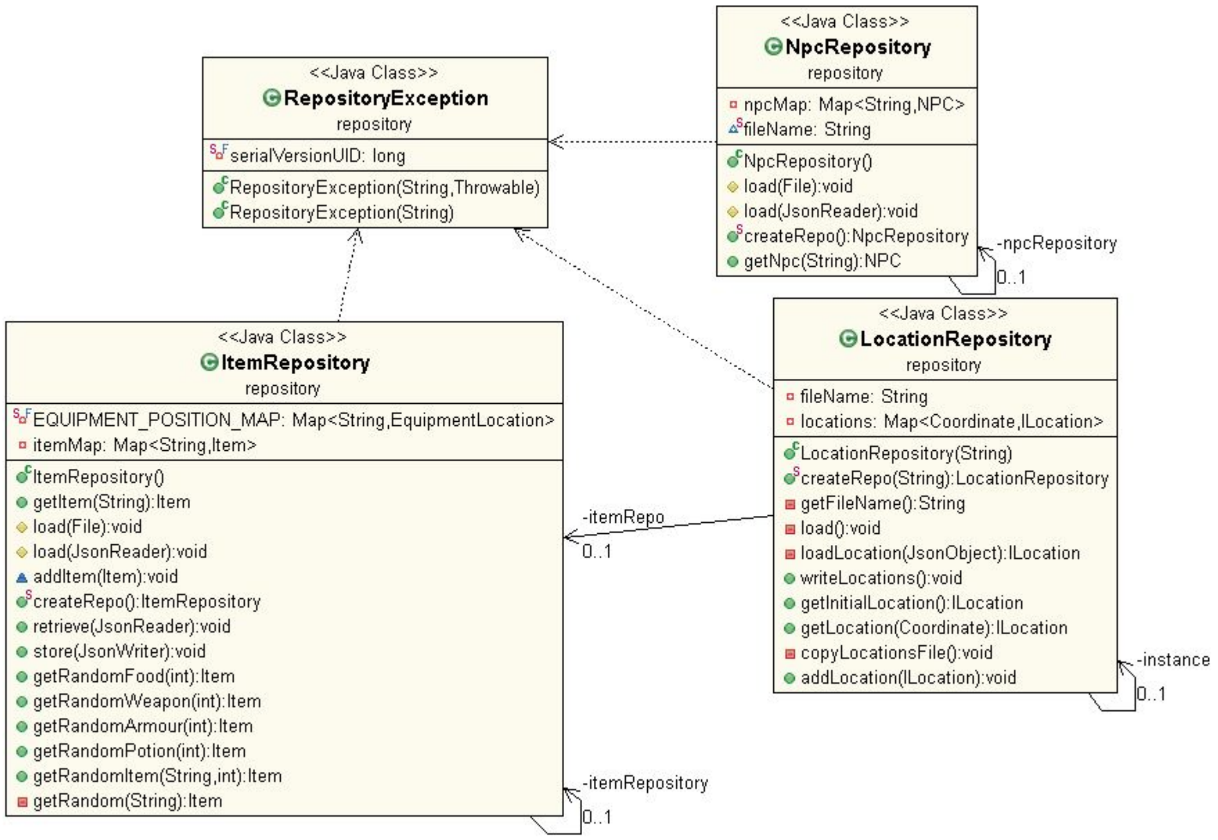
Summary of Project

JAdventure is a small single-player, text-based game engine written in Java. It comes with a sample adventure game that can be played. It is used to demonstrate the capabilities of the engine. As a text-based game, its algorithmic/technological side is fairly minimal and most of the code is concerned with defining the actual domain logic.

Class Diagram of Actual System

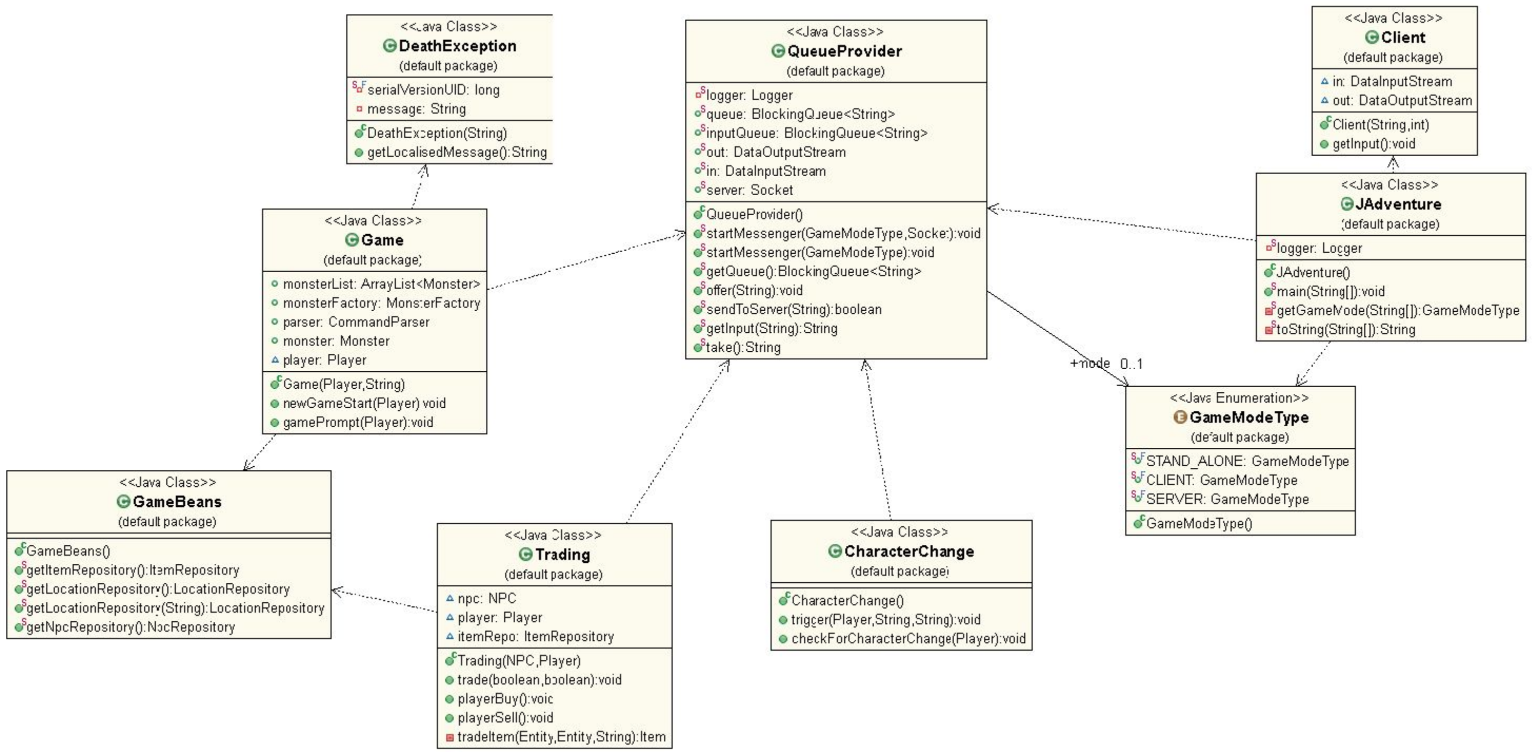
The following UML diagrams are done at the package level. This implies that the relationship that will be displayed in each diagram are self-contained within that package. There are relationships between those classes and others in different packages, but those were not shown due to the way the project was designed. Had those relationship been included, the uml would not have been understandable. Most of the classes have multiple dependencies with a lot of other classes. This means that it is difficult to keep track of all the relationship in the project because of this problem of high level of coupling

The Repository Package



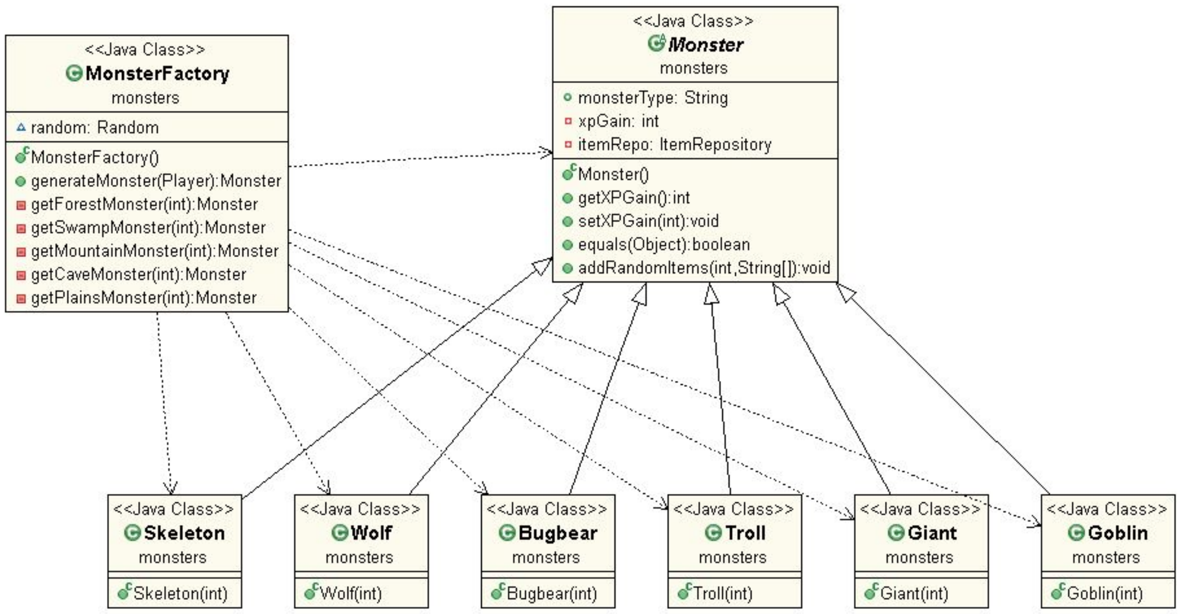
The repository package contains the classes that communicates with the JSON files that contain the different data used in the game. The ItemRepository class will open a file that contains the list of usable items in the game. This class can load and get Items that can then be used somewhere in the game and will be used to save the item of a certain game in a JSON file. The LocationRepository and the NpcRepository class do a similar thing but for the locations and npcs respectively. Each class can throw a RepositoryException in the situation where something goes wrong. Each of them also contains a private static instance of itself from which it can be deducted that it is or could be a class that uses the singleton pattern to prevent mutliple instances of itself.

The Default Package



The default package contains all the classes that are not part of a particular package. One of the important class in this package is the QueueProvider class. This class has the purpose of queuing up the text that will be displayed to the screen and, when it is the right time, print out this text to the user. The JAdventure class is the main class of the whole project. It is simply being used to create a MainMenu from where the player will be able to start the game. Another important class is the Game class where the user will input the data. The newGameStart function will ask the user whether he or she wants to start a game. The gamePrompt function will wait for input from the user and if that input is significant then the corresponding action will take place.

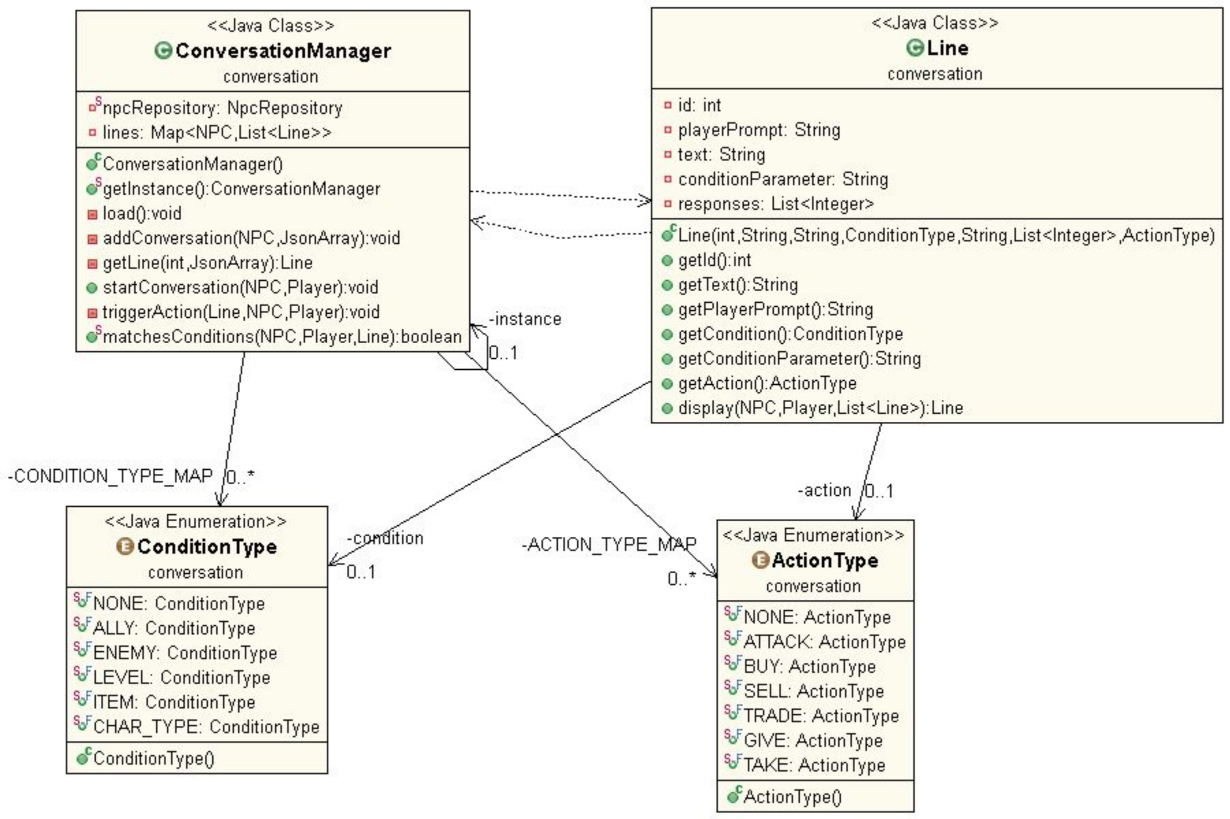
The Monster Package



The monster package is associated with the Player class, and contains the MonsterFactory class, the Monster class, and all the different types of monster classes. The monster factory is used to create instantiations of any type of monster. When the player lands on a location, the Game class asks the factory to generate a new monster for that particular location. The developers attempted to implement the factory pattern in the MonsterFactory class. Coupling is quite high

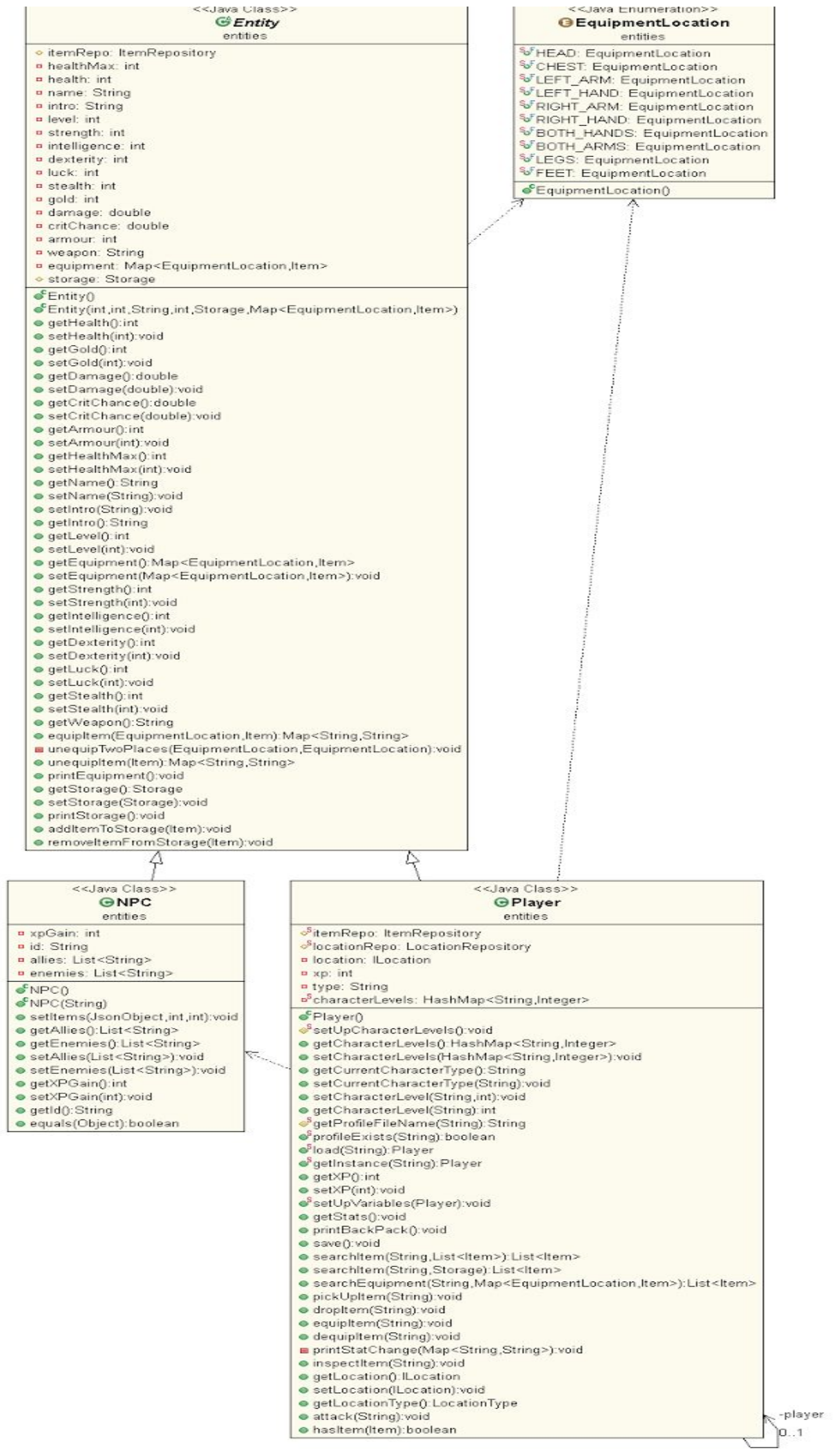
.

The Conversation Package



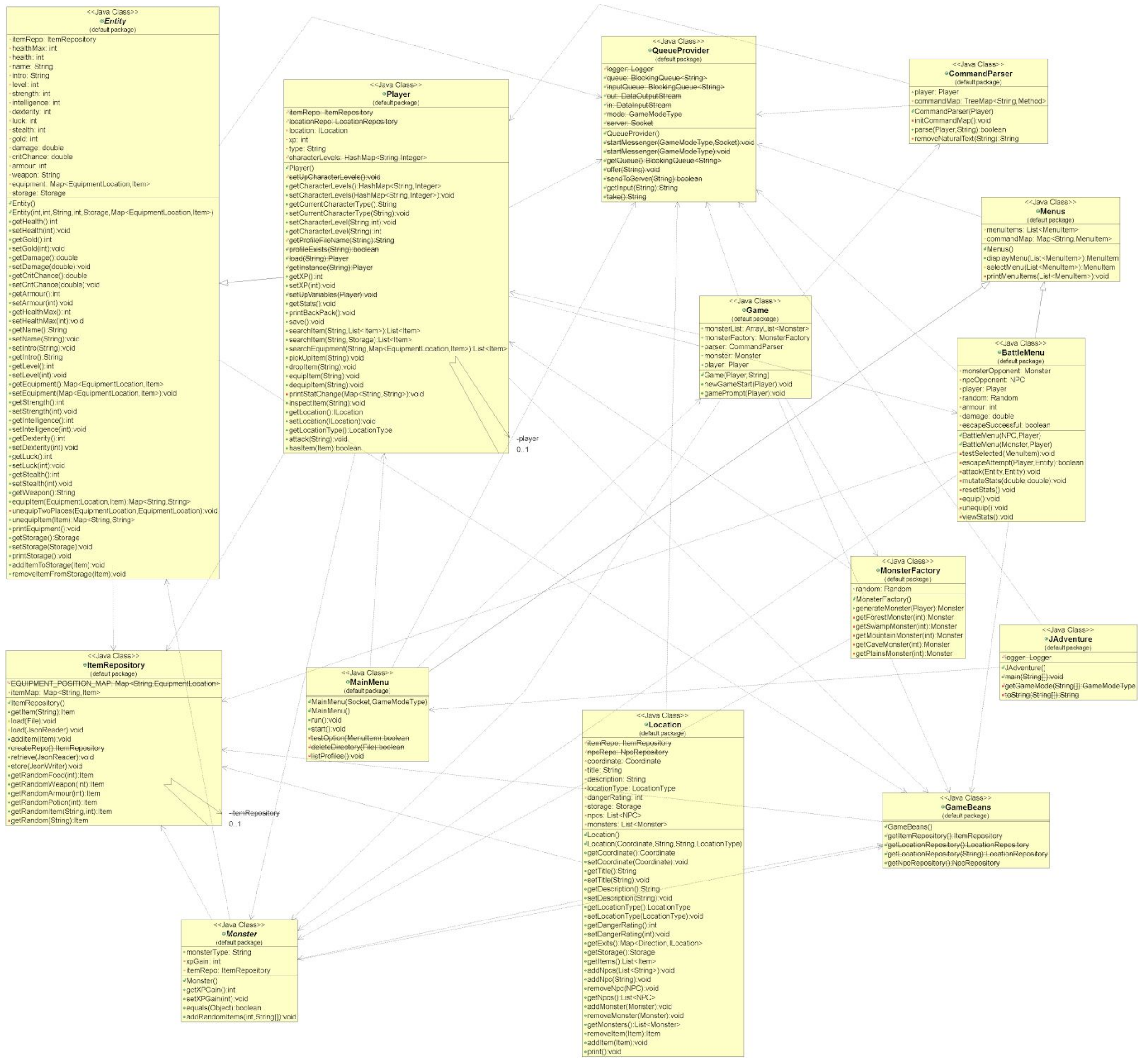
The conversationManager is the class that will handle the conversations that the user will be able to have.The startConversation method is the class that will start the conversation given the player and an NPC. This class gets the possible conversation from a JSON and will then use the appropriate function to display the text to the user.

The Entities Package



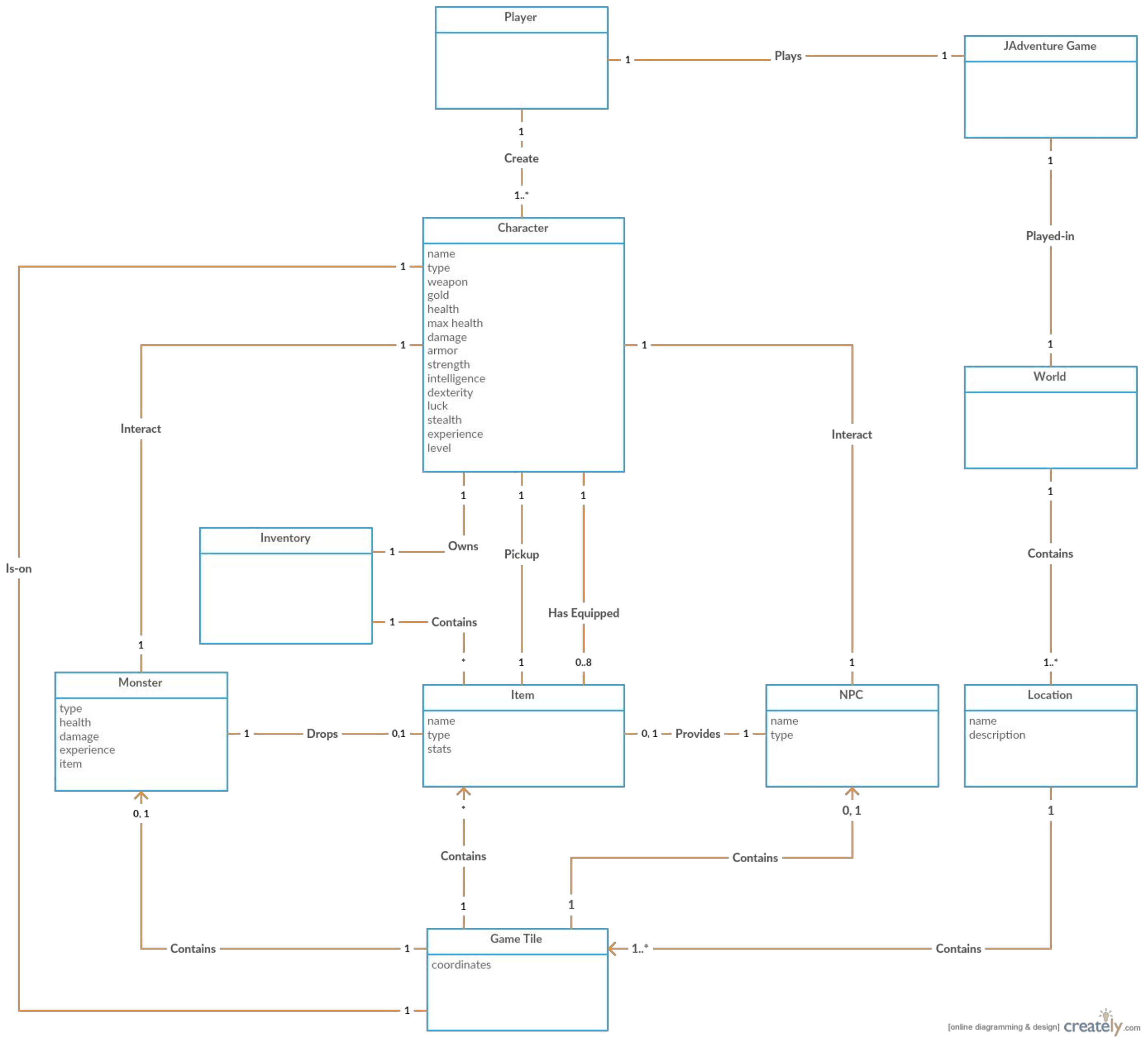
The Entities package is related to the class Monster in addition to the relations shown in this diagram. The class Entity is the superclass of NPC, Player and Monster. Entity contains all the attributes and methods shared by NPC, Player and Monster. The equipment location is just an enum that is used by Player and Entity.

Higher level view to show the high coupling of the project (DO NOT TRY TO UNDERSTAND IT)



This class diagram contains the classes that were deemed the most important to get a basic idea of the overall structure of the program. As can be seen, the high level of dependencies and link between each class makes it practically impossible to understand the relationship between them. Had all the classes actually been shown in the diagram, it would have been an even bigger mess that would be impossible to decipher. This UML is here to show how the architecture of the system was badly implemented and how it could lead to difficulty in refactoring. The high level of coupling of each class makes it easy to see some of the problem that the architecture suffered and why it would require some major overhaul.

Conceptual Diagram



Equivalent classes

Conceptual Class	Actual Class
Character	entities
Player	player
item	item, itemStack
gameTile	Location,LocationManager, Coordinate
Inventory	Storage, Backpack
Monster	monsters,
NPC	NPC
JadventureGame	JAdventure, QueueProvider
World	Game
Location	LocationType

UML Class description

The most important classes are Entity, Menus and Game. All the monsters, the player and the NPCs inherits from the superclass Entity. Menus have a few subclasses which altogether provide a way for the game to display information in the form of menus. Game is the center of JAdventure or in other words the game engine.

- ➔ Game.java
    - ◆ Responsibilities
        - Start new game
        - Continue existing game
        - Runs main loop that keeps taking player input and parses it using CommandParser
    - ◆ Dependencies
        - Player
        - LocationRepository
        - CommandParser
        - MonsterFactory, Monster

- ➔ Menus.java
    - ◆ Responsibilities
        - ● Store menu items
        - ● Store command map (string to MenuItem). Strings are either auto generated consecutive integers or commands from MenuItem. Initialized when the menu is first displayed
        - ● Display menu items along with their commands
        - ● Handle selection of menu items
    - ◆ Dependencies
        - ● QueueProvider
- ➔ Entity.java
    - ◆ Responsibilities
        - ● Abstract base class for all entities
        - ● Store entity properties. These include
            - ○ Name
            - ○ Current health and max health
            - ○ Level
            - ○ Attributes, i.e. strength, dexterity, etc
            - ○ Damage, crit chance, armour
            - ○ Weapon, Equipment and Storage
            - ○ Equip/unequip items
            - ○ Print equipement list
            - ○ Add and remove items from storage
    - ◆ Dependencies
        - ● GameBeans
        - ● ItemRepository, ItemStack, Item
        - ● Storage
        - ● QueueProvider

Reverse engineering tool

To produce our UML based on the code, we choose an Eclipse plugin named ObjectAid UML Explorer. After installing it on our system, it was very easy to start using it. We just had to create a new Class Diagram file and then drag our java classes into it. It automatically displays a UML of all classes and the different associations between them. The only problem with the tool was the fact that it did not properly display the relationship between the different packages. It only displayed the relationships between the classes within each package. One feature that will be useful for our refactoring later on is that whenever we update the code in Eclipse, the corresponding UML diagram updates real time.

Player.java and MonsterFactory.java relationship example

MonsterFactory.java:

```
public class MonsterFactory {
    Random random = new Random();


    public Monster generateMonster(Player player) {
        int randomInt = random.nextInt(5) + 1;
        if (randomInt <= player.getLocation().getDangerRating()) {
            switch (player.getLocationType()) {
                case FOREST:
                    return getForestMonster(player.getLevel());
                case SWAMP:
                    return getSwampMonster(player.getLevel());
                //...
                ...//
                default: // any non-hostile location
                    return null;
            }
        } else {
            return null;
        }
    }


              //other methods using Player
}
```

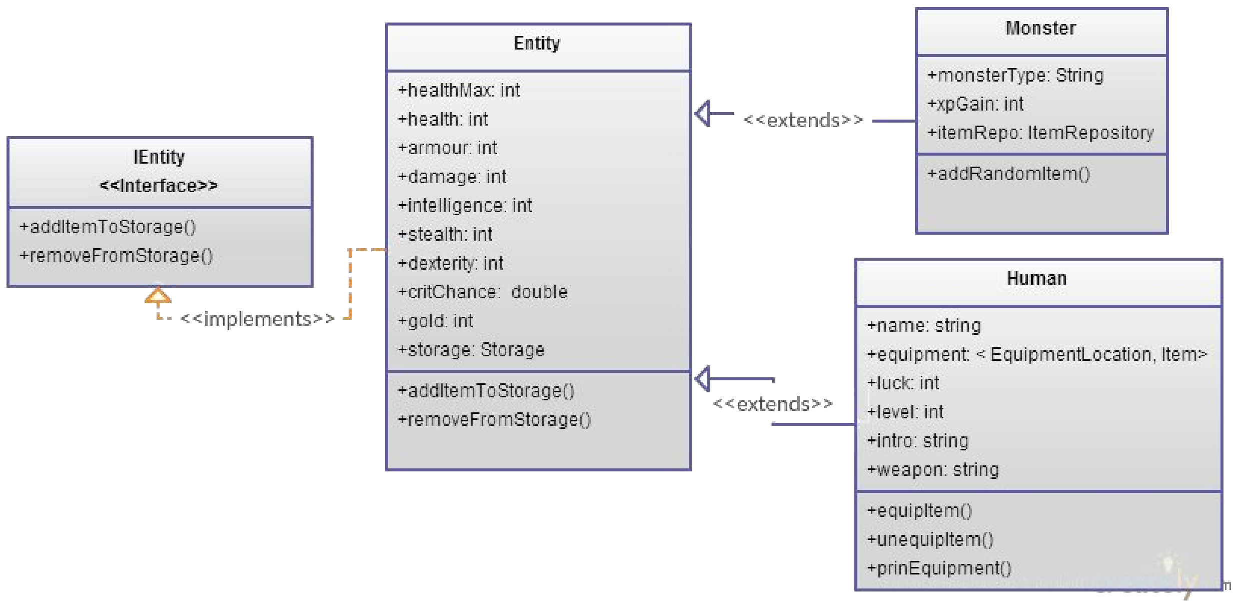
Player.java:

```
public class Player extends Entity {

//...

...//

private ILocation location;

public LocationType getLocationType() {

        return getLocation().getLocationType();

   }

//...

...//

}
```

Code Smells and System Level Refactorings

1) All dependencies are on implementations instead of interfaces and some classes inherits unnecessary attributes and method.

We would create a new Interface IEntity an additional abstract Human class. The methods equipItem, unequipItem and printEquipment will be moved to the Human class because Monster do not need them, therefore increasing cohesion in the Entity class. Some Entity's variable does not belong to both Human and Monster, therefore some attributes such as luck, weapon and equipment others will need to be moved to their corresponding class (Human or Monster). Human would extend Entity. NPC and Player would extend Human instead of extending Entity. Monster would stay the way it is currently implemented.

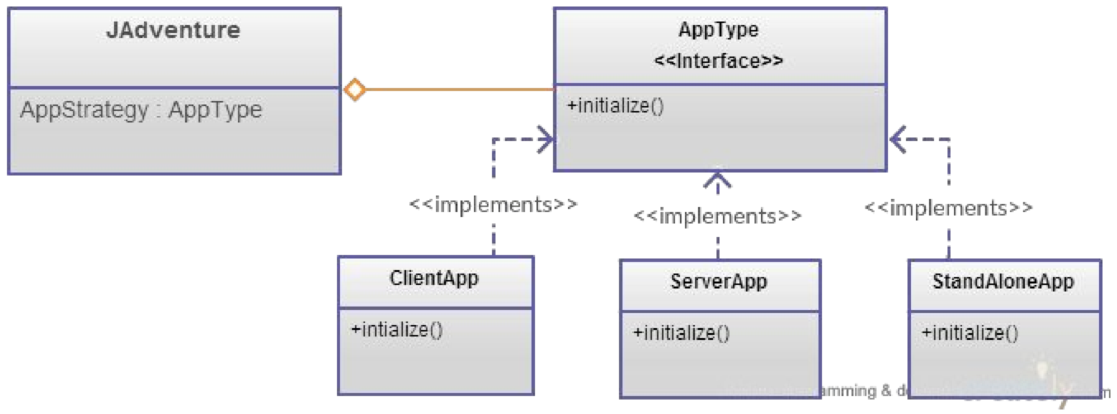


2) Long method

In the JAdventure code, there is multiple instances where the method definition is way too long so has to become very confusing and difficult to understand. This problem is particularly frequent in the Player.java class where methods such as save and load are pretty bulky. To get rid of this problem, the best and easiest solution is to simply extract other simpler methods from this long one to make it easier to read. In the case of the save method, we could first start off by adding a method called addParametersToJSON that would be passed the JSON object as a parameter and then add all the parameters to it from this new method. A separate method can then be created to add the items to the hash map. This would then in turn be added to the json object. The next method that can be extracted from this existing method is a addEquipmentToJSON method that will add the equipment location to a map, compare it with them items, and then add it to the json file. The last bit of the method that could be implemented in this new method is the creation of the save file which could all take place in its own method.

3) Conditionals are used all over the place instead of proper polymorphism. Spaghetti code all around.

The JAdventure class which is used to start the application has two large conditionals in its main method which are overly complicated and add too many responsibilities. The conditional statements are initializing the application based on the game mode read from the command line. Game modes client, server and standalone are possible where the client and server game modes require networking and error handling code. By using polymorphism and the strategy pattern we can extract the initialization logic for each type. Here we can create an interface “AppType” with the abstract method initialize. Each game mode will be a class (ClientApp, ServerApp, StandaloneApp) which implements the interface. The main method can then use polymorphism by declaring a compile time type “AppType” and assigning a concrete class at runtime based on the gamemode command line argument. The “AppType” object can then call initialize method which contains specific initialization logic dependent on the game mode. By removing the initialization logic into separate classes we can reduce the complex conditional logic found in the main method and increase its cohesion.



4) Responsibilities are not well separated, everything is mixed up.

The class BattleMenu is a good example of having too many responsibilities resulting in very low cohesion. All of its functionality is found in two overloaded constructors which accept either a monster or NPC argument and perform similar operations. Within the constructor its first responsibility is to generate the battle menu options. It then performs the actual battle through a loop until a player or monster reaches zero health. After this depending on who won the battle it handles the logic related to player or monster death. If the player dies death exceptions are thrown and restart prompts are displayed. If the player defeats the monster the implementation for gaining experience and leveling up is included. After this the loot from the monster is iterated through and prompts are made to offer them to a player to pick up. Finally gold is picked and messages are displayed based on experience and level gains. To have all of these responsibilities implemented in a single method is very poor design and especially so since this is in a constructor for a battle menu. These constructors are a prime candidates for the extract method refactoring technique where all of the separate functionality listed here can be made to exist in their own respective methods. Ideally the constructor will just initialize the menu options for battling.

5) Misused factory pattern in MonsterFactory, also has low cohesion (should use strategy pattern)

The factory pattern is a static class that contains one static instance of itself - making it a singleton - a private constructor, private attributes corresponding to the entities that are to be created, and a static method called getInstance which returns the instance of the factory class. If none exists, then one is created. Once the instance is created, it can be used to access public methods within the factory class that create and return instances of classes that require complex instantiations. An example in JAdventure is the MonsterFactory class that has been made to check the type of the player's current location, and return a new instance of a random monster depending on the type of the location. Some problems with it however are that the class is not made as a singleton, and that it has low cohesion. It has low cohesion because it also generates the monster through the method generateMonster(Player player), but this shouldn't be the responsibility of the factory. This should instead be the responsibility of a new class called MonsterGenerator. The player class should use this MonsterGenerator class when the player lands on a location. The player should then ask the MonsterGenerator class to decide which specific type of monster it wants to generate, and then call the MonsterFactory class to create and return an instance of that monster type. This would be considered a correct implementation of the Factory pattern, and it would simultaneously use the Strategy pattern.

6) Using exceptions to control logical flow

Exceptions are used for control flow in several places. Exceptions are supposed to be used to handle specific error that arise in unique situation. They are supposed to be rare because they are supposed to handle something that is not supposed to happen regularly. Theses exceptions also adds even more complexity to your code because it will make it harder to maintain it properly. In our application, it become quite a problem because some exceptions aren't catch locally but so far up the chain that we had to track it through multiple java file.

In class BattleMenu, the method Battlemenu will throw a DeathException when the player dies. It will get caught during a gamePrompt to stop the game. Having an exception handling this type of action that should be quite frequent is bad design that could be improved immensely by refactoring. This should be replaced by a Observer pattern where you would track the health attribute of the player. Whenever the player object is changed it will trigger a function that will verify if the health is 0. If it was, then it would launch all the available option for the user : restart game or quit.

7) Rename class methods to be more clear and understandable

- Class: Player
  Method: attack()
  ResponsibilitySearches through npc and monster lists to find the actual opponent. Attacking performed by creation of BattleMenu object.
  Solution: Extract searching to new method "getOpponent()"

- Class QueueProvider
  Method: offer()
  Responsibility: Outputs messages to command line or server depending on game mode.
  Rename: Rename method to outputToServerOrConsole()

- Class: BattleMenu
  Method: testSelected()
  Responsibilities: Selects battle action based on menu option chosen
  Rename: selectBattleAction()

Specific Refactorings that you will implement in Milestone 4.

The three refactorings we plan to implement are the following:

1) Refactoring BattleMenu.java to use the observer pattern instead of DeathException. Remove duplicated code.
2) Refactor Entity.java, add Human.java and IEntity.java to make use of interface implementations and to give access to specific attributes and methods.
3) Refactoring JAdventure.java to use the strategy pattern instead of conditional statements. This involves creating an interface and classes which implement their initialization logic for each game types. Interface AppType and implementing classes ClientApp, ServerApp, StandaloneApp will be added.
4) Refactoring Player.java to correctly use the singleton patern. Refactoring the CommandParser.java to use the singleton patern because everyone is supposed to use the same commands

. *Code for refactoring #1*

```
public class BattleMenu extends Menus {


public BattleMenu(NPC npcOpponent, Player player) throws DeathException {

//...

throw new DeathException("restart");

...//...

throw new DeathException("close");

...//

}

public BattleMenu(Monster monster, Player player) throws DeathException {

//...
throw new DeathException("restart");

...//...

throw new DeathException("close");

...//

}
```

Code for refactoring #2

```
   public abstract class Entity {


   protected ItemRepository itemRepo = GameBeans.getItemRepository();


// All entities can attack, have health, have names

   private int healthMax;

   private int health;

   private String name;

   private String intro;

   private int level;

   // Statistics

   private int strength;

   private int intelligence;

   private int dexterity;

   private int luck;

   private int stealth;

   private int gold;

   private double damage = 30;

   private double critChance = 0.0;

   private int armour;

   private String weapon = "hands";

   private Map<EquipmentLocation, Item> equipment;

   protected Storage storage;
   public Map<String, String> equipItem(EquipmentLocation place, Item item) {...}

   public Map<String, String> unequipItem(EquipmentLocation place, Item item) {...}

private void unequipTwoPlaces(EquipmentLocation leftLocation, EquipmentLocation rightLocation) {...}

public void printEquipment() {...}

public void addItemToStorage(Item item) {...}

public void removeItemFromStorage(Item item) {...}
```

Code for refactoring #3

```
public class JAdventure

public static void main(String[] args) {

String serverName = "localhost";

GameModeType mode = getGameMode(args);

int port = 4044;

     if (mode == GameModeType.SERVER) {...}

     else if (mode == GameModeType.CLIENT) {...}

     if (GameModeType.CLIENT == mode) {...}

     else if (GameModeType.SERVER == mode) {...}

     else {...}
```