Broderick Schmidt and Will Van Vurren
9/20/23
CSCI 4448
Project 2.1

1)

Design by contract is the OOAD idea that we design our schema based on public methods and class/subclass designs we have put in place. These ‘contracts’ are put in place to ensure that data passed between different objects adheres to certain rules that are necessary for the functionality of the program.

An explicit contract is one where this compliance is forced. The programming language Eiffel is built around this principle, which only consists of class and subclasses dictated by design by contract. Implicit is when the conformation to necessary principles is assumed by the data taken into the program.

```
    //implicit contract implementation
    public static void withdraw(amount) {
        if (amount <= this.balance) {
            this.balance =- amount;
        } //assume the amount is non-negative

            return error;
    }

    //explicit contract implementation
    public static void withdraw(amount) {
        if (amount <= balance) {
            balance -= amount;
        }
        else { //if the condition is not met, explicitly throw an exception
            throw new InsufficientFundsException("Insufficient funds to withdraw " + amount);
        }
        //the postcondition is the balance is reduced to the balance - withdrawl amount
    }
}

public class Dog {
```

2)

Modern Java interfaces can have default methods implementations that can be overridden if necessary by implementing classes. This differs from other languages in that Java interfaces can contain methods that are implemented to a default state so they can be used by other classes without being defined each time.

```java
public class Dog {
    void makeNoise();

    void eatFood();
    //this default method can be called for all Dog class objects
    default void jump() {
        System.out.println(x:"Jump!");
    }
}

//Pitbull class will define specific behavior from the abstract Dog class
class Pitbull implements Dog {
    //Pitbull implements the first two methods but can still use the jump() method if it wants to
    public void makeNoise() {
        System.out.println(x:"WOOF!");
    }

    public void  eatFood() {
        System.out.println(x:"Eating steak!");
    }

}
```

Java interfaces can also have static methods, which aids in efficient memory management by marking methods as static so they can be accessed without creating a specific instance of the class.

```java
interface Math{
    static add(int a, int b){
        return a + b;
    }
}

//add() can be used without instantiation because it is marked as static

public class Main{
    public static void main(String[] args) {
        int x;
        int y;
        int sum = Math.add(x, y);
    }
}
```

Another way that Java differs from standard OO interfaces is that Java interfaces can use lambda expressions with abstract methods. This makes it easy to write expressive functions.

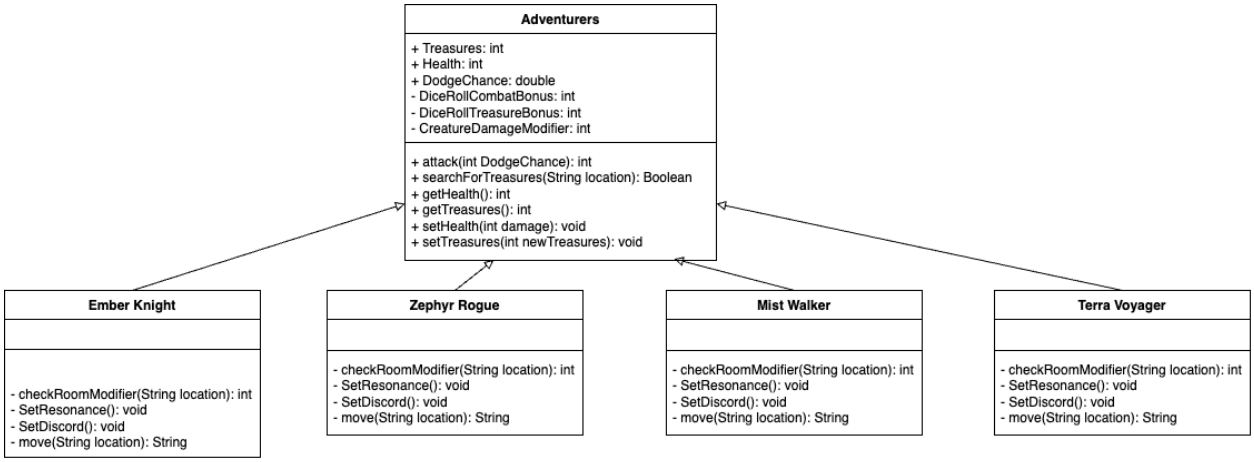
```java
interface Math{
    int operate(int a, int b);
    }

//operate is an abstract method that can be overridden in the main class with lambda functions

public class Main{
    public static void main(String[] args) {
        Math addition = (a, b) -> a + b;
        Math subtraction = (a, b) -> a - b;
        Math multiplication = (a, b) -> a * b;
        Math division = (a, b) -> a / b;
    }
}
```

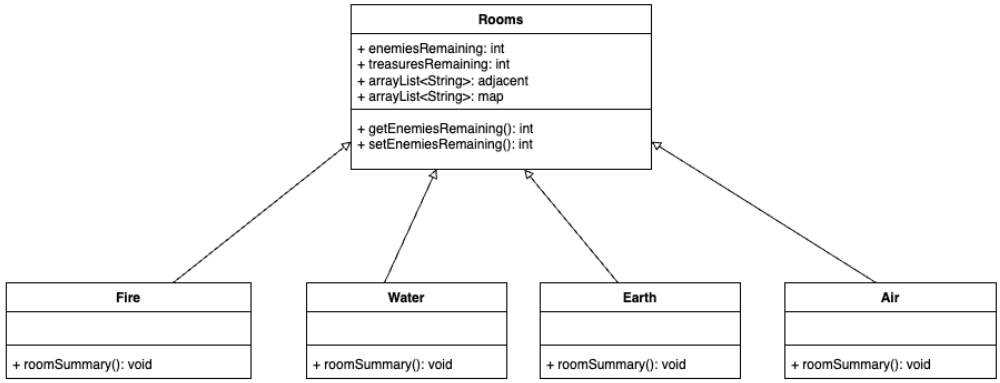
3)

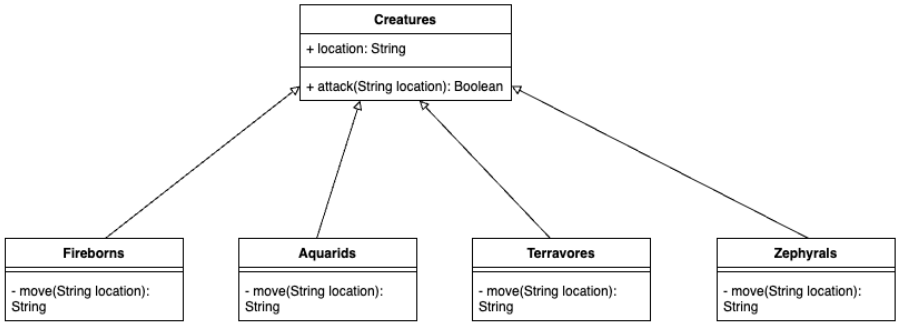
Abstraction focuses more on what information needs to be relayed to the user based on its relevance, whereas encapsulation focuses more on hiding information based on its need for protection.

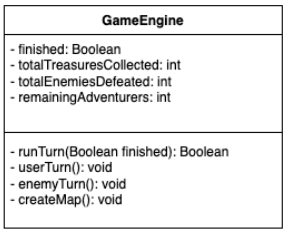
In abstraction, this is seen in its use of abstract classes and interfaces at the design level. Real world concepts are abstracted into classes that only store necessary data to define the abstraction. In encapsulation, this is usually shown by using getter and setter methods to access the data that needs to be accessed.

4) on next page


Adventurers
+ Treasures: int
+ Health: int
+ DodgeChance: double
- DiceRollCombatBonus: int
- DiceRollTreasureBonus: int
- CreatureDamageModifier: int
+ attack(int DodgeChance): int
+ searchForTreasures(String location): Boolean
+ getHealth(): int
+ getTreasures(): int
+ setHealth(int damage): void
+ setTreasures(int newTreasures): void
Ember Knight
- checkRoomModifier(String location): int
- SetResonance(): void
- SetDiscord(): void
- move(String location): String
Zephyr Rogue
- checkRoomModifier(String location): int
- SetResonance(): void
- SetDiscord(): void
- move(String location): String
Mist Walker
- checkRoomModifier(String location): int
- SetResonance(): void
- SetDiscord(): void
- move(String location): String
Terra Voyager
- checkRoomModifier(String location): int
- SetResonance(): void
- SetDiscord(): void
- move(String location): String


Rooms
+ enemiesRemaining: int
+ treasuresRemaining: int
+ arrayList<String>: adjacent
+ arrayList<String>: map
+ getEnemiesRemaining(): int
+ setEnemiesRemaining(): int
Fire
+ roomSummary(): void
Water
+ roomSummary(): void
Earth
+ roomSummary(): void
Air
+ roomSummary(): void


Creatures
+ location: String
+ attack(String location): Boolean
Fireborns
- move(String location): String
Aquarids
- move(String location): String
Terravores
- move(String location): String
Zephyrals
- move(String location): String


GameEngine
- finished: Boolean
- totalTreasuresCollected: int
- totalEnemiesDefeated: int
- remainingAdventurers: int
- runTurn(Boolean finished): Boolean
- userTurn(): void
- enemyTurn(): void
- createMap(): void