CompSci 5448 Object Oriented Analysis and Design: Project 2.1
by David Chaparro and Julia Kent

1. (5 points) What does “design by contract” mean in an OO program? What is the difference between implicit and explicit contracts? Provide a text, pseudo code, or code example of each contract type.

In Object Oriented Programming, "design by contract" is a software design approach focused on specifying "contracts" to define interactions between classes/objects and their clients. This is often seen by defining how a class is to be used, ensuring that objects have a valid state, and making the code more robust and easier to maintain. Design by contract can be achieved by organizing communication into provider/client benefit/obligations (responsibilities) and by establishing pre and post conditions. If preconditions of the class are not obeyed, the service provider will deny service. And if postconditions are violated, this is an indication that there is a service provider issue (commonly an implementation error). Establishing which part of the code is not working as expected makes debugging easier.

Design-by-contract is important when leveraging polymorphism; a class's subclasses should follow the contract established by the parent class's public methods. This is the case unless the design is dangerously method-overloaded and a subclass changes a method's behavior in a way that does not follow the pre-established contract. When the contract is not followed, the code documentation is more likely to have misleading information. Typically, abstract classes act as a blueprint for other subclasses and are a clear example of design-by-contract. The abstract class informs you that all of its subclasses should have certain characteristics.

There is implicit design-by-contract, where there is no forced compliance but the code design still respects this schema, and there is explicit design-by-contract, where the code must follow the contract. Explicit design is implemented at the programming level, and must be enforced. Abstract classes and interfaces let you make design-by-contract more explicit.

One component of design by contract is validating method argument types. In Python some functions specify the input arguments but not their types. Below is an example of implicit design-by-contract in Python:

```
class Example (
      def add_one(s):
            """
            Adds 1 to the end of a string

            Parameters:
                  s (str): A string to be added to

            Returns:
                  s1 (str): The string plus 1
            """
            s1 = s + 1
            return s1
      )
```

In this example, the intended type of the input and return value are specified in the docstring. However, they are not explicitly required from the function argument definition. An integer could be passed in to this function and return that integer plus one without raising an error. Other types of inputs would not work, like passing in a pandas Dataframe, for instance.

Compare this to the explicit contract in this Python example:

```
class Example (
      def add_one(s: str) -> str:
            """
            Adds 1 to the end of a string

            Parameters:
                  s (str): A string to be added to

            Returns:
                  s1 (str): The string plus 1
            """
            s1 = s + 1
            return s1
      )
```

Now the function will raise a TypeError if the intended type is not passed in.
You can also add "assert" statements to make sure pre-or-post conditions are met.

```
class Example (
      def add_one(s: str) -> str:
            """
            Adds 1 to the end of a string

            Parameters:
                  s (str): A string to be added to

            Returns:
                  s1 (str): The string plus 1
            """
            s1 = s + 1

            assert isinstance(s1, str)
            return s1
      )
```

Some languages, such as Ada let you specify pre and post conditions in a with block like so:

```
  type Example;
  function Fx( Arg : Type ) return Type
  with
      Pre => **some pre-condition**
      Post => **some post-condition**
```

(LeadingAgile "Design by Contract",
https://www.leadingagile.com/2018/05/design-by-contract-part-one/)
(Embedded "Contract-based programming: making software more reliable",
https://www.embedded.com/contract-based-programming-making-software-more-reliable/)

2. (5 points) What are three ways modern Java interfaces differ from a standard OO interface design that describes only abstract method signatures to be implemented? Provide a Java code example of each.

Here are 3 ways modern Java interfaces differ from standard OO interface design:

1. Java classes can only have one parent class (single-inheritance). This is as opposed to multiple inheritance in OO interface design. Instead, Java can extend from an interface as another way to achieve abstraction. You can extend a java class from multiple interfaces, but each interface method must be specified in the class's definition. Java doesn't demonstrate extended interface capabilities.

Here is an example of utilizing multiple interfaces in Java:

```
interface SomeInterface {
  public void someMethod(); // interface method
}

interface AnotherInterface {
  public void anotherMethod(); // interface method
}

class SomeClass implements SomeInterface, AnotherInterface {
  public void someMethod() {
    System.out.println("blah");
  }
  public void anotherMethod() {
    System.out.println("blah blah");
  }
}

class Main {
  public static void main(String[] args) {
    SomeClass someObj = new SomeClass();
    someObj.someMethod();
    someObj.anotherMethod();
  }
}
```

Multiple inheritance leads to the "Diamond Problem" which will be discussed in point 3.
(W3Schools "Java Interfaces", *https://www.w3schools.com/java/java_interface.asp)*

2. Pre Java 8, Interfaces could only have method declarations, which is the traditional no-implementation form of an interface in other Object Oriented programs. Post Java 8, there are now default and static methods for interfaces as well as lambda expressions. It is also possible to store some data as constants in java interfaces.

In the below example, the keyword `default` in an interface tells the implementing class that it is not mandatory to provide implementation. Default methods are implicitly public. This is useful for extending interfaces with new methods. This is because, before any new methods added to the interface had to be included in all implementations, and now new methods are automatically available to all implementations (and we don't have to manually edit each implementing class, preserving backward compatibility). You can see that the `defaultMethod()` method is available to an implementing class without needing to be included in that class's code. Similarly, any new functionality added to our interface at a later date would not break the implementing classes.

```
//INTERFACE
public interface SomeInterface{
```

```
    String someMethod();

    default defaultMethod() {
        // default method code //
    }
}


//IMPLEMENTING CLASS
public class SomeClass implements SomeInterface{

    private String str;

    @Override
    public String someMethod() {
        return str;
    }
}

//Calling methods from IMPLEMENTING CLASS
public static void main(String[] args) {
    SomeInterface someclass = new SomeClass("str");
    someclass.defaultMethod();
}
```

The keyword `static` in an interface lets an implementing class directly call the method from the interface. Static methods don't belong to any one object, so they have to be called like `SomeInterface.staticMethod()` no matter where you are calling the method from (just like a static method within a Class), as demonstrated below:

```
//INTERFACE
public interface SomeInterface {
      static staticMethod(String[] args) {
            // static method code //
      }
}

//Calling methods from anywhere
SomeInterface.staticMethod();
```

This is powerful or useful because it allows us to put multiple methods together without creating an object, increasing the code's cohesion. Before, it was somewhat common to see artificial utility classes built to house utility methods that could reasonably be set to `static`.

Java 8 added support for Lambda expressions via functional interfaces, which may change the way some interfaces are implemented. Lambda expressions can be passed around (like objects) and enable the creation of functionality that doesn't belong to any one class. Here is an example of a Lambda expression in Java:

```
interface SomeFunctionalInterface {
    // An abstract function
    void abstractFx(int x);
}

class SomeClass
```

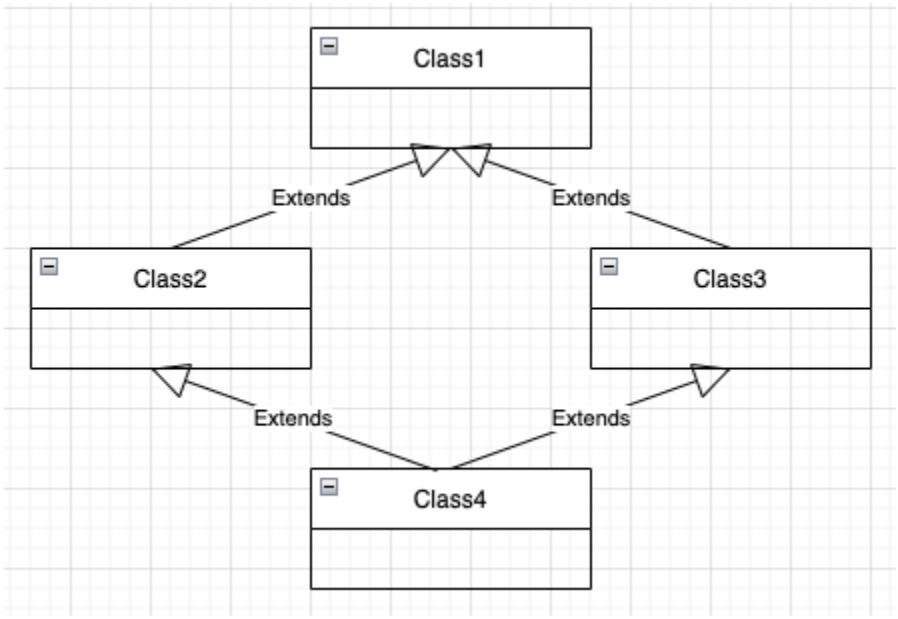
```
{
    public static void main(String args[])
    {
        // lambda expression to implement SomeFunctionalInterface.
        // which by default, implements abstractFx()
        SomeFunctionalInterface fxlambda = (int x)->System.out.println(x*2);

        // Call the lambda expression
        fxlambda.abstractFx(3);
    }
}
```

OUTPUT::
```
6
```

Lamda expressions are a concept form functional programming that have been included into various OO languages to get the benefits of functional programming. In Java, the abstract functions still have to be implemented via a functional interface. In Python, for contrast, Lambda expressions can be defined on the fly. Lambda functions also work differently in C and C++.

(Digital Ocean "Java8 Interface Changes - Static Method, Default Method", *https://www.digitalocean.com/community/tutorials/java-8-interface-changes-static-method-default-method*)
(Geeks for Geeks "Lambda Expressions in Java 8", *https://www.geeksforgeeks.org/lambda-expressions-java-8/*)
(Baeldung "Java Static Default MEthods", https://www.baeldung.com/java-static-default-methods)
Adrian Finlay, "Lambda Expressions in Java, Python, C, and C++", ,*https://medium.com/@afinlay/lambda-expressions-in-java-python-c-c-8cdbca5a5e8b*)

3. Because of the possibility for multiple implementations of interfaces in a Java class, we have to address the Diamond Problem.



The Diamond Problem is a typical Object Oriented challenge, but is usually seen in multiple inheritance of classes (rather than interfaces). It is possible for the two child classes, Class2 and Class3, to establish methods with the same name. Then when Class4 inherits from both of them, which version of the method does it use? The answer can be ambiguous.

Each language has its own way of addressing this problem by establishing the order that methods from subclasses override each other. Here is an example of how Python and Java treat the problem.

In Python whichever method is inherited from first trumps preceding methods. This is shown below where the method that is used in Class4 matches what we'd expect from Class2 (the class inherited from first):

```
class Class1:
      def method(self):
          print("of Class1")

class Class2(Class1):
      def method(self):
          print("of Class2")

class Class3(Class1):
      def method(self):
          print("of Class3")

class Class4(Class2, Class3):
      pass


obj = Class4()
obj.method()
```

OUTPUT::

```
of Class2
```

In Java, pretend the above diagram said "Interface1" etcetera, because Java tried to solve this problem by not allowing multiple inheritance of classes:

```
class Class1() {
      public void method() {
            System.out.println("of Class1");
      }
}

class Class2 extends Class1() {
      @Override
      public void method() {
            System.out.println("of Class2");
      }
}

class Class3 extends Class1() {
      @Override
      public void method() {
            System.out.println("of Class3");
      }
```

```
}

class Class4 extends Class2, Class3 () {
	Class4 class4 = new Class4();
	class4.method();

}
```

OUTPUT::
ERROR

But with the addition of multiple inheritance of interfaces, Java reintroduced the problem. Here is how Java solves it: If the interfaces have default methods (as discussed above), then both methods with identical names can be called within the implemented class.

```
interface Interface2{
	default void method() {
		System.out.println("of Interface2");
	}
}

interface Interface3{
	default void method() {
		System.out.println("of Interface3");
	}
}

class SomeClass implements Interface2, Interface3 () {
	public void method() {
	Interface2.super.method();
	Interface3.super.method();
	}

	public static void main(String argos[]) {
		SomeClass someclass = new SomeClass();
		someclass.method();
	}
}
```

OUTPUT::
```
of Interface 2
of Interface 3
```

And if we don't use default methods within Interfaces2 and 3, Java can grab the method from Interface1:

```
interface Interface1 () {
	default void method() {
		System.out.println("of Interface1");
	}
}

interface Interface2 extends Interface1() {

}
```

```
interface Interface3 extends Interface1() {

}

class SomeClass implements Interface2, Interface3 () {
      public static void main(String argos[]) {
            SomeClass someclass = new SomeClass();
            someclass.method();
      }
}
```

OUTPUT::
```
of Interface 1
```

(GeeksforGeeks "Multiple Inheritance in Python",
https://www.geeksforgeeks.org/multiple-inheritance-in-python/)
(JavatPoint "What is Diamond Problem in Java",
https://www.javatpoint.com/what-is-diamond-problem-in-java)

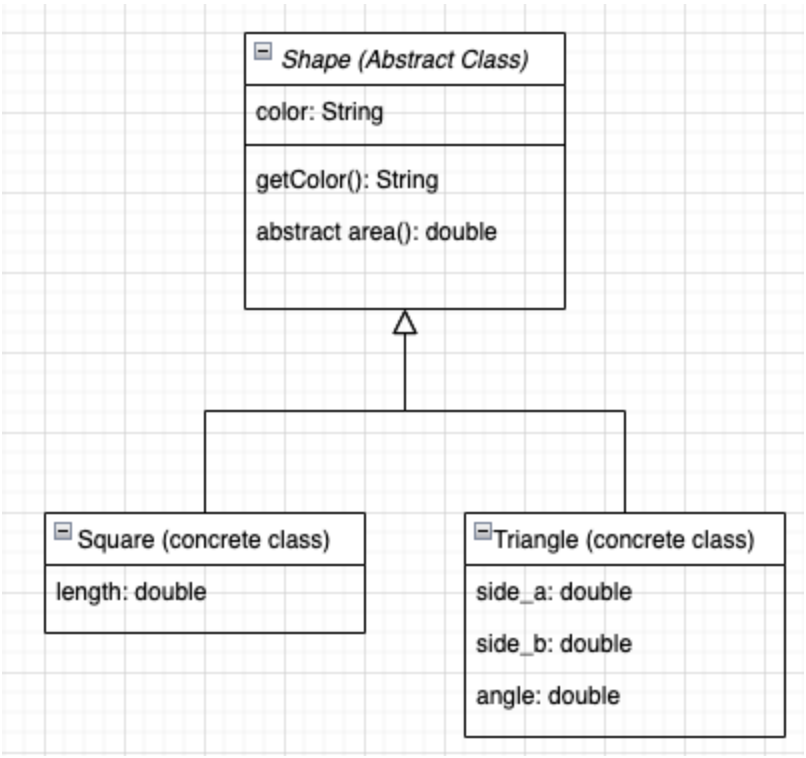
3. (5 points) Describe the differences and relationship between abstraction and encapsulation. Provide a text, pseudo code, or Java code example that illustrates the difference.

Abstraction largely pertains to displaying only the essential details to the end user. It involves hiding unnecessary details to the person using the specified object, so that use is as simple as possible. Abstraction is largely a high level design concept which exposes important to the user and hides unnecessary details while encapsulation deals with grouping specific data together to be hidden.

Encapsulation is a concept that is used as a part of data abstraction and is used to specifically control the access that a user has to methods or data. Encapsulation can occur within data abstraction (like setting methods to private) and is specifically used for data hiding.

Encapsulation is a concept that takes place at the programming and implementation level as opposed to the abstraction which takes place at the design process. Abstraction can be implemented by creating abstract classes and interfaces while encapsulation is largely the access requirements set by the program.

An example of Abstraction vs encapsulation is provided here and shown below:
(GeeksforGeeks "Abstraction in Java",
https://www.geeksforgeeks.org/abstraction-in-java-2/?ref=lbp)



Above is an example of abstraction, which is part of the design process. In this example, an abstract class is created and the functions inside of it are used to interact with the objects but no details of the area() or getColor() methods are mentioned. The user only interacts with those functions but has no knowledge of the details. Each shape class overrides the abstract area function with a concrete area function specific to that shape.

```
// Java program to demonstrate encapsulation
class Encapsulate {
    // private variables declared
    // these can only be accessed by
    // public methods of class
    private String geekName;
    private int geekRoll;
    private int geekAge;

    // get method for age to access
    // private variable geekAge
    public int getAge() { return geekAge; }

    // get method for name to access
    // private variable geekName
    public String getName() { return geekName; }

    // get method for roll to access
    // private variable geekRoll
    public int getRoll() { return geekRoll; }

    // set method for age to access
    // private variable geekage
    public void setAge(int newAge) { geekAge = newAge; }
```

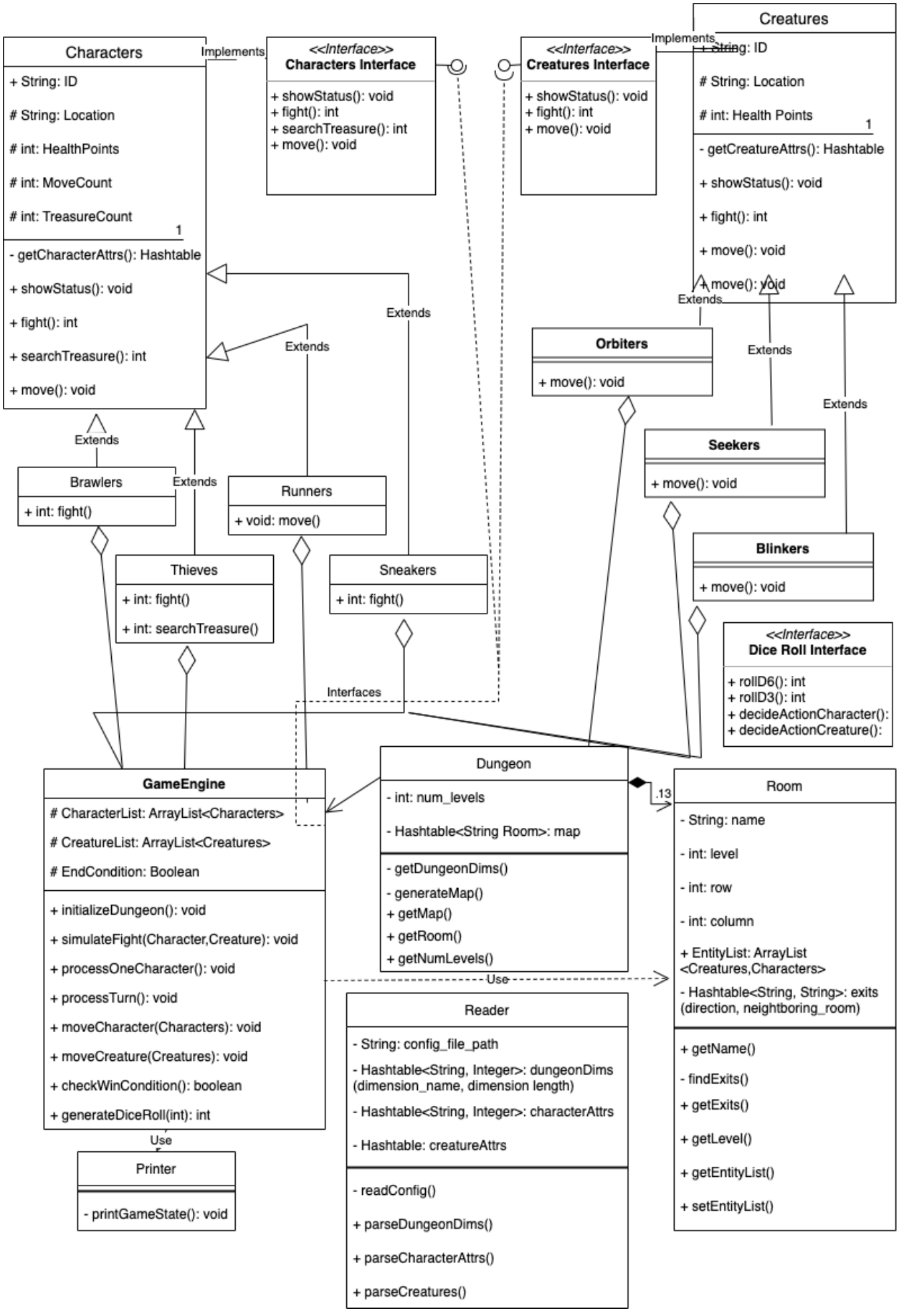
```
    // set method for name to access
    // private variable geekName
    public void setName(String newName)
    {
        geekName = newName;
    }

    // set method for roll to access
    // private variable geekRoll
    public void setRoll(int newRoll) { geekRoll = newRoll; }
}
```

The above example of code (also from the link above) represents encapsulation. Encapsulation occurs at the programming level when access modifiers are applied to methods or variables to restrict access to them. In the above example, the variables geekName, geekAge and geekRoll are all set to private and cannot be accessed from the object directly. One can access the variables using the methods that are public. Thus, the encapsulation only hides the variables geekName, geekAge, and geekRoll. In this example, abstraction only refers to the structure of which methods and which variables are free to use from an outside user.

Encapsulation is especially useful because it specifies ways that users can interact with an object. By setting a variable to private and instead creating a read function to display a variable, you can manage how a variable is accessed by controlling the method accessing it (For example you can track how many times that variable is read).

4. (5 points) Draw a UML class diagram for the RotLA simulation described in Project 2.2. The class diagram should contain any classes, abstract classes, or interfaces you plan to implement to make the system work. Classes should include any key methods or attributes (not including constructors). Delegation or inheritance links should be clear. Multiplicity and accessibility tags are optional. Design of this UML diagram should be a team activity if possible and should help with eventual code development.

Our UML Diagram is attached below. See the next page.


Characters
+ String: ID
# String: Location
# int: HealthPoints
# int: MoveCount
# int: TreasureCount
1
- getCharacterAttrs(): Hashtable
+ showStatus(): void
+ fight(): int
+ searchTreasure(): int
+ move(): void
Implements
<<Interface>>
Characters Interface
+ showStatus(): void
+ fight(): int
+ searchTreasure(): int
+ move(): void
<<Interface>>
Creatures Interface
+ showStatus(): void
+ fight(): int
+ move(): void
Implements
Creatures
+ String: ID
# String: Location
# int: Health Points
1
- getCreatureAttrs(): Hashtable
+ showStatus(): void
+ fight(): int
+ move(): void
+ move(): void
Extends
Extends
Extends
Extends
Extends
Extends
Extends
Brawlers
+ int: fight()
Runners
+ void: move()
Thieves
+ int: fight()
+ int: searchTreasure()
Sneakers
+ int: fight()
Orbiters
+ move(): void
Seekers
+ move(): void
Blinkers
+ move(): void
<<Interface>>
Dice Roll Interface
+ rollD6(): int
+ rollD3(): int
+ decideActionCharacter():
+ decideActionCreature():
Interfaces
GameEngine
CharacterList: ArrayList<Characters>
CreatureList: ArrayList<Creatures>
EndCondition: Boolean
+ initializeDungeon(): void
+ simulateFight(Character,Creature): void
+ processOneCharacter(): void
+ processTurn(): void
+ moveCharacter(Characters): void
+ moveCreature(Creatures): void
+ checkWinCondition(): boolean
+ generateDiceRoll(int): int
Use
Printer
- printGameState(): void
Dungeon
- int: num_levels
- Hashtable<String Room>: map
- getDungeonDims()
- generateMap()
+ getMap()
+ getRoom()
+ getNumLevels()
.13
Use
Room
- String: name
- int: level
- int: row
- int: column
+ EntityList: ArrayList
<Creatures,Characters>
- Hashtable<String, String>: exits
(direction, neightboring_room)
+ getName()
- findExits()
+ getExits()
+ getLevel()
+ getEntityList()
+ setEntityList()
Reader
- String: config_file_path
- Hashtable<String, Integer>: dungeonDims
(dimension_name, dimension length)
- Hashtable<String, Integer>: characterAttrs
- Hashtable: creatureAttrs
- readConfig()
+ parseDungeonDims()
+ parseCharacterAttrs()
+ parseCreatures()