SWEN20003
Object Oriented Software Development
Workshop 7 (Solutions)

Eleanor McMurtry

Semester 2, 2020

Workshop

Questions

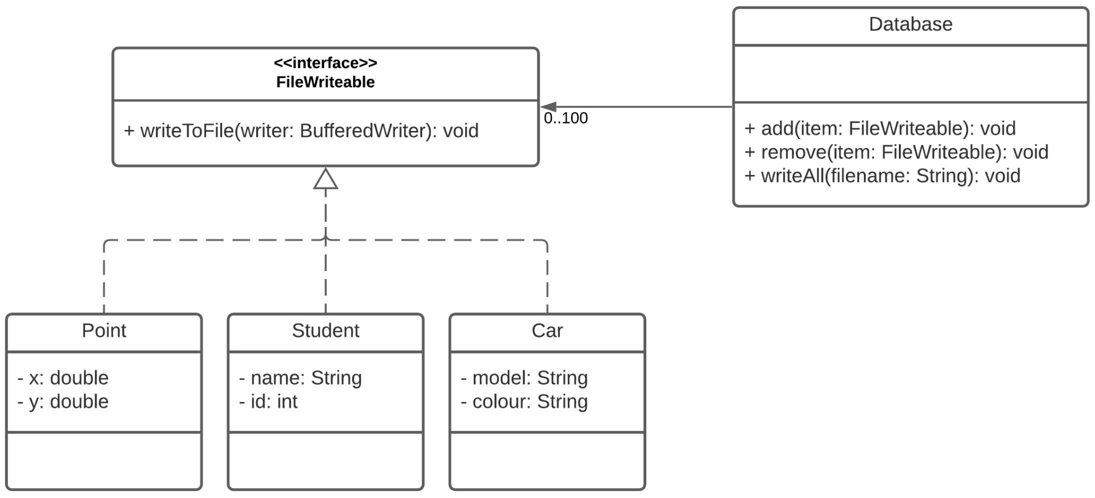
1. Implement Java classes following the diagram on the previous page.

   Solution:

```java
public abstract class Person implements Comparable<Person> {
    private final String name;
    private final int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public int compareTo(Person other) {
        int result = name.compareTo(other.name);
        if (result != 0) {
            return result;
        } else {
            return age - other.age;
        }
    }
}

public class Student extends Person {
    private final int number;
    private final List<Subject> subjects = new ArrayList<>();

    public Student(String name, int age, int number) {
        super(name, age);
        this.number = number;
    }

    public void enrol(Subject subject) {
        subjects.add(subject);
    }
}

public class Subject {
    private final List<Student> students = new ArrayList<>();

    public void enrol(Student student) {
        students.add(student);
        }
    }
```

2. Create a UML diagram to represent the classes and interface from Question 1 last week.

   Solution:

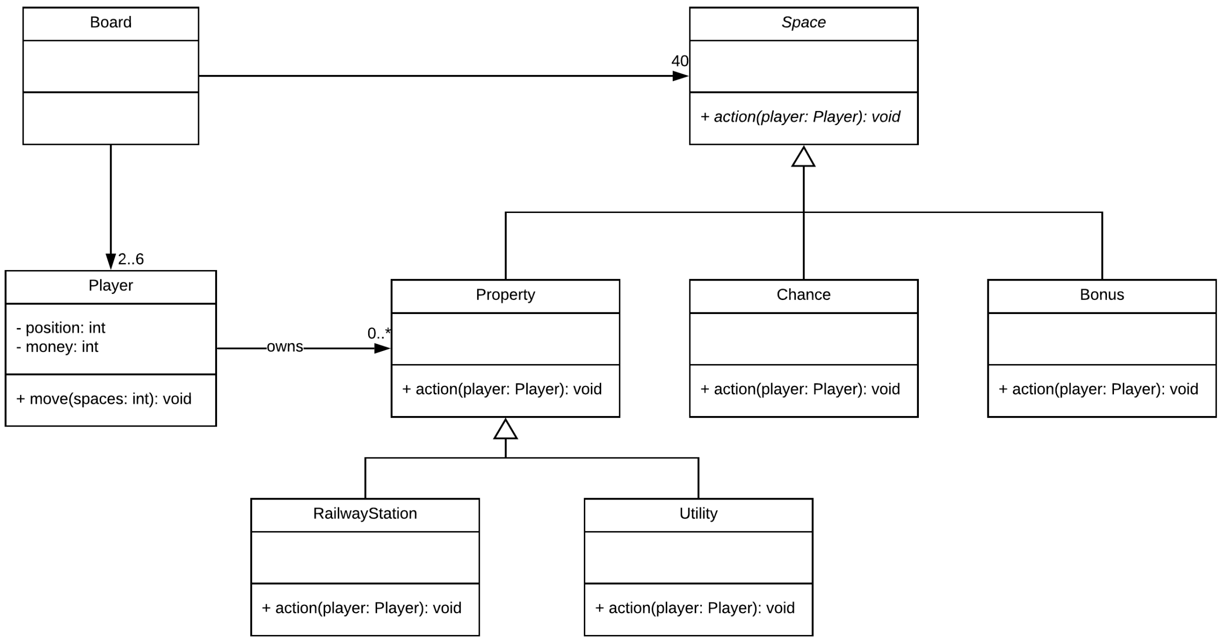


<<interface>>
FileWriteable
+ writeToFile(writer: BufferedWriter): void
0..100
Database
+ add(item: FileWriteable): void
+ remove(item: FileWriteable): void
+ writeAll(filename: String): void
Point
- x: double
- y: double
Student
- name: String
- id: int
Car
- model: String
- colour: String

3. Create a UML diagram representing a design for the following scenario:

   - The game of Monopoly is defined by a *board*, which contains 40 *spaces*, and between 2 to 6 *players*.
   - A space can be either a *property*, *chance*, or *bonus*, and each of the types has a different *action* when a player lands on them.
   - Properties may additionally be *railway stations* or *utilities*, each with a different action when a player lands on them.
   - Players each have a position on the board, an amount of money that they have, a number of properties that they own, and can move along the board.

   Solution:

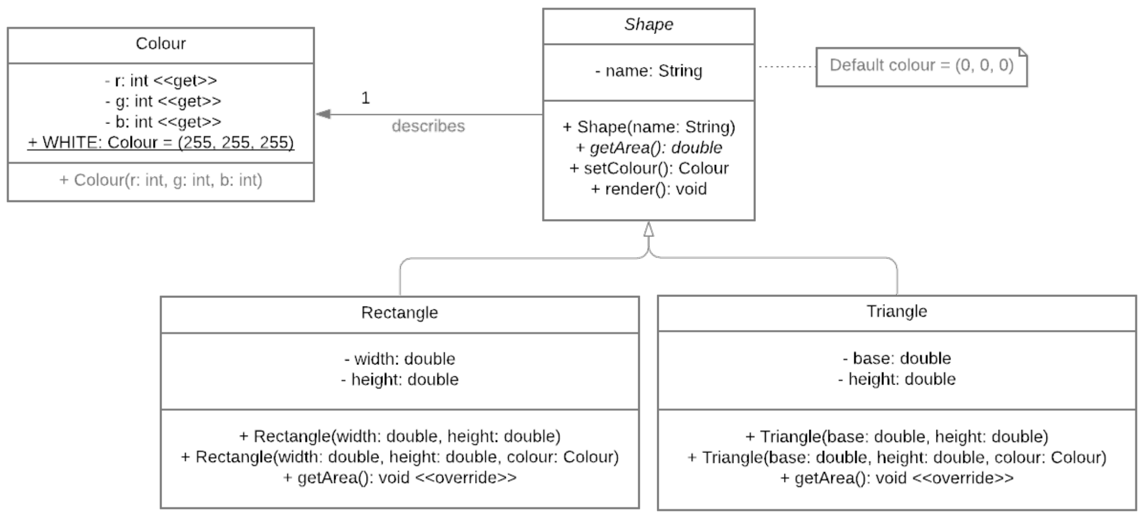


Board
40
Space
+ action(player: Player): void
2..6
Player
- position: int
- money: int
+ move(spaces: int): void
owns
0..*
Property
+ action(player: Player): void
Chance
+ action(player: Player): void
Bonus
+ action(player: Player): void
RailwayStation
+ action(player: Player): void
Utility
+ action(player: Player): void

4. Create a UML diagram representing a design for the following scenario:

   - We are ambitious Java enthusiasts and are already ready to begin creating our own small 'graphics' library. We are designing a system to render simple shapes onto the screen. For now, we are concerned about two types of shapes in particular: **square**s and **triangle**s. A shape has a specific area associated with it, and it can also be rendered to the screen.
   - A shape also has a **colour** associated with it. We will be using the the RGB colour system which specifies a colour through three values: *red*, *green*, *blue*. The red, green, and blue values of a colour must be within the range of 0-255 (inclusive) at all times. If a colour is not specified, a shape's default colour is black (red = 0, green = 0, blue = 0).

   Solution: Note the use of `<<get>>` to mark attributes with getters, and the use of `<<override>>` to


Colour
- r: int <<get>>
- g: int <<get>>
- b: int <<get>>
+ WHITE: Colour = (255, 255, 255)
+ Colour(r: int, g: int, b: int)
1
describes
Shape
- name: String
+ Shape(name: String)
+ getArea(): double
+ setColour(): Colour
+ render(): void
Default colour = (0, 0, 0)
Rectangle
- width: double
- height: double
+ Rectangle(width: double, height: double)
+ Rectangle(width: double, height: double, colour: Colour)
+ getArea(): void <<override>>
Triangle
- base: double
- height: double
+ Triangle(base: double, height: double)
+ Triangle(base: double, height: double, colour: Colour)
+ getArea(): void <<override>>

   mark overridden methods. These **are not required** for the subject, but are useful for communication.