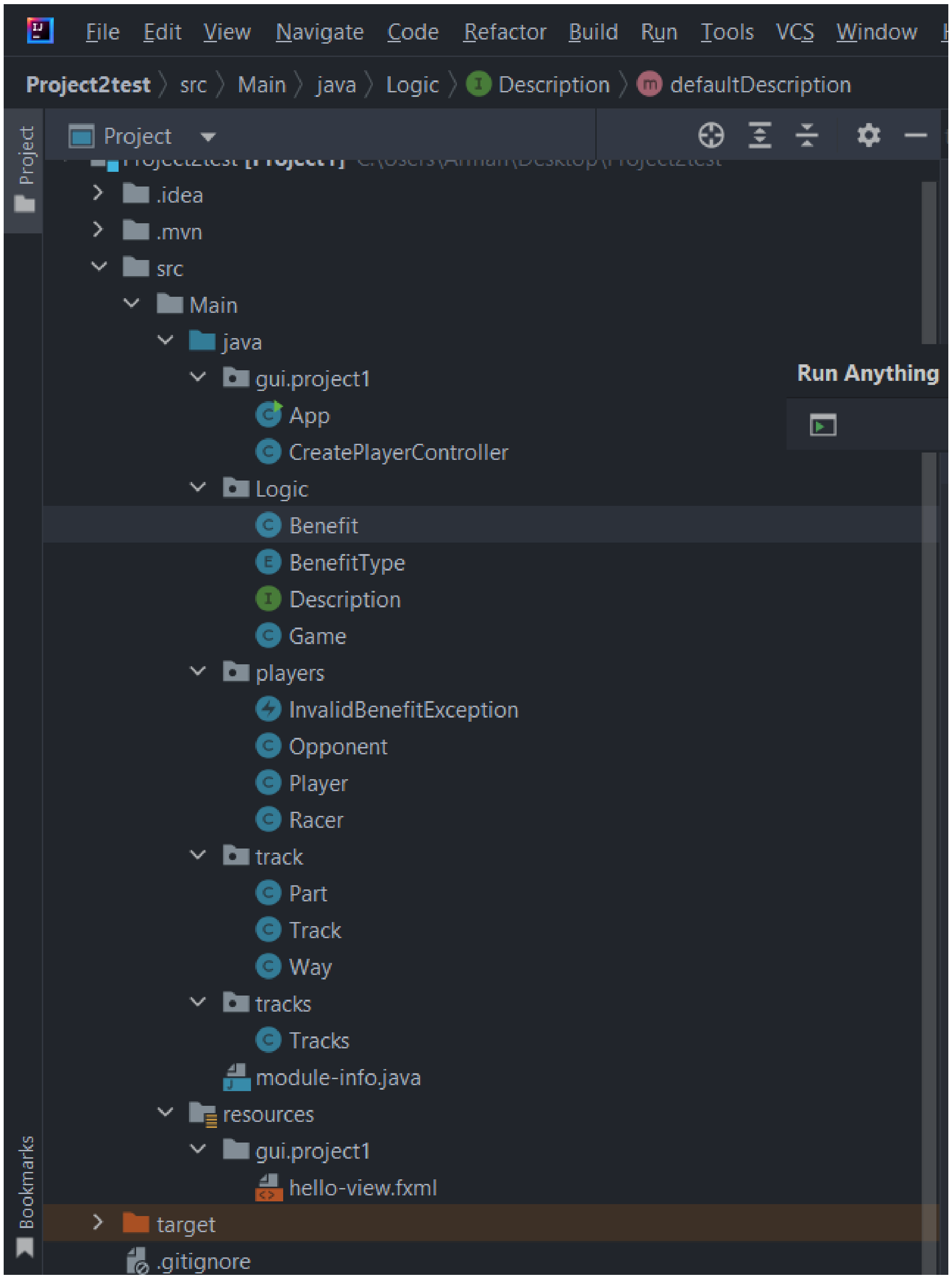
RACING TRACK NAVIGATION

Arman Vardanyan

Program pozostáva z nasledujúcich častí:



```
Project2test [Project1] C:\Users\Arman\Desktop\Project2test
  > .idea
  > .mvn
  v src
    v Main
      v java
        v gui.project1
            App
            CreatePlayerController
        v Logic
            Benefit
            BenefitType
            Description
            Game
        v players
            InvalidBenefitException
            Opponent
            Player
            Racer
        v track
            Part
            Track
            Way
        v tracks
            Tracks
          module-info.java
      v resources
        v gui.project1
            hello-view.fxml
  > target
    .gitignore
```

Hlavne Classi su: -Racer, z ktoreho dedia Player a Opponent
-Track, ktora má Part, Part má Way a Way má Benefity

-Benefity maju typi z enum a implementuju interface

-celý interface je v CreatePlayerController a celá funckinalita gui implementovana v game classe

program ako celok funguje ako v prvom popise hry. Program je v podstate navigátor pretekárskej dráhy, v programe sa vytvorí hráč a súperi a začína sa ťahový pretek, v programe je niekoľko chýb, ktoré treba ešte dokončiť a vylepšiť, ale tie najjednoduchšie akcie fungujú bezchybne. program končí vyhlásením víťazného jazdca

v guthub bola odovzdana pracovna verzia.

Z Ďalšich kriterii su:

- handling exceptional states using own exceptions – one exception is sufficient, but it has to be actually thrown and handled

```java
package players;

public class InvalidBenefitException extends Exception {
    public InvalidBenefitException(String message) {
        super(message);
    }
}
```

```java
public void addBenefit(String name) throws InvalidBenefitException {
    try {
        BenefitType type = BenefitType.valueOf(name.toUpperCase());
        this.benefit = new Benefit(name, type);
    } catch (IllegalArgumentException e) {
        throw new InvalidBenefitException("Invalid benefit: " + name);
    }
}
```

- providing a graphical user interface separated from application logic and with at least part of the event handlers created manually – counts as a fulfillment of two further criteria

```java
package gui.project1;

import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.Label;
import Logic.Game;
import javafx.scene.control.TextArea;
import players.InvalidBenefitException;


public class CreatePlayerController {

    @FXML
    public TextArea consoleTextArea;

    private Game game;

    public CreatePlayerController() {
        game = new Game("Hello", consoleTextArea);
    }

    @FXML
    private Label welcomeText;

    @FXML
    void onHelloWorld(ActionEvent event) {
        game.helloWorld();
    }

    @FXML
    void onCreatePlayer(ActionEvent event) {
        game.createPlayer();
    }

    @FXML
    void onCreateOpponent(ActionEvent event) {
       game.createOpponent();
    }



    public void onPlay(ActionEvent actionEvent) {
        game.play();
    }

    public void onAction(ActionEvent actionEvent) throws InvalidBenefitException
{
        game.next();
    }

    @FXML
    public void onRestart(ActionEvent actionEvent) {
        game.resetGame();
    }

    @FXML
    public void onCreateTrack(ActionEvent actionEvent) {
        // TODO
    }

    @FXML
    public void onStart(ActionEvent actionEvent) {
        game.start();
    }

    public void onChooseOne(ActionEvent actionEvent) {
        game.chooseOne();
    }

    public void onChoosetwo(ActionEvent actionEvent) {
        game.chooseTwo();

    }
```

- using generics in own classes – implementing and using an own generic class (as in the linked list example provided with lecture 5)

```
package Logic;


public class Benefit<T> implements Description {
    private String name;
    private BenefitType type;
    private T cast;

    private String description;

    public Benefit(String name, T cast) {
        this.name = name;
        this.cast = cast;
    }

```

- explicit use of RTTI – for example, to determine the type of on object or to create an object of a certain type (as in determining the number of beings in the ogre and knights game)

```
// Method to create a Benefit object based on the provided type
private Benefit createBenefitInstance(String name, BenefitType type) {
    Benefit benefit = null;

    // Use RTTI to create an object of a specific type
    if (type == BenefitType.TURBO) {
        benefit = new TurboBenefit(name, type);  // Assuming TurboBenefit is a
subclass of Benefit
    } else if (type == BenefitType.OIL) {
        benefit = new OilBenefit(name, type);  // Assuming OilBenefit is a
subclass of Benefit
    }

    return benefit;
}
```

- using nested classes and interfaces – only using them within the application logic counts, not in the GUI, whereby the interfaces have to be own (one possibility is present in the inner class example provided with lecture 4)

```
•
   public class Benefit<T> implements Description {
       private String name;
       private BenefitType type;
       private T cast;


       public Benefit(String name, BenefitType type) {
           this.name = name;
           this.type = type;
       }
```

- using default method implementation in interfaces

```
package Logic;

public interface Description {
    void addDescription(String Desctiption);

    default void defaultDescription() {
        System.out.println("this is an benefit which you can get on the road");
    }
}
```