Universidade do Minho
Escola de Engenharia

Racing Manager

Modelação Conceptual e Implementação da Solução

Trabalho Prático
Desenvolvimento de Sistemas de Software

Grupo 39

Gabriela Santos Ferreira da Cunha - a97393

João António Redondo Martins - a96215

João Pedro Antunes Gonçalves - a95019

Miguel de Sousa Braga - a97698

Nuno Guilherme Cruz Varela - a96455


a97393


a96215


a95019


a97698


a96455

Repositório GitHub: `https://github.com/GVarelaa/ProjetoDSS39`

janeiro, 2023

Conteúdo

Lista de Figuras

1 Introdução

Nesta última fase do trabalho implementamos o jogo de simulação de corridas que tínhamos vindo a projetar e a planear nas duas fase anteriores. Procuramos dar um foco maior aos processos de criação e simulação de campeonatos. Neste relatório apresentamos o processo de desenvolvimento seguido, começando pelo levantamento de alterações no modelo arquitetural, passando depois para a modelação da arquitetura com persistência e consequentes alterações e, por fim, a implementação do jogo.

2 Alterações relativamente às fase anteriores

2.1 Modelação dos algoritmos de simulação

Um dos pontos que não tinha ficado claro na segunda fase do trabalho, que diz respeito à modelação conceptual do sistema, foi a modelação dos algoritmos que realizam a simulação de uma volta, quer no modo normal, quer no modo *premium*. No diagrama de sequência da operação "simulateNextLap", foi agora adicionada uma nova condição para verificar e o carro não estava já avariado.

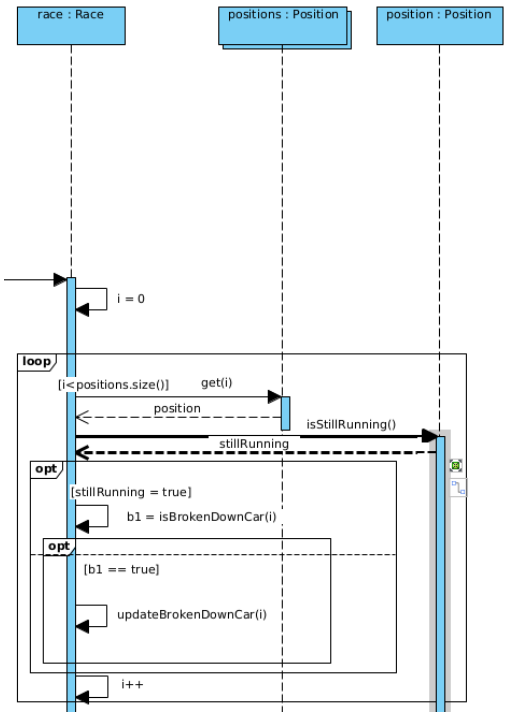

Figura 1: Simulação das avarias.

Quanto à simulação do setor de cada corrida, começamos por verificar se o carro saiu da pista. Caso este tenha saído da pista, atualizamos a sua posição na corrida. Em seguida, verificamos se este ultrapassa o carro que está à sua frente. Em caso de ultrapassagem, atualizamos novamente a sua posição na corrida.

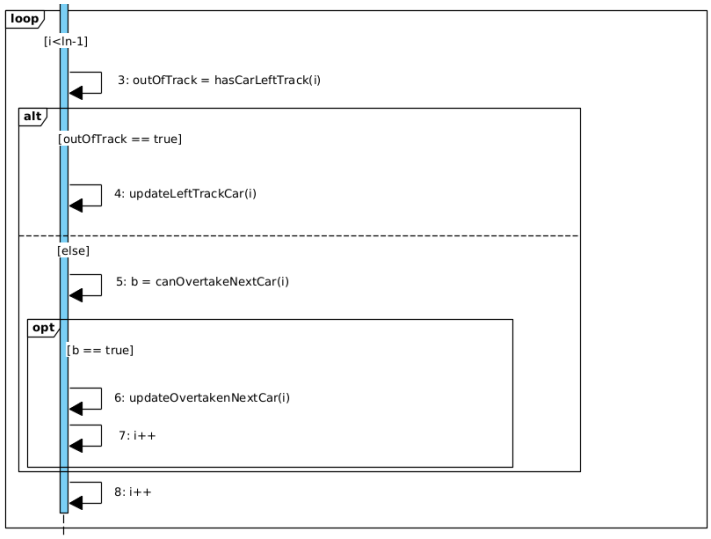

Figura 2: Simulação dos acidentes e das ultrapassagens.

A operação "updateLeftTrackCar" verifica se o carro na posição imediatamente anterior ainda está em corrida. Caso esteja, haverá uma troca de posições entre os dois carros. Consideramos, por isso, que, caso um carro saia da pista ele só descerá, no máximo, uma posição.

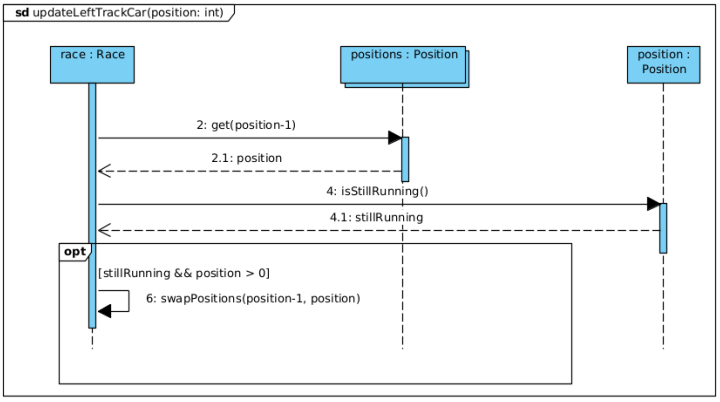

Figura 3: Diagrama de sequência da operação "updateLeftTrackCar".

Quanto à operação "updateOvertakenNextCar", apenas precisamos de garantir que o carro não é o primeiro para fazer a troca de posições entre este e o jogador que está à sua frente.

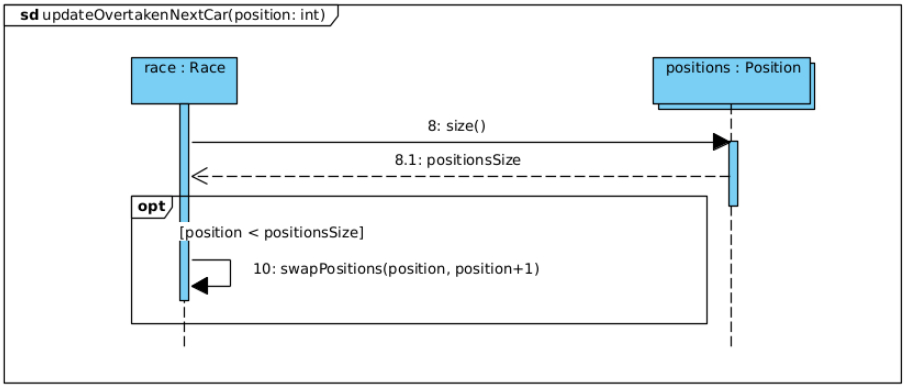

Figura 4: Diagrama de sequência da operação "updateOvertakenNextCar".

Primeiramente, a operação "updateOvertakenNextCar" deverá colocar o carro respondido como inválido para continuar a corrida. Depois, deverá colocá-lo no fim da lista, havendo um avanço de uma posição para cada um dos jogadores que vinha atrás desse jogador.

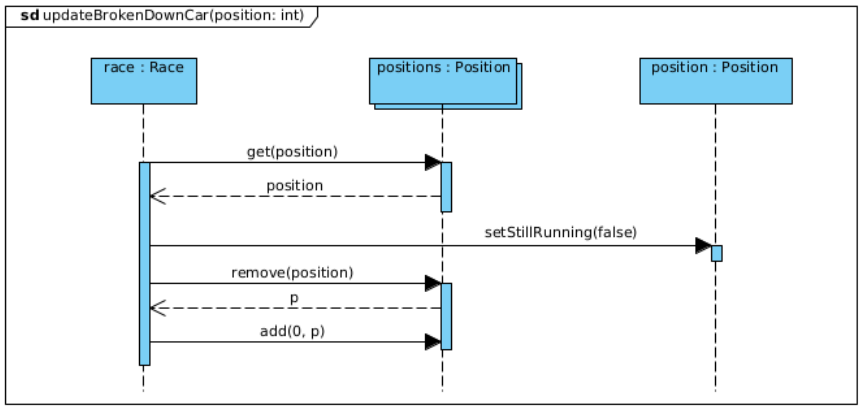

Figura 5: Diagrama de sequência da operação "updateBrokenDownCar".

A lógica da operação "hasCarLeftTrackSeq" assenta no cálculo da probalidade de o carro ter um despiste. Essa probabilidade depende de dois fatores: o fator de risco que é calculado em função do grau de dificuldade do setor, do tipo de

setor a simular e do critério SVA do piloto e o fator metereológico, que depende do estado do tempo e do tipo de pneus que o utilizador escolheu. Por fim, deve ser simulado o resultado, recorrendo a uma função que simula esse evento aleatória, dada uma probabilidade.

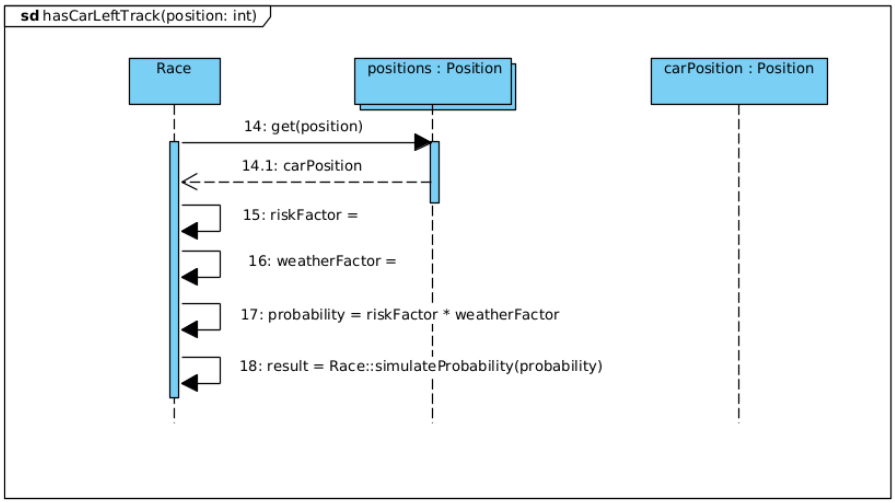

Figura 6: Diagrama de sequência da operação "hasCarLeftTrack".

2.2 Novo atributo adicionado: estado dos pneus

Após uma nova leitura do enunciado, com o intuito de entender melhor os requisitos do processo de simulação de forma a modelar melhor os algoritmos, apercebemo-nos que tínhamos deixado de parte um atributo importante do carro: o estado dos pneus. Assim, o primeiro passo foi atualizar a documentação existente, começando pelo modelo de domínio:

Figura 7: Atualização no modelo de domínio.

Em termos de tipos, estipulamos que este atributo será um valor em vírgula flutuante entre 0 e 100, sendo 100 o melhor estado possível e 0 o pior estado possível. Assim, este passa a ser uma variável de instância do tipo float da classe "Car".

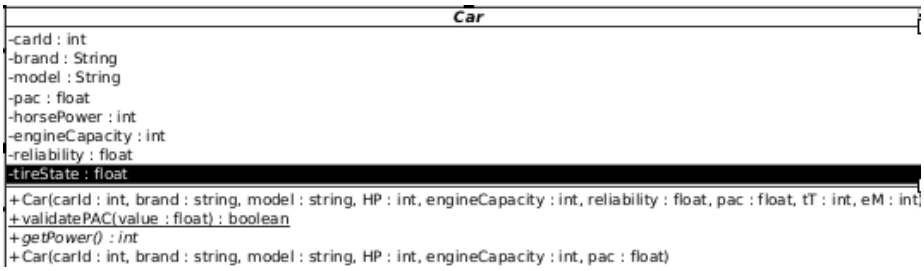

Figura 8: Atualização na classe "Car".

3 Esquema lógico da BD

Uma das fases do processo de preparação para a implementação física do sistema foi a elaboração do esquema lógico da base de dados. Nesta fase, optamos por um esquema lógico a englobar os tipos de dados a persistir nos vários subsistemas, uma vez que, para implementar as funcionalidades relativas ao processo de simulação, precisamos de entidades como carros, pilotos e circuitos.

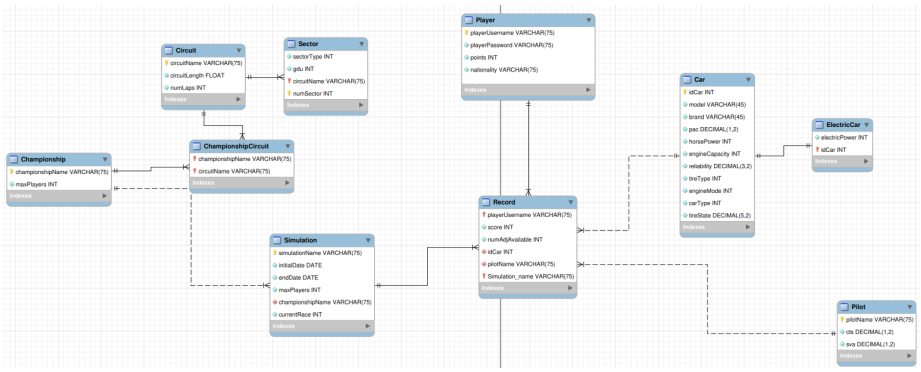

Figura 9: Modelo lógico da base de dados.

Após considerarmos quais os processos que podem ocorrer dentro do jogo, chegamos à conclusão que iríamos persistir as seguintes classes: Player, Pilot, Car, Circuit, Sector, Championship, Simulation e Record. As tabelas que irão ser geradas permitirão assim aos utilizadores da aplicação registar-se, criar pilotos, carros, circuitos, campeonatos, assim como simulações. É importante persistir os registos de cada jogador (Records) em disco, pois podemos querer, mais tarde, consultar as classificações de uma dada corrida, assim como recomeçar a simulação de um campeonato. A tabela *ChampionshipCircuit*, que não é mapeada diretamente numa classe da nossa aplicação, representa a possibilidade de um campeonato poder ter vários circuitos e um circuito estar presente em vários campeonatos. Exprime, por isso, um relacionamento N:N. Outro porme-

nor de análise é a representação do tipo de carro. Se, em java, é possível dar tipos aos objetos de uma dada superclasse, através do processo de generalização e herança, no paradigma relacional, isso já não é possível. Assim, resolvemos atribuir um inteiro a cada registo da tabela "Car" de modo a distinguir os carros em C1, C2, SC e GT. Temos noção de que esta não é a melhor opção, devido à normalização de dados, no entanto, é a que simplifica mais nesta fase precoce do desenvolvimento, onde estamos a dar um maior destaque à simulação.

4 Arquitetura com persistência de dados

Nesta fase do trabalho, necessitamos de pensar em aspetos mais ao nível da implementação do sistema. No entanto, a necessidade de modelar e de pensar no problema antes de o tentar resolver não foi descurada. Assim, um dos primeiros passos dados foi atualizar a arquitetura do sistema para permitir a persistência dos dados, neste caso, em Bases de Dados.

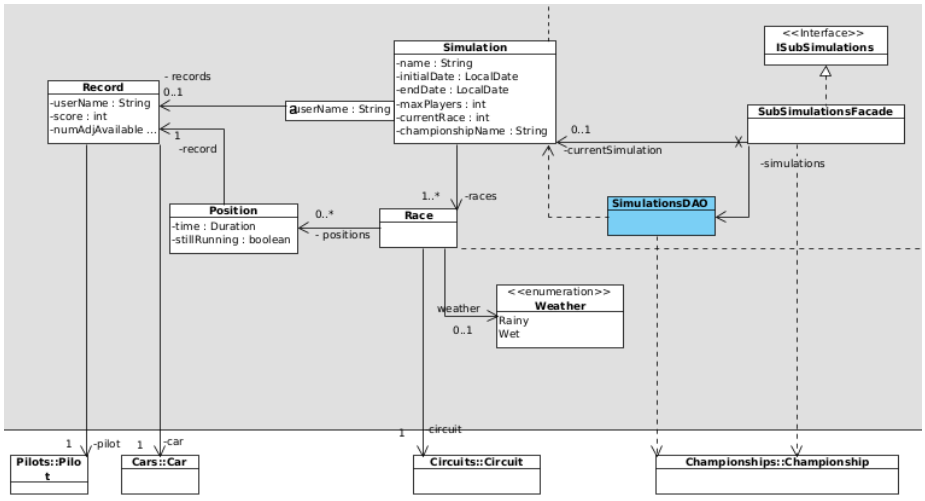

Figura 10: Arquitetura com persistência.

Uma alteração que foi necessária efetuar foi a adição de uma variável de instância "currentSimulation" à fachada deste subsistema. Uma vez que não serão persistadas as posições ocupadas pelos jogadores durante cada corrida, temos de guardar a simulação corrente entre as chamadas ao método "simulateNextLap".

Nos diagramas seguintes, podemos ver como a adição desta variável de instância teve repercussões nos diagramas de sequência elaborados para a fase 2.

Primeiramente, na operação "startSimulation", deixamos de aceder ao mapa em memória, sendo explícita a chamada do método *get* ao objeto "simulations" da classe "SimulationsDAO". Esta classe pretende encapsular o acesso à base de dados ou a outro tipo de armazenamento de dados, mesmo em memória. A API de acesso aos dados mantém-se, pelo que as alterações a realizar não foram muitas. É, também, nesta operação que atualizamos o valor da simulação atual em memória, após ter sido começada a simulação da próxima corrida.

Foi criado um script para criação da base de dados, com todas as tabelas e colunas descritos no modelo lógico. Foi criado também um utilizador de base de dados da nossa aplicação. Nesta fase precoce do desenvolvimento não possuímos qualquer mecanismo de criação de utilizadores, pelo que o administrador da nossa aplicação pode ser considerado o utilizador da base de dados.

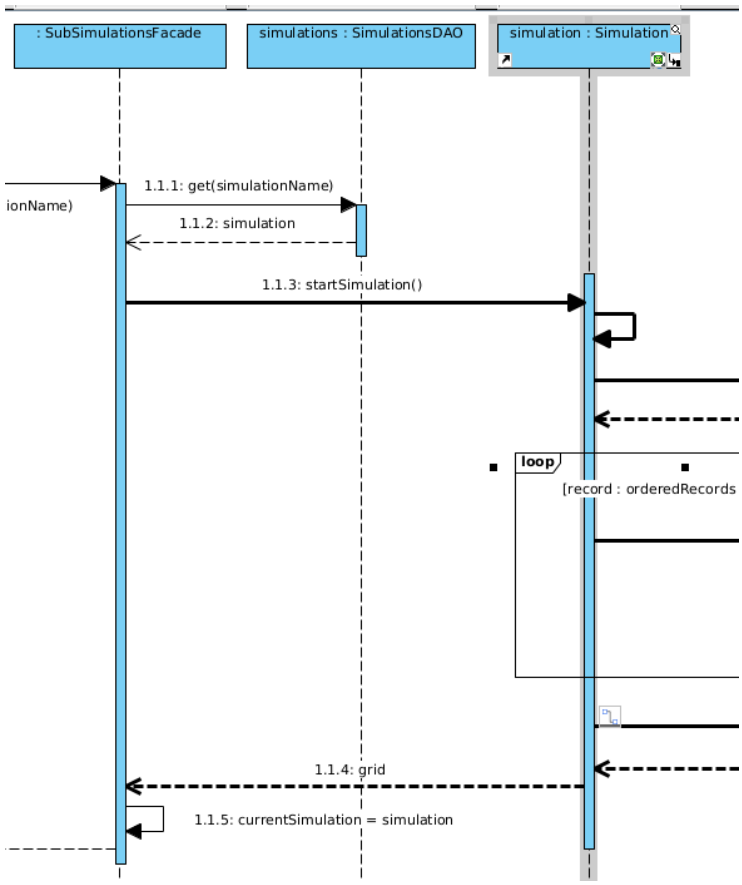

Figura 11: Alterações efetuadas no diagrama de sequência da operação "startSimulation".

As seguintes operações de simulação da próxima volta já serão chamadas no objeto em memória, sem haver qualquer consulta da base de dados.

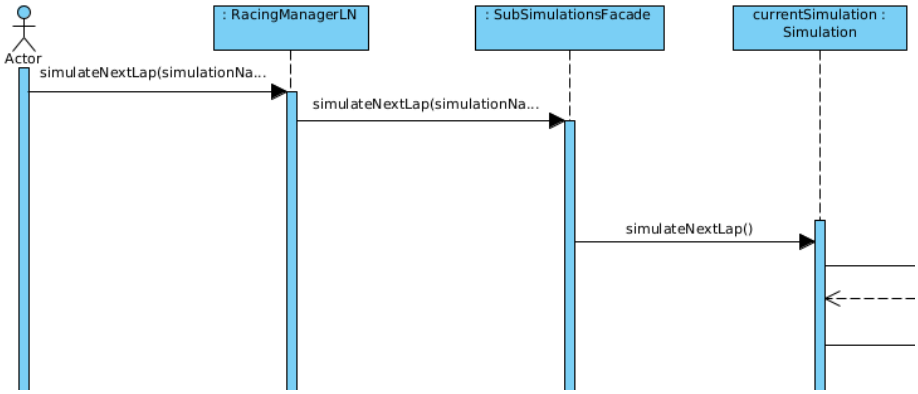

Figura 12: Alterações efetuadas no diagrama de sequência da operação "simulateNextLap"

Finalmente, outro pormenor relacionado à persistência de dados tem a ver com o facto de atualizações em memória não se refletirem de forma automática a atualizações em disco. Um exemplo disso é a operação "addRecord", onde um novo registo é adicionado a uma simulação. Sem a operação *put* final, o novo registo não seria visível em futuros acessos a essa simulação. A única exceção é quando o objeto está a ser tratado em memória, como acontece durante a simulação de uma corrida. No entanto, apesar de não ser aí necessária a escrita da simulação na base de dados (pois os únicos dados alterados foram os relativos à simulação, que não são persistidos), é fundamental a atualização da simulação quando terminada a simulação com o método "finishSimulation", que é responsável por atualizar as pontuações dos jogadores no campeonato que é simulado.

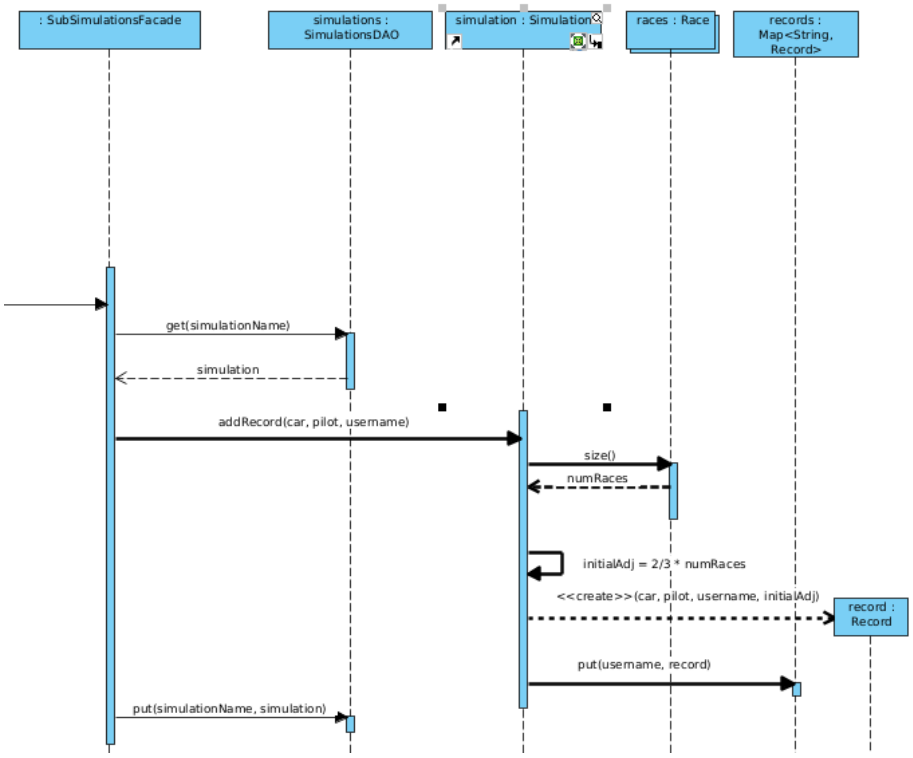

Figura 13: Alterações efetuadas no diagrama de sequência da operação "addRecord"

5 Povoamento da base de dados

De forma a agilizar o processo de povoar a base de dados, foi criado um script SQL para inserir alguns registos nas tabelas relacionais. No entanto, este processo pode ser feito via interface, algo que será um processo mais lento. Com isto, utilizamos esta via para testar as funcionalidades do nosso sistema. Em seguida, expõe-se o script usado para povoamento da base de dados.

```sql
INSERT INTO Pilot (pilotName, cts, sva)
VALUES ('Lewis Hamilton', 0.35, 0.4),
       ('Max Verstappen', 0.5, 0.8),
       ('Charles Leclerc', 0.6, 0.7),
       ('Lando Norris', 0.7, 0.5),
       ('Nicholas Latifi', 0.6, 0.3);

INSERT INTO Player (playerUsername, playerPassword, points, country)
VALUES ('varelzzz', '12345', 0, 'Portugal'),
       ('gabs', 'ola123', 0, 'Portugal'),
       ('tonegamer', '123ole', 0, 'França'),
       ('mikeb', 'abcd', 0, 'Portugal'),
       ('tunes', '987654321', 0, 'Marrocos');

INSERT INTO Circuit (circuitName, circuitLength, numLaps)
VALUES ('Monaco', 3.3, 5),
       ('Gualtar Campus', 3, 7);

INSERT INTO Championship (championshipName, maxPlayers)
VALUES ('DSS Lovers', 10);

INSERT INTO Sector (sectorType, gdu, circuitName, numSector)
VALUES (0, 1, 'Monaco', 0),
       (1, 0, 'Monaco', 1),
       (2, 1, 'Monaco', 2),
       (0, 1, 'Gualtar Campus', 0),
       (1, 0, 'Gualtar Campus', 1),
       (2, 2, 'Gualtar Campus', 2),
       (0, 1, 'Gualtar Campus', 3),
       (1, 0, 'Gualtar Campus', 4),
       (0, 1, 'Gualtar Campus', 5);

INSERT INTO Car (model, brand, pac, horsePower, engineCapacity, reliability, tireType, engineMode, carType)
VALUES ('W13','Mercedes', 0.97, 1070, 6000, 0.96, 1, 0, 1),
       ('RB18','RedBull', 0.94, 1080, 6000, 0.95, 1, 0, 1),
       ('F1-75','Ferrari', 0.93, 1075, 6000, 0.94, 1, 0, 1),
       ('MCL36','McLaren', 0.93, 1065, 6000, 0.94, 1, 0, 1),
       ('A522','Alpine', 0.95, 1065, 6000, 0.94, 1, 0, 1);

INSERT INTO ChampionshipCircuit(championshipName, circuitName)
VALUES ('DSS Lovers', 'Monaco'),
       ('DSS Lovers', 'Gualtar Campus');

INSERT INTO Simulation (simulationName, initialDate, endDate, maxPlayers, championshipName)
VALUES ('UMinho', '2022-12-17', NULL, 4, 'DSS Lovers');

INSERT INTO Record (playerUsername, score, numAdjAvailable, idCar, pilotName, simulationName)
VALUES ('mikeb', 0, 2, 1, 'Max Verstappen', 'UMinho'),
       ('tunes', 0, 2, 2, 'Max Verstappen', 'UMinho'),
       ('varelzzz', 0, 2, 3, 'Nicholas Latifi', 'UMinho'),
       ('gabs', 0, 2, 4, 'Charles Leclerc', 'UMinho'),
       ('tonegamer', 0, 2, 5, 'Lando Norris', 'UMinho');
```

Figura 14: Povoamento da base de dados.

6 Implementação do sistema

Tal como tínhamos detalhado na segunda fase do trabalho, a nossa aplicação será estruturada em 3 níveis/camadas: camada de interface com o utilizador, camada da lógica de negócio e camada de dados. O nosso foco estendeu-se por todas as camadas, sendo que na lógica de negócio focou-se mais concretamente no subsistema das simulações.

6.1 *User Interface*

6.1.1 MVC

Um dos aspetos não considerados durante a fase de modelação conceptual e estrutural da nossa aplicação foi a interface entre o componente da interface com o utlizador e o componente da lógica de negócios. Numa arquitetura mais simplificada poderia ser adicionado diretamente à interface com o utilizador uma instância da fachada da lógica de negócio. No entanto, do ponto de vista da escalabilidade da nossa aplicação é mais útil que possa existir um elo de ligação entre os dois componentes, com o nome de controlador. Uma das utilidades deste componente passa por garantir a independência entre as *Views* e o *Model*, sendo a grande responsabilidade deste encaminhar as mensagens da *view* para o *model* (e vice-versa), gerindo os tipos de mensagens enviadas de forma a satisfazer as APIs disponibilizadas.

6.1.2 Funcionalidades da interface

A camada de interface com o utilizador consiste, neste momento, numa sequência de menus onde o utilizador pode efetuar login e se registar como jogador ou administrador, sendo que se o efetuar como administrador pode criar pilotos, campeonatos, carros, simulaçóes, etc. De seguida, se estiver autenticado como jogador pode escolher a simulação a jogar , e simular as corridas relativas a essa simulação. Também é capaz de ver a classificação global antes de escolher uma simulação. Quando uma corrida começa é mostrada a grelha inicial e o tempo metereológico previsto e quando uma corrida acaba o jogador é capaz de ver a classificação do campeonato até essa altura, sendo que é mostrada a grelha final. Também é capaz de ver a grelha inicial no ínicio de uma simulação e a grelha final no final desta. As interações possíveis entre o cliente e a aplicação são as especificadas em cada use case.

6.2 Lógica de negócio

6.2.1 Simulação normal

A simulação normal tem apenas em conta as posições relativas entre os jogadores. Tal como fica evidenciado no diagrama de sequência da operação "simulateNextLap", necessitamos de uma função que verifique se o carro sofre uma avaria ou não. Assim, precisamos de definir uma probabilidade de o carro sofrer uma avaria. A probabilidade terá em consideração os seguintes fatores: fiabilidade do carro e modo de funcionamento do motor.

```java
private boolean isBrokenDownCar(int positionIndex) {
    Position p = this.positions.get(positionIndex);
    Car playerCar = p.getRecord().getCar();
    float reliability = playerCar.getReliability();
    EngineMode em = playerCar.getEngineMode();

    float engineAddedProb=0;

    switch (em){
        case CONSERVATIVE -> engineAddedProb = +0.1f;
        case AGGRESSIVE -> engineAddedProb = -0.1f;
    }

    // total probability = reliability + engineAddedProb
    return Race.simulateProbability(1 - (reliability + engineAddedProb));
}
```

Figura 15: Simulação de avaria na simulação de um carro

Após ter sido feita a verificação de avaria nos carros, é simulada a volta com a simulação de cada um dos setores da mesma. Em cada setor e para cada carro, começa-se por verificar se há um despiste do mesmo, definindo um fator de risco. Este fator depende do grau de dificuldade de ultrapassagem do setor e do tipo de setor, da conjugação das condições metereológicas com o critério CTS e do critério SVA do piloto. Note-se que, havendo um despiste de um carro, esse carro só pode descer no máximo uma posição na corrida.

Por fim, definimos a função "canOvertakeNextCar" que é responsável por avaliar se há ultrapassagens entre dois carros em posições contíguas. Mais uma vez, definimos a probabilidade de haver ultrapassagem como a combinação de vários fatores: fator de categorias (se um carro for de categoria superior tem mais probabilidade de utrapassar), fator de SVAs (um condutor com um SVA mais alto tem mais probabilidade de ultrapassar um condutor com um SVA baixo), fator de pneus que verifica os tipo de pneus dos dois carros, comparando-os, o fator de downforce que aumenta a probabilidade de ultrapassagem em curva para carros com este valor mais alto e o fator do sector que depende apenas do grau de ultrapassagem no setor.

```java
private boolean canOvertakeNextCar(int positionIndex, Sector s) {
    Car currentCar = this.positions.get(positionIndex).getRecord().getCar();
    Car nextCar = this.positions.get(positionIndex).getRecord().getCar();
    Pilot currentPilot = this.positions.get(positionIndex).getRecord().getPilot();
    Pilot nextPilot = this.positions.get(positionIndex).getRecord().getPilot();

    float categoriesFactor = Race.calculateCategoryFactor(currentCar, nextCar);
    float svasFactor = currentPilot.getSva() - nextPilot.getSva();
    float tireFactor = currentCar.getTireState()/100 - nextCar.getTireState()/100;
    float downforceFactor = Race.calculateDownforceFactor(currentCar.getPac(), s);
    float sectorFactor = Race.calculateSectorFactor(s);

    return Race.simulateProbability(0.25f + categoriesFactor + svasFactor + tireFactor + downforceFactor + sectorFactor);
}
```

Figura 16: Simulação de ultrapassagem

6.2.2 Simulação premium

O processo de simulação tem por base as diferenças de tempos entre jogadores. A variável *time*, da classe *Position*, dá-nos o tempo "gasto" por esse jogador até à volta atual, em milisegundos. Por exemplo, um tempo de 90000 no início da segunda volta indica que se passaram 90s até esse jogador passar na linha de partida para a segunda volta. Na nossa implementação consideramos que a simulação é realizada volta a volta e não setor a setor, sendo que os tempos de cada jogador são incrementados em função da prestação em cada volta. Ao contrário da simulação normal, as ultrapassagens são implícitas uma vez que estas acontecem dependendo dos tempos no final de cada volta. A nossa abordagem teve por base ideias retiradas do código legado, como o cálculo do tempo de realização da volta (*lapTime*). Este tempo depende de inúmeros fatores, alguns deles iguais para todos os carros, como o tamanho do circuito e o estado do tempo, outros diferentes, como os critérios SVA e CTS, o tipo de pneus e o tipo de carro. Isto permite uma maior flexibilidade na simulação de cada volta.

```
private long getLapTime(int position) {
    Car c = this.positions.get(position).getRecord().getCar();
    Pilot p = this.positions.get(position).getRecord().getPilot();
    Weather w = this.weather;

    long averageLapTime; // milliseconds
    if (w == Weather.RAINY) {
        averageLapTime = 3000 + (long) (p.getCts() * 1000);
        switch (c.getTireType()) {
            case SOFT -> averageLapTime += 2000;
            case HARD -> averageLapTime += 1000;
        }
    } else {
        averageLapTime = 1000 - (long) (1000 * p.getCts());

        switch (c.getTireType()) {
            case HARD -> averageLapTime += 1000;
            case RAIN -> averageLapTime += 2000;
        }
    }
    averageLapTime += p.getSva() * 1000;
    averageLapTime += c.getPac() * 1000;

    switch (c.getEngineMode()) {
        case CONSERVATIVE -> averageLapTime += 1000;
        case AGGRESSIVE -> averageLapTime -= 1000;
    }

    if (c instanceof C1) {
        averageLapTime += (long) (circuit.getLength() * 23550); // km * s/km = s monaco track
    } else if (c instanceof C2) {
        averageLapTime += (long) (circuit.getLength() * 24550);
    } else if (c instanceof GT) {
        averageLapTime += (long) (circuit.getLength() * 26550);
    } else if (c instanceof SC) {
        averageLapTime += (long) (circuit.getLength() * 27550);
    }

    averageLapTime += (c.getEngineCapacity()/c.getPower()) * 1000L;
```

Figura 17: Cálculo do tempo de uma volta

6.2.3 Tratamento de exceções

Uma das preocupações tida no desenvolvimento da nossa aplicação teve a ver com o tratamento de erros, resultado da inserção de valores inválidos (caso da downforce, que deve ser validada) ou de comportamentos inválidos do programa (por exemplo, se o número de afinações para um dado carro tiver sido atingido). Foram assim criadas as seguintes exceções:

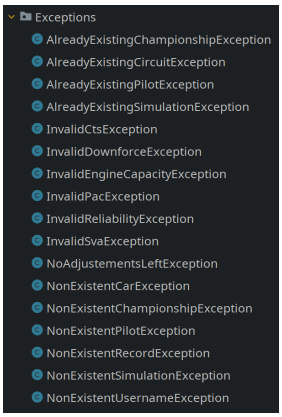

Figura 18: Exceções criadas

Por exemplo, considerando o excerto do use case "Configurar Corridas" apresentado abaixo, é possível que o utilizador tenha inserido um valor de downforce inválido. Quem valida o valor da downforce deve ser a classe carro, pelo que um valor inválido origina uma exceção no método *setPac*, sendo mais tarde apanhada na View.

Fluxo Normal		4. Valida o valor da downforce.		Validar o valor da downforce	addAdjustment(champName: String, userName: String, valueDownforce : float, engineMode : int) : void
		5. Verifica que o carro pode ser afinado.		Verifica se o carro pode sofrer alterações	
		6. Calcula o número de afinações restantes do utilizador.		Calcular o número de afinações	
		7. Regista alterações.		Registar as afinações do carro	
	8. Escolhe o tipo de pneus.		UI	---	---
		9. Regista tipo de pneu escolhido.		Registar tipo de pneu	setTiresType(champName : String, userName : String,tiresType : int) : void

Figura 19: Possíveis erros na interação com o cliente

6.3 Camada de dados - DAOs

Para efetuar a conversão do modelo relacional para o modelo orientado aos objetos, procedemos à criação de um mapeamento objeto-relacional. Para isso, recorremos aos *Data Access Objects* (DAO). O primeiro passo para o acesso à base de dados ser realizado de forma correta é a instalação do *driver* JDBC de acesso à base de dados. Este encontra-se numa pasta *lib* dentro do repositório para que seja imediata a sua configuração. Neste trabalho, o motor de base de dados usado é o *MariaDB*. No entanto, o uso de um motor de base de dados *MySql* seria semelhante. A configuração para acesso à base de dados, através do utilizador criado anteriormente são os seguintes:

```java
public class DAOconfig {
    35 usages
    public static final String USERNAME = "admin";
    35 usages
    public static final String PASSWORD = "admin123";
    1 usage
    private static final String DATABASE = "RacingManager";
    1 usage
    private static final String DRIVER = "jdbc:mariadb";        // Usar para MariaDB
    //private static final String DRIVER = "jdbc:mysql";        // Usar para MySQL
    35 usages
    public static final String URL = DRIVER+"://localhost/"+DATABASE;
}
```

Figura 20: Informação para acesso à base de dados

Um dos cuidados tidos na elaboração destas classes de acesso a dados foi o da segurança. O bem conhecido ataque "SQL Injection" resulta da inserção de código malicioso em expressões SQL (a partir da interface com o utilizador) como forma de aceder à base de dados. Uma forma de prevenir isso é com o recurso a *PreparedStatements*.

```java
@Override
public Circuit get(Object key) {
    Circuit circuit = null;

    try (Connection c = DriverManager.getConnection(DAOconfig.URL, DAOconfig.USERNAME, DAOconfig.PASSWORD);
         PreparedStatement st1 = c.prepareStatement( sql: "SELECT * FROM Circuit WHERE circuitName=?;");
         PreparedStatement st2 = c.prepareStatement( sql: "SELECT * FROM Sector WHERE circuitName=?;")) {

        st1.setString( parameterIndex: 1, key.toString());
        ResultSet rs1 = st1.executeQuery();
        st2.setString( parameterIndex: 1, key.toString());
        ResultSet rs2 = st2.executeQuery();

        List<Sector> sectors = new ArrayList<>();
        while (rs2.next()) {
            int sectorType = rs2.getInt( columnIndex: 1);
            int gdu = rs2.getInt( columnIndex: 2);
            GDU g = switch (gdu) {
                case 0 -> GDU.POSSIBLE;
                case 1 -> GDU.HARD;
                case 2 -> GDU.IMPOSSIBLE;
                default -> null;
                // throw exception
            };
```

Figura 21: Proteção dos acessos à base de dados

7 Manual de utilização

De modo a utilizar a aplicação, o utilizador deverá, primeiramente, descarregar o repositório, onde já vem incluídas as bibliotecas necessárias ao acesso à

base de dados. Em seguida, será preciso configurar as bibliotecas importadas pelo executável, selecionando o menu "Project Structure" no Intellij e depois adicionando a biblioteca "maria-db-java-cliente", que se encontra na pasta lib.

Será preciso, ainda, configurar os dados de acesso à base de dados, nomeadamente o *USERNAME* e a *PASSWORD*. Uma das formas é adicionar o utilizador *admin* à base de dados, correndo o seguinte script:

```
CREATE USER 'admin'@localhost IDENTIFIED BY 'admin123';
grant all privileges on *.* to 'admin'@localhost;
flush privileges;
```

Figura 22: Dados de acesso à base de dados

Outra forma é alterar os dados de acesso no ficheiro DAOconfig, adicionando o utilizador "root" e a respetiva palavra-passe.

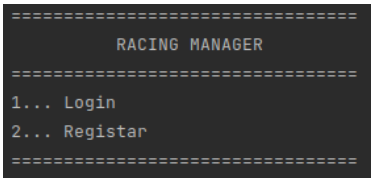

Figura 23: Menu de Login/Registo.

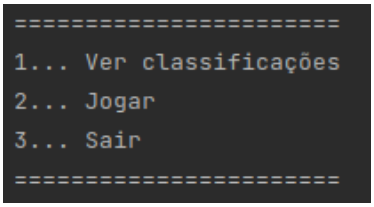

Figura 24: Menu para consultar classificações ou seguir para simulação.

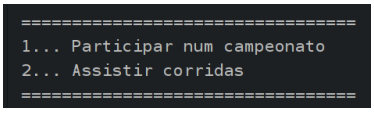

Figura 25: Menu relativo às simulações.

8 Conclusões finais

Nesta última fase do trabalho de Desenvolvimento de Sistemas de Software procedemos à implementação de algumas funcionalidades anteriormente levantadas. Num primeiro passo, procuramos entender que mudanças a nossa arquitetura teria de sofrer para poder suportar a persistência de dados. Em seguida, implementamos as funcionalidades relativas ao cenário 5, com especial destaque para a simulação. O processo de implementação foi complementado por um processo simultâneo de *testing* das funcionalidades. Não nos foi possível implementar da melhor maneira possível tudo aquilo que estava estipulado na fase de levantamento de requisitos. Temos consciência de que podíamos melhorar vários aspetos do nosso trabalho como a arquitetura do sistema e da base de dados, o processo de simulação das corridas ou até a interface gráfica, mas, por falta de melhor organização, não nos foi possível concretizar.

Fazendo uma retrospetiva final do trabalho, tendo em conta as 3 fases, foi-nos possível acompanhar o desenvolvimento de uma aplicação (neste caso um jogo) desde a sua fase de análise de requisitos até à modelação arquitetural e, mais tarde, implementação. Foi-nos também possível observar que o processo de desenvolvimento não é um processo em cascata, mas sim um processo iterativo no qual é possível corrigir decisões anteriores com base em novo conhecimento adquirido do problema. A modelação, através de diagramas, permitiu-nos pensar nos impactos que as diferentes alternativas, quer para a modelação estrutural, quer para a modelação comportamental iriam causar no sistema.