



Universidade do Minho  
Escola de Engenharia

# *Racing Manager*

## Conceção 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

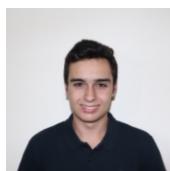
Nuno Guilherme Cruz Varela - a96455



a97393



a96215



a95019



a97698



a96455

**Repositório GitHub:** <https://github.com/GVarelaaa/ProjetoDSS39>

novembro, 2022

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>4</b>
<b>2</b>	<b>Atualizações relativamente à Fase 1</b>	<b>4</b>
<b>3</b>	<b>APIs e Subsistemas</b>	<b>6</b>
3.1	Separação em subsistemas campeonato e simulação . . . . .	8
<b>4</b>	<b>Diagrama de componentes</b>	<b>9</b>
<b>5</b>	<b>Diagrama de <i>packages</i></b>	<b>10</b>
<b>6</b>	<b>Diagrama de classes</b>	<b>11</b>
<b>7</b>	<b>Código legado</b>	<b>13</b>
<b>8</b>	<b>Diagramas de sequência</b>	<b>14</b>
8.1	Operação “getGlobalClassification“ . . . . .	14
8.2	Operação “startSimulation“ . . . . .	15
8.3	Operação “simulateNextLap“ . . . . .	16
<b>9</b>	<b>Diagrama de atividades</b>	<b>19</b>
<b>10</b>	<b>Conclusão</b>	<b>20</b>
<b>11</b>	<b>Anexos</b>	<b>21</b>
11.1	Diagramas de classes . . . . .	21
11.1.1	RacingManagerLN . . . . .	21
11.1.2	SubUsers . . . . .	22
11.1.3	SubCircuits . . . . .	22
11.1.4	SubCars . . . . .	23
11.1.5	SubPilots . . . . .	23
11.1.6	SubChampionships . . . . .	24
11.1.7	SubSimulations . . . . .	24
11.2	Diagrama de componentes . . . . .	25
11.3	Diagrama de <i>packages</i> . . . . .	26
11.4	Diagramas de sequência . . . . .	26
11.4.1	“addSimulation“ . . . . .	26
11.4.2	“addRecord“ . . . . .	27
11.4.3	“startSimulation“ . . . . .	27
11.4.4	“getWeather“ . . . . .	28
11.4.5	“addAdjustment“ . . . . .	28
11.4.6	“simulateNextLap“ . . . . .	29
11.4.7	“simulateSector“ . . . . .	30
11.4.8	“showGrid“ . . . . .	31
11.4.9	“finishSimulation“ . . . . .	31

11.4.10 “getCategoryClassification“ . . . . .	32
11.4.11 “getGlobalClassification“ . . . . .	32
11.5 Diagrama de atividades . . . . .	33

## **Lista de Figuras**

1 Alteração no modelo de domínio. . . . .	5
2 Operações de criar piloto. . . . .	6
3 Operações de participar no campeonato. . . . .	7
4 Diagrama de componentes. . . . .	9
5 Diagrama de <i>packages</i> . . . . .	10
6 Interface da lógica de negócio. . . . .	11
7 Diagrama de classes do subsistema das simulações. . . . .	12
8 Diagrama de classes do código legado. . . . .	13
9 Diagrama de sequência da operação “getGlobalClassification“ por datas. . . . .	15
10 Diagrama de sequência da operação “startSimulation“. . . . .	16
11 Pré-condição em OCL. . . . .	16
12 Diagrama de sequência da operação “simulateNextLap“. . . . .	17
13 Diagrama de sequência da operação “simulateSector“. . . . .	18
14 Diagrama de atividades relativo ao processo de jogar. . . . .	19
15 Diagrama de classes relativo à fachada da lógica de negócios. . . . .	21
16 Diagrama de classes relativo ao subsistema dos utilizadores. . . . .	22
17 Diagrama de classes relativo ao subsistema dos circuitos. . . . .	22
18 Diagrama de classes relativo ao subsistema dos carros. . . . .	23
19 Diagrama de classes relativo ao subsistema dos pilotos. . . . .	23
20 Diagrama de classes relativo ao subsistema dos campeonatos. . . . .	24
21 Diagrama de classes relativo ao subsistema das simulações. . . . .	24
22 Diagrama de componentes. . . . .	25
23 Diagrama de <i>packages</i> . . . . .	26
24 Diagrama de sequência da operação “addSimulation“. . . . .	26
25 Diagrama de sequência da operação “addRecord“. . . . .	27
26 Diagrama de sequência da operação “startSimulation“. . . . .	27
27 Diagrama de sequência da operação “getWeather“. . . . .	28
28 Diagrama de sequência da operação “addAdjustment“. . . . .	28
29 Diagrama de sequência da operação “simulateNextLap“. . . . .	29
30 Diagrama de sequência da operação “simulateSector“. . . . .	30
31 Diagrama de sequência da operação “showGrid“. . . . .	31
32 Diagrama de sequência da operação “finishSimulation“. . . . .	31
33 Diagrama de sequência da operação “getCategoryClassification“. . . . .	32
34 Diagrama de sequência da operação “getGlobalClassification“ . . . . .	32
35 Diagrama de atividades do processo jogar. . . . .	33

## 1 Introdução

Para esta fase do trabalho, apresentamos a nossa proposta de arquitetura conceitual do sistema e os modelos comportamentais necessários para descrever o comportamento pretendido para o sistema.

Este processo surge no seguimento do processo de análise de requisitos, pelo que a conclusão com sucesso da fase anterior é fundamental. Contudo, ao longo desta fase foram surgindo algumas inconsistências no modelo de domínio e na especificação dos *use cases* apresentados.

Neste relatório, começamos, então, por abordar quais os aspectos que foram aprimorados relativamente à primeira entrega, seguindo-se a apresentação das interfaces do sistema e dos seus subsistemas, resumidos no diagrama de componentes. Posteriormente, apresentamos o diagrama de *packages* e procedemos à explicação do modelo de classes para a lógica de negócio, suportando essa arquitetura com alguns diagramas de sequência que consideramos relevantes. Para terminar, apresentamos o exemplo de um diagrama de atividades que visa descrever o processo de jogo.

Para a definição destes elementos, procuramos seguir uma abordagem orientada aos requisitos. Sendo assim, cada diagrama elaborado deverá ser assente em um ou mais requisitos do sistema, que poderão ser ou não materializados em *use cases*. Procuramos, deste modo, ganhar uma maior rastreabilidade na definição do sistema.

## 2 Atualizações relativamente à Fase 1

Nesta secção do relatório apresentamos as melhorias efetuadas no modelo de domínio e na especificação dos requisitos apresentados na primeira fase.

Começando pelo modelo de domínio, a única alteração efetuada foi na entidade “Número de voltas”.

No modelo anterior, o número de voltas era apresentado como um atributo da entidade “Corrida”. Apesar de isso fazer sentido do ponto de vista conceitual, não faz sentido do ponto de vista das funcionalidades oferecidas pela nossa aplicação. Na prática, é o jogador que começa uma dada corrida e o administrador que cria o circuito que, mais tarde, fará parte das corridas de um dado campeonato. Assim, faz sentido que o número de voltas já esteja predefinido no circuito e que seja o administrador a defini-lo, sendo esse valor constante para todas as corridas realizadas nesse circuito.

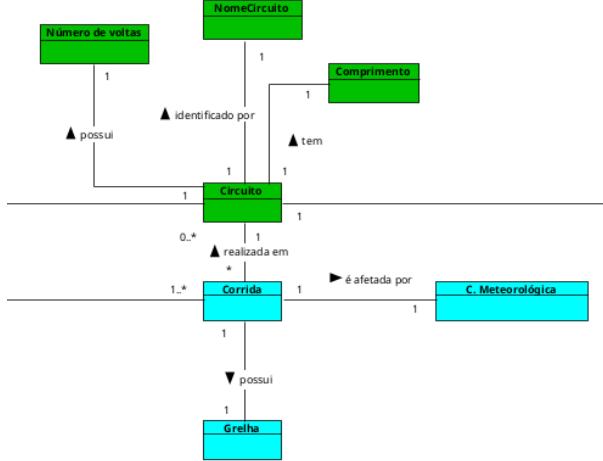


Figura 1: Alteração no modelo de domínio.

Relativamente aos *use cases*, o que sofreu mais alterações foi o “Assistir volta“. A nossa concepção inicial do sistema era de um jogo distribuído, onde vários utilizadores poderiam assistir ao desenrolar da corrida com vistas diferentes, focadas nos eventos de interesse para esse dado utilizador. No entanto, isso complicava em demasia o nosso sistema, pois teríamos múltiplos UIs. Assim, a solução passou por combinar a simulação da corrida de todos os jogadores. Agora, do ponto de vista da única UI existente, a atualização das posições de cada carro e dos restantes eventos possíveis são apresentadas numa única vista, sendo necessária apenas uma operação de mudança de estado. Posteriormente no relatório, explicaremos esta operação em detalhe.

Outro aspecto que foi aprimorado nesta fase foi a sequência de interações entre o utilizador e o sistema em cada *use case*. Mais concretamente, foram revistos os passos em que o sistema validava os dados introduzidos pelo utilizador. Na primeira fase, as validações eram feitas imediatamente após o utilizador inserir um dado valor. Nesta segunda fase, as validações são realizadas apenas no final, juntamente com a operação principal. Esta alteração foi detetada quando começámos a elaborar os diagramas de sequência para cada operação da interface da lógica de negócios. Na secção seguinte, apresentaremos alguns exemplos de *use cases* onde esta situação se evidenciou.

### 3 APIs e Subsistemas

Nesta secção apresentamos o processo de identificação das operações que o modelo da lógica de negócio do nosso sistema deve suportar. A identificação destas operações é fundamentada pelos *use cases* anteriormente definidos. Deste modo, para cada *use case*, definimos as responsabilidades da camada da lógica de negócio na forma de operações que esta camada deve exportar para a camada de interface com o utilizador. Ao mesmo tempo, dividimos as operações encontradas em diferentes subsistemas.

Para exemplificar o processo, utilizamos como exemplo o *use case* “Criar Piloto”:

Use Case	Criar piloto	1. Dividir os fluxos em sequências de transações	2. Identificar responsabilidades da LN	3. definir API (identificar métodos)	4. identificar subsistemas (agrupar métodos)
Ator	Administrador				
Cenários	Cenario 7				
Descrição	O administrador cria e configura um novo piloto.				
Pré-Condição	O utilizador está autenticado como administrador.				
Pós-Condição	O novo piloto é adicionado à lista de pilotos disponíveis.				
	Ator	Sistema			
Fluxo Normal	1. Indica o nome do piloto.	UI	---	---	---
	2. Indica o nível de pericia do piloto no critério “Chuva vs Tempo Seco” (CTS).	UI	---	----	---
	3. Indica o nível de pericia do piloto no critério “Segurança vs Agressividade” (SVA).	UI	----	----	----
	4. Valida o nome do piloto.		Validar o nome do piloto		
	5. Valida o valor introduzido no critério CTS		Validar o valor CTS		
	6. Valida o valor introduzido no critério SVA.		Validar o valor SVA	addPilot(name: String, valueCTS : float, valueSVA : float) : void	subPilots
	7. Regista e adiciona o novo piloto à lista de pilotos disponíveis.		Adicionar piloto à lista dos pilotos		
Fluxo Alternativo [Níveis de pericia standard] (Passos 2 e 3)	3.1. Indica que quer os valores default para os níveis de pericia.	UI	---	---	---
	3.2. Valida o nome do piloto.		Validar o nome do piloto.		
	3.3. Regista e adiciona o novo piloto à lista de pilotos disponíveis.		Adicionar piloto à lista dos pilotos	addPilot(name: String) : boolean	subPilots

Figura 2: Operações de criar piloto.

Neste *use case*, podemos identificar uma operação denominada “addPilot” que adiciona um novo piloto ao sistema. No entanto, esta operação tem por base outras operações mais simples de verificação como, por exemplo, validar se o nome do piloto já não existe no sistema. Na verdade, podemos considerar que estas são as pré-condições para uma invocação sem erros da operação “addPilot”. Como tal, erros nestas validações irão gerar situações de exceção que deverão ser tratadas pela UI. Outra possibilidade, diferente da que está presente no exemplo acima, seria testar a validade dos valores introduzidos pelo utilizador imediatamente após eles serem inseridos. No entanto, isso levaria a um processo menos elegante e eficiente, visto que a UI teria de ter acesso a mais operações da LN.

O mesmo acontece no *use case* “Participar no campeonato“. Olhando para a figura 3, podemos ver como, mais uma vez, as operações de validar se um carro existe ou, por exemplo, se um piloto existe são encapsuladas numa única operação, “addRecord“, que recebe o nome do campeonato, o ID do carro, o ID do piloto e o nome do utilizador. A operação verificará situações de inconsistência, levantando exceções sempre que algum dos erros possíveis se verificar.

		1. Dividir os fluxos em sequências de transações	2. Identificar responsabilidades da LN	3. definir API (identificar métodos)	4. identificar sub sistemas (agrupar métodos)
Use Case	Participar no campeonato				
Ator	Jogador				
Cenários	Cenário 8				
Descrição	Jogador escolhe o carro, o piloto e prepara-se para entrar no campeonato				
Pré-Condição	O utilizador está autenticado como jogador.				
Pós-Condição	O jogador está pronto para participar no campeonato.				
	Ator	Sistema			
Fluxo Normal	1. Lista opções “Começar campeonato” e “Juntar-se a campeonato existente”.	UI	---	---	---
	2. Escolhe “Juntar-se a campeonato existente”.	UI	---	---	---
	3. Apresenta os campeonatos que o utilizador pode escolher.		Apresentar os vários campeonatos criados pelo administrador	showSimulations(): String	subChampionships
	4. Escolhe o campeonato.	UI	---	---	---
	5. Apresenta os carros que o utilizador pode escolher.		Apresentar os vários carros disponíveis	showAllCars(): String	subCars
	6. Escolhe o carro com o qual vai participar nas diversas corridas.	UI	---	---	---
	7. Apresenta os pilotos que o utilizador pode escolher.		Apresentar os vários pilotos disponíveis	showAllPilots(): String	subPilots
	8. Escolhe o piloto que o vai representar.	UI	---	---	---
	9. Valida se o campeonato existe no sistema.		Validar se a simulação existe no sistema		subSimulations
	10. Valida se o carro existe no sistema.		Validar se o carro é válido		subCars
	11. Valida se o piloto existe no sistema.		Validar se o piloto é válido		subPilots
	12. Valida se o utilizador existe no sistema.		Validar se o utilizador é válido	addRecord(simulationName: String, carID: int, pilotID: int, username: String): void	subUsers
	13. Valida se há lugar para mais jogadores no campeonato.		Validar se o número máximo de pilotos foi atingido		
	14. Adiciona o registo do jogador ao campeonato.		Adicionar registo do jogador a um campeonato		subSimulations

Figura 3: Operações de participar no campeonato.

No *use case* anterior visualizamos uma interação mais prolongada entre a camada da interface com o utilizador e a camada da lógica de negócio. Uma das novas alterações relativamente à primeira fase foi a possibilidade da UI poder receber informação sobre o conjunto dos campeonatos, carros e pilotos que existem no sistema. Para suportar estas funcionalidades, foram adicionadas as operações “showChampionships“, “showSimulations“, “showAllCars“, “showAllPilots“ e “showAllCircuits“.

### **3.1 Separação em subsistemas campeonato e simulação**

Numa primeira abordagem, optamos por dividir o problema em 5 subsistemas: carros, pilotos, circuitos, utilizadores e campeonatos, em que o subsistema dos campeonatos albergava os métodos relativos à simulação.

Contudo, esta abordagem levantava problemas na medida em que os campeonatos não estavam a ser guardados num subsistema à parte, de tal modo que pudessem ser jogados novamente, isto é, o utilizador iria jogar um campeonato e, no final, este seria descartado. Com isto, era possível haver situações em que os utilizadores não teriam campeonatos disponíveis para jogar.

De modo a ultrapassar este obstáculo, decidimos criar um novo subsistema para simulações, o que nos permite guardar os campeonatos, sempre que são criados pelo administrador, num subsistema de campeonatos à parte, tal como os carros, pilotos, utilizadores e circuitos. Desta maneira, quando se pretende começar um novo campeonato, o utilizador escolhe um dos campeonatos criados pelo administrador (presentes no subsistema dos campeonatos) e adiciona-se uma nova simulação à fachada das simulações.

A lógica de negócio passa, assim, a ser estruturada por 6 subsistemas. Temos consciência que não é a melhor divisão, uma vez que vamos aumentar o número de dependências entre subsistemas, algo que será explicado mais adiante. No entanto, garante-nos uma melhor manutenção do sistema no futuro.

## 4 Diagrama de componentes

A arquitetura do nosso sistema será uma arquitetura com 3 camadas: camada da interface com o utilizador, camada da lógica de negócios e camada de dados. É fulcral que estas 3 camadas possam operar de forma independente, pelo que é necessário definir as interfaces que fazem a fronteira entre os diversos subsistemas.

Focando na camada da lógica de negócios e com base na análise detalhada dos *use cases*, verificamos que esta será constituída por 6 componentes correspondentes aos subsistemas dos utilizadores, carros, pilotos, circuitos, campeonatos e simulações. Para além disso, cada um dos subsistemas representa uma área muito específica da nossa aplicação, sendo útil para a possibilidade de, no futuro, acrescentar funcionalidades numa única vertente do sistema sem muitas preocupações.

Quanto à camada da ligação de dados, é importante garantir que alterações nas estruturas onde os dados são armazenados não tenham um impacto significativo nas camadas acima. Como tal, esta camada deverá fornecer uma API semelhante à das estruturas em java, encapsulando operações de criação, leitura, atualização e remoção de dados em métodos com uma assinatura igual àquela que é fornecida pelo java.

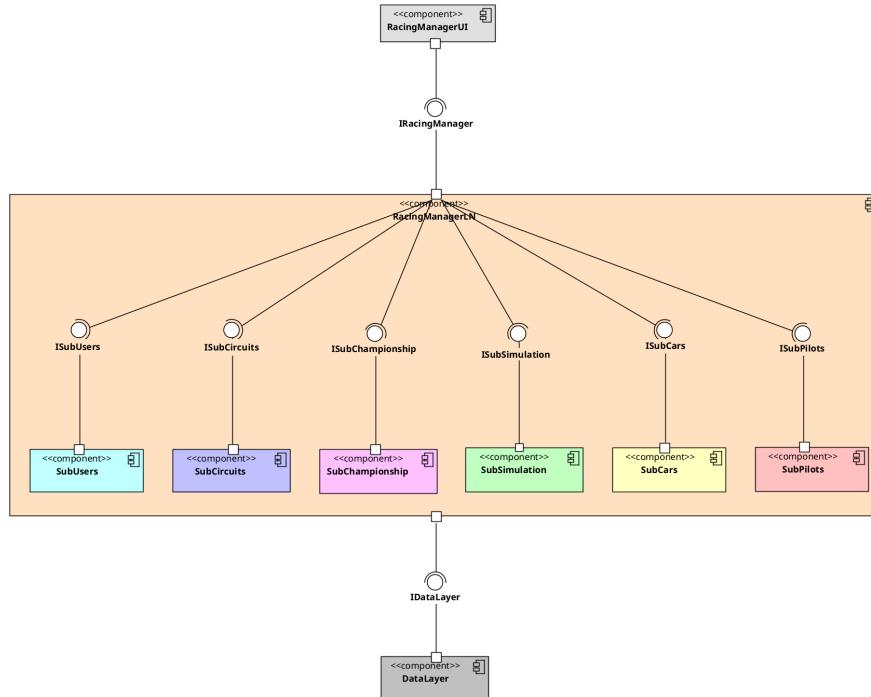


Figura 4: Diagrama de componentes.

## 5 Diagrama de *packages*

Tendo em conta a dimensão do sistema, elaboramos um diagrama de *packages* de modo a gerir o elevado número de classes e a melhor identificar cada uma e o seu papel dentro do sistema, indicando as dependências entre elas.

Na figura 5, verificamos que o *package* da interface do utilizador importa o conteúdo público do *package* “RacingManagerLN”, isto é, a classe correspondente à lógica de negócios. Este *package* contém também os *packages* dos subsistemas existentes, cujo conteúdo público é importado pela classe “RacingManagerLN”, e que não são visíveis à interface do utilizador (*User Interface*). Em relação aos subsistemas, verificamos que o *package* “Championships” acede aos elementos públicos do *package* “Circuits” e o *package* “Simulations” ao de todos os *packages* relativos a subsistemas, à exceção do *package* “Users”.

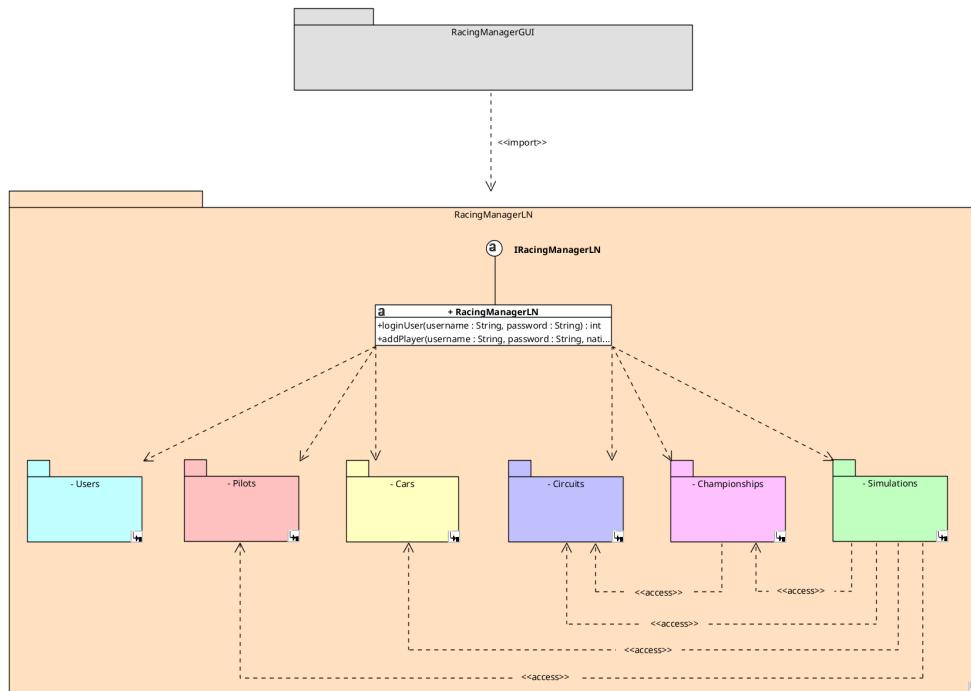


Figura 5: Diagrama de *packages*.

## 6 Diagrama de classes

Através do processo de análise referido na secção 3, verificamos que a interface que a camada da lógica de negócio da nossa aplicação deve exportar, dividida pelos vários subsistemas, resume-se ao seguinte:



Figura 6: Interface da lógica de negócio.

O subsistema das simulações é o subsistema que exporta grande parte das funções presentes na interface da lógica de negócio, visto que é ele que implementa as funcionalidades relacionadas com a simulação do jogo. O diagrama de classes respetivo é apresentado na figura 7:

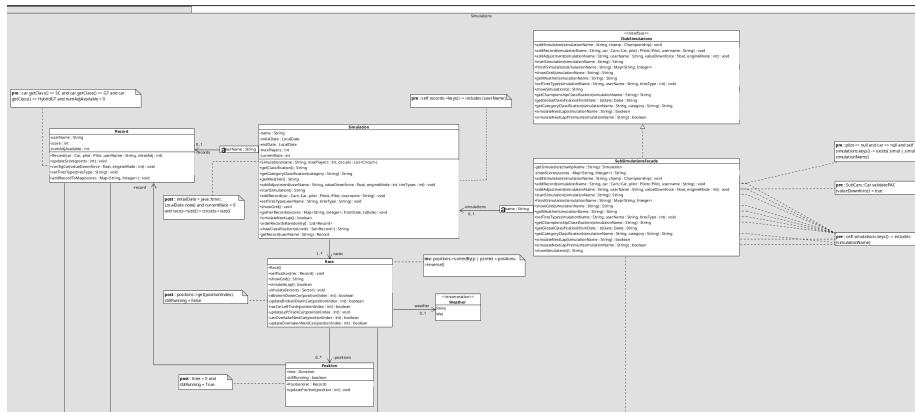


Figura 7: Diagrama de classes do subsistema das simulações.

Este diagrama foi feito de modo a responder de forma mais eficiente aos métodos que lidam com a simulação dos campeonatos.

De modo a dar início a uma simulação, a “SubSimulationsFacade” conhece a estrutura de um campeonato, pois é a partir deste que são extraídos alguns elementos importantes para começar a simulação - os circuitos do campeonato escolhido e o número máximo de participantes do mesmo. Assim sendo, é passada como argumento ao método “addSimulation” uma cópia do campeonato escolhido para não permitir alterações no objeto “Campeonato” inicialmente criado.

Cada simulação tem, assim, associada uma lista de corridas, cada uma realizada no respetivo circuito. Esta lista é fixa e é definida a partir dos circuitos escolhidos durante a criação do campeonato pelo administrador. Em cada corrida temos ainda uma lista de posições (objetos “Position”) de jogadores, ordenadas pela posição que esse jogador ocupa na corrida num determinado momento. Esta lista encontra-se inicialmente vazia, mas é preenchida durante a operação *startSimulation*, detalhada no diagrama de sequência respetivo na secção 8. Para além da lista das corridas de um dado campeonato, é ainda guardado o índice da corrida atual na lista respetiva, valor que será atualizado após a simulação de cada uma das corridas, e a lista dos “Records”, correspondentes aos registos feitos no campeonato.

Associado a cada posição encontra-se um apontador para o registo de cada jogador participante, inicialmente criado pelo método “addRecord” e guardado na variável de instância “records” na classe “Simulation”. Este registo é responsável por guardar os dados globais desse jogador no campeonato como pontuação acumulada, número de afinações disponíveis, carro e piloto escolhidos,

para além do nome do próprio utilizador. É importante realçar que tanto o piloto como o carro de cada registo do jogador são, na verdade, cópias dos pilotos e dos carros originais, ou seja, a coleção dos carros segue uma estratégia de composição de modo a obter apenas cópias dos carros existentes. Da mesma forma, o subsistema dos pilotos exporta apenas cópias de pilotos anteriormente criados, permitindo ao utilizador realizar alterações nos mesmos ao longo do campeonato sem interferir nos pilotos originais (caso isso passe a ser possível numa versão futura do jogo).

7 Código legado

De modo a validar o trabalho desenvolvido e a explorar novas funcionalidades, o nosso grupo gerou, de forma praticamente automática, o diagrama de classes referente ao código disponibilizado. Apesar de este ser apresentado no *Visual Paradigm* de uma forma incomum (não representando mapeamentos corretamente, por exemplo), este permitiu-nos ter uma ideia geral da estrutura da aplicação disponibilizada. Há aspectos que não iremos implementar no nosso trabalho, como a possibilidade de apostar numa corrida ou a possibilidade de ter equipas. Contudo, há funcionalidades em que as duas aplicações se intersetam e aí foi importante validar se o nosso trabalho está de acordo com o apresentado.

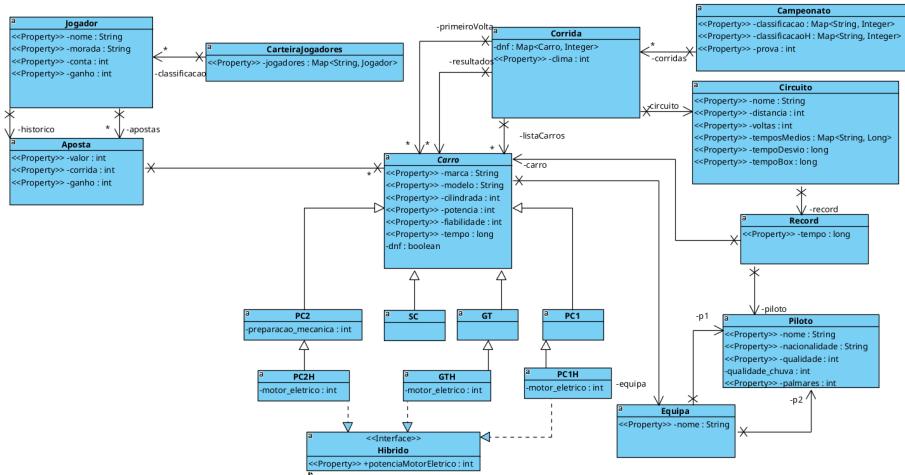


Figura 8: Diagrama de classes do código legado.

Um dos aspectos semelhantes nas duas aplicações é a hierarquia dos carros. Tanto no nosso trabalho como no trabalho disponibilizado, há a possibilidade de ter carros a combustão e carros híbridos. No nosso modelo, a classe “Car” é abstrata, pelo que carros não híbridos definem obrigatoriamente a sua potência.

Assim, a implementação da interface “Hybrid” torna-se útil pois permite assegurar que também os carros híbridos definem o método “getPower“ em função da potência elétrica e da potência de combustão.

Em relação à parte da simulação, foi importante analisar a classe *Record*, verificando que esta contabiliza o tempo que se passou desde o início da corrida. Também foi possível saber que esse tempo é incrementado num dado valor a cada volta, dependendo de muitos fatores, como a fiabilidade, a potência do carro e as características do piloto. Decidimos, por isso, acrescentar essa funcionalidade ao nosso trabalho, de modo a suportar o modo *Premium* de simulação.

Quanto ao resto do modelo, encontramos mais alguns pontos de contacto com a nossa abordagem, como a associação entre campeonato, corridas e circuito. Contudo, a estratégia utilizada por nós difere da que observamos no código disponibilizado. Por exemplo, não consideramos que a associação entre circuito e registo seja a melhor, uma vez que faz mais sentido que uma corrida contenha vários regtos, uma para cada jogador participante.

## 8 Diagramas de sequênciа

De modo a podermos verificar se o nosso diagrama de classes consegue dar resposta às operações pretendidas da forma mais eficiente possível, decidimos elaborar alguns diagramas de sequênciа para operações críticas da nossa aplicação:

### 8.1 Operação “getGlobalClassification“

Uma operação presente na interface da lógica de negócios da nossa aplicação é a operação “getGlobalClassification“ que recebe um intervalo de datas, devolvendo o *ranking* dos jogadores que participaram em campeonatos realizados nesse intervalo de datas. Numa primeira fase, consideramos que esta operação deveria ser realizada no subsistema dos utilizadores. No entanto, o facto de ter associada uma data de regsto, fez com que esta operação fosse realizada do lado das simulações, pois deve ser este sistema a guardar os regtos num dado campeonato, que é realizado num dado intervalo de datas.

Tal como podemos ver no diagrama de sequênciа representado na figura 9, é criado um *TreeMap* ao nível da fachada do subsistema das simulações que coleciona os regtos de participação dos utilizadores. A opção por um *TreeMap* deve-se ao facto de querermos que a estrutura esteja ordenada por pontuação. Criada a estrutura, resta-nos iterar por todos os campeonatos existentes no jogo, adicionando todos os regtos ao “map“ que é passado por referéncia. Um pormenor importante é o facto da operação de filtro ser realizada ao nível do regsto, o que contribui para uma boa distribuição do código pelos diversos componentes.

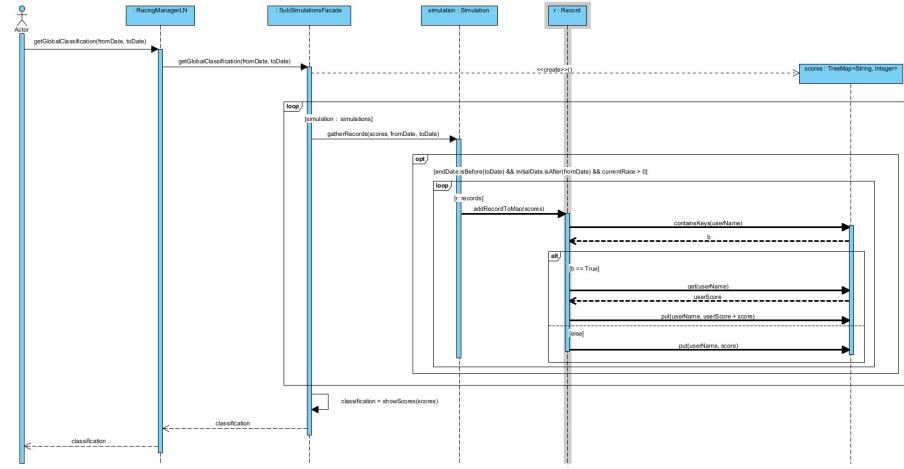


Figura 9: Diagrama de sequência da operação “getGlobalClassification” por datas.

## 8.2 Operação “startSimulation“

Outra operação fundamental na nossa aplicação é a de começar a simulação de uma nova corrida. Esta operação terá a responsabilidade de gerar e apresentar a grelha de partida da corrida. Seguindo o diagrama de sequência da figura 10, verificamos que, ao nível da simulação, o algoritmo começa por ordenar de forma aleatória os registos do campeonato, guardando-os numa variável “orderedRecords“. Depois, itera por todos os registos ordenados aleatoriamente, criando uma nova “Position“ tendo o registo por base. Essa “Position“ é, por fim, adicionada ao fim da lista das posições. Para terminar a operação, é invocado o método “showGrid“, que converte a lista das posições numa “String“ que depois é devolvida.

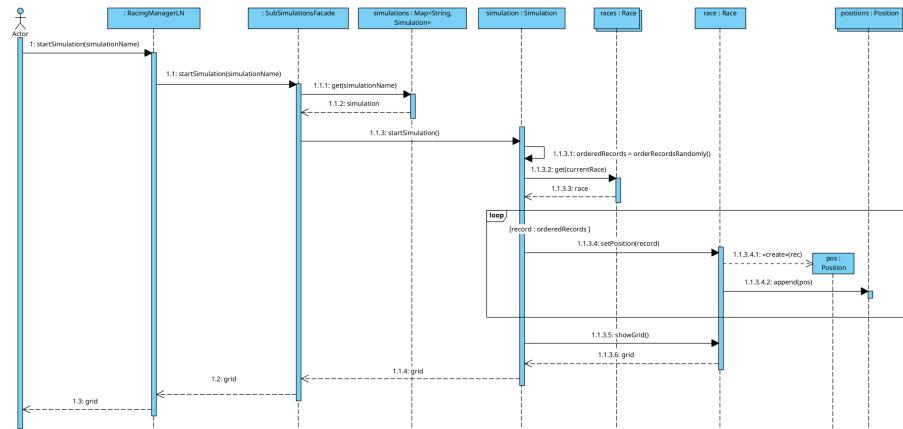


Figura 10: Diagrama de sequência da operação “startSimulation“.

Neste diagrama, a simulação é adquirida no subsistema das simulações, pois é este que contém a coleção de todos os campeonatos simulados ou em simulação, tal como é visível no diagrama de classe correspondente. Neste caso, assumimos que a simulação existe, pelo que não é necessário tratar essa situação de exceção. Podemos fazer esta assunção, indicando uma pré-condição no método `subChampionshipFacade :: startSimulation`:

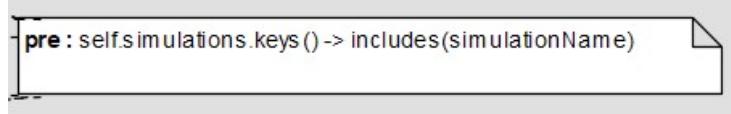


Figura 11: Pré-condição em OCL.

### 8.3 Operação “simulateNextLap“

Uma operação fundamental a realizar durante uma corrida é a de simular a próxima volta. Esta operação vai determinar a posição dos pilotos nessa volta. Através do diagrama de sequência da figura 12, podemos constatar que, ao nível do “SubSimulationsFacade“, começamos por obter a simulação em questão e, seguidamente, ao nível da simulação, obtemos a corrida que está a ocorrer no momento. De seguida, ao nível da corrida, fazemos a simulação da volta seguinte, onde para cada carro verificamos se o mesmo avariou (“isBrokenDownCar“) e, em caso afirmativo, o seu estado é atualizado na corrida (“updateBrokenDownCar“). Em seguida, faz-se uma iteração na lista de setores da corrida e simula-se cada setor. De forma a tornar o diagrama mais legível e menos complexo, a simulação de cada setor é feita num diagrama de sequência à parte que é passado como referência - “simulateSector“.

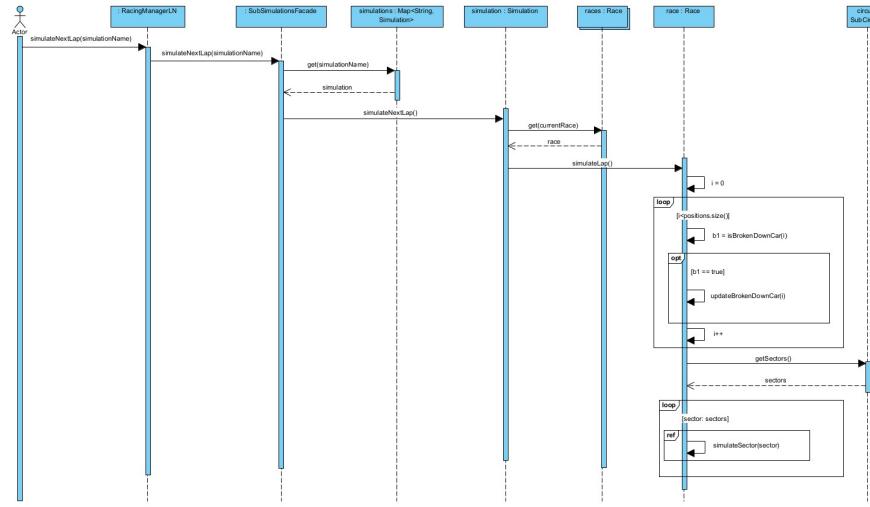


Figura 12: Diagrama de sequência da operação “simulateNextLap“.

Para fazer a simulação num dado setor, ao nível da corrida, testamos para todos os carros menos o primeiro (último da lista de posições), se o carro saiu da pista a tentar fazer uma ultrapassagem nesse setor e, se esse for o caso, é atualizada a sua posição na pista. Se a ultrapassagem for efetuada com sucesso, a posição é atualizada para a posição seguinte à que se encontra, enquanto que o carro que foi ultrapassado perde uma posição. É importante referir que o algoritmo não permite, para já, que um carro faça duas ultrapassagens no mesmo setor. Isto justifica a incrementação do contador *i* no caso de haver ultrapassagem, dado que queremos agora validar se a ultrapassagem acontece nas próximas duas posições e não entre o carro que ultrapassou e o que vai à sua frente.

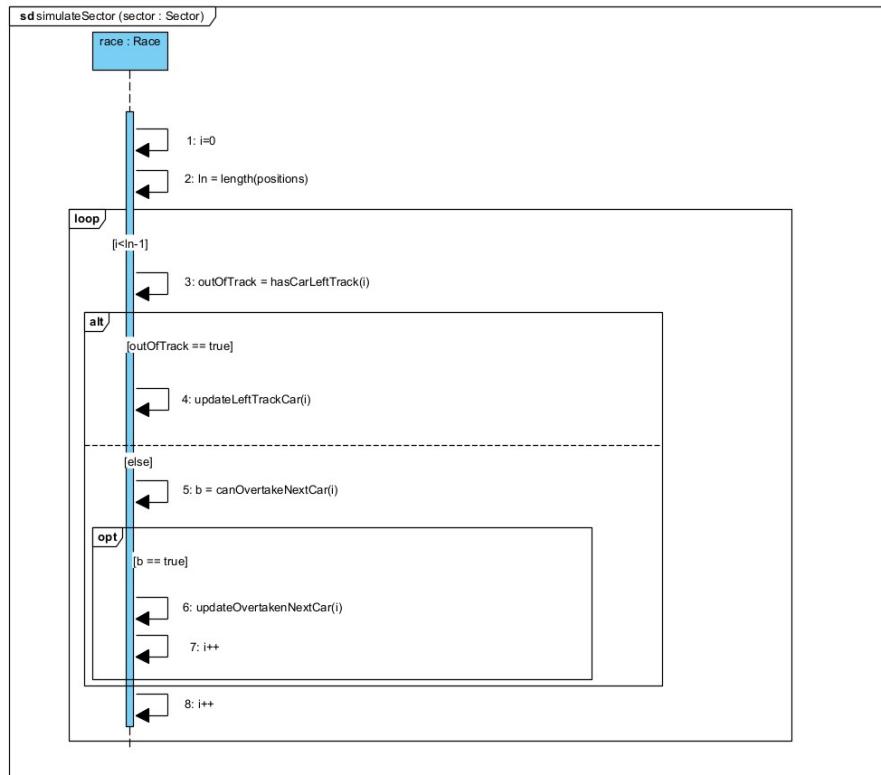


Figura 13: Diagrama de sequência da operação “simulateSector“.

Todos os diagramas de sequência serviram para validar, fundamentalmente, o diagrama de classes do subsistema das simulações. Os restantes diagramas não são apresentados com tanto detalhe, mas encontram-se presentes em anexo.

## 9 Diagrama de atividades

O processo de jogar um campeonato é um processo bastante complexo de definir apenas a partir das especificações de *use cases*. De modo a poder analisar as diferentes atividades que ocorrem no processo de jogar, elaboramos o seguinte diagrama de atividades:

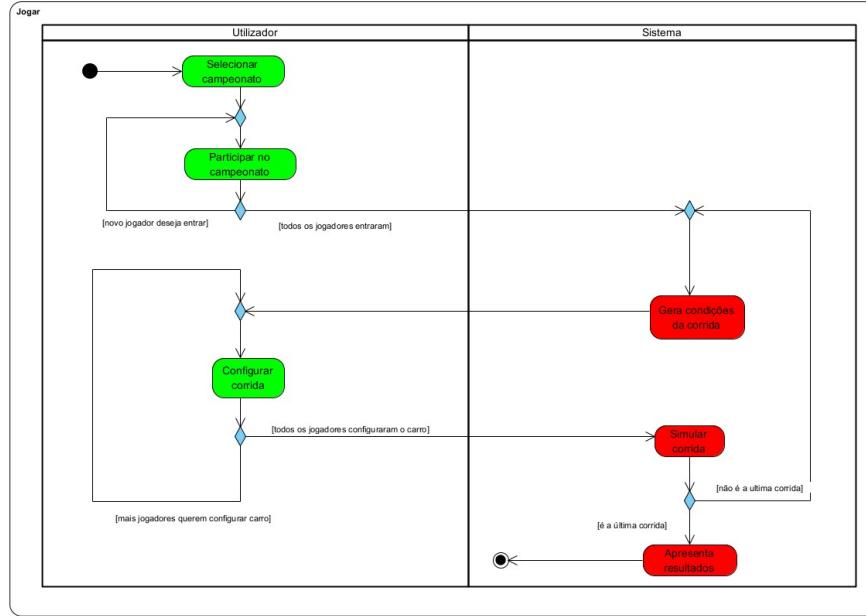


Figura 14: Diagrama de atividades relativo ao processo de jogar.

De uma forma geral, um dado jogador começa por escolher um campeonato para jogar, onde se vão juntando mais jogadores que escolhem o piloto e o carro com o qual querem participar no campeonato. Em seguida, o sistema gera as condições meteorológicas da primeira corrida, permitindo aos utilizadores configurar os carros, se tal for possível. Quando todos tiverem configurado o seu carro, começa a simulação da corrida. Enquanto houver mais corridas para simular, o processo de configuração e simulação é repetido.

As ações representadas poderiam ser consideradas outra atividade, visto que cada uma delas é constituída por sequências de ações mais simples. No entanto, o diagrama foi realizado de modo a que cada ação (minimamente complexa) corresponesse a um *use case* bem definido, pelo que os fluxos de ações já estão devidamente representados.

Outros processos importantes poderiam ser representados através de diagramas de atividades. Um bom exemplo disso seriam os algoritmos de simulação, que mais tarde iremos implementar, quer para a simulação normal quer para a simulação *Premium*. Estes algoritmos terão também um lado probabilístico, pelo que o seu planeamento atempado será essencial. No entanto, nesta fase, procuramos não entrar em grande detalhe nos algoritmos utilizados, preferindo uma especificação a mais alto nível do nosso sistema.

## 10 Conclusão

Para concluir, nesta segunda fase do trabalho procedemos ao desenvolvimento da especificação da nossa aplicação, *RacingManager*, cujos requisitos tinham sido analisados na primeira fase, focando na camada da lógica de negócio.

Ao longo deste processo, discutimos as diferentes formas de cumprir com os requisitos estabelecidos, procurando maximizar a eficiência das operações sem esquecer a modularidade do sistema.

Consideramos que tanto a primeira fase como a segunda, foram um processo bastante enriquecedor para todos os elementos do grupo, na medida em que ficamos a conhecer as diferentes fases da especificação do sistema e a contribuição de cada uma para o desenvolvimento de um projeto, tendo em vista um resultado final. Através da linguagem UML, elaboramos diagramas que representam o comportamento que a aplicação deve ter nas diversas situações.

Neste momento, consideramos estar, por isso, preparados para implementar, numa última fase do projeto, o sistema que nos foi proposto desenvolver.

# 11 Anexos

## 11.1 Diagramas de classes

### 11.1.1 RacingManagerLN

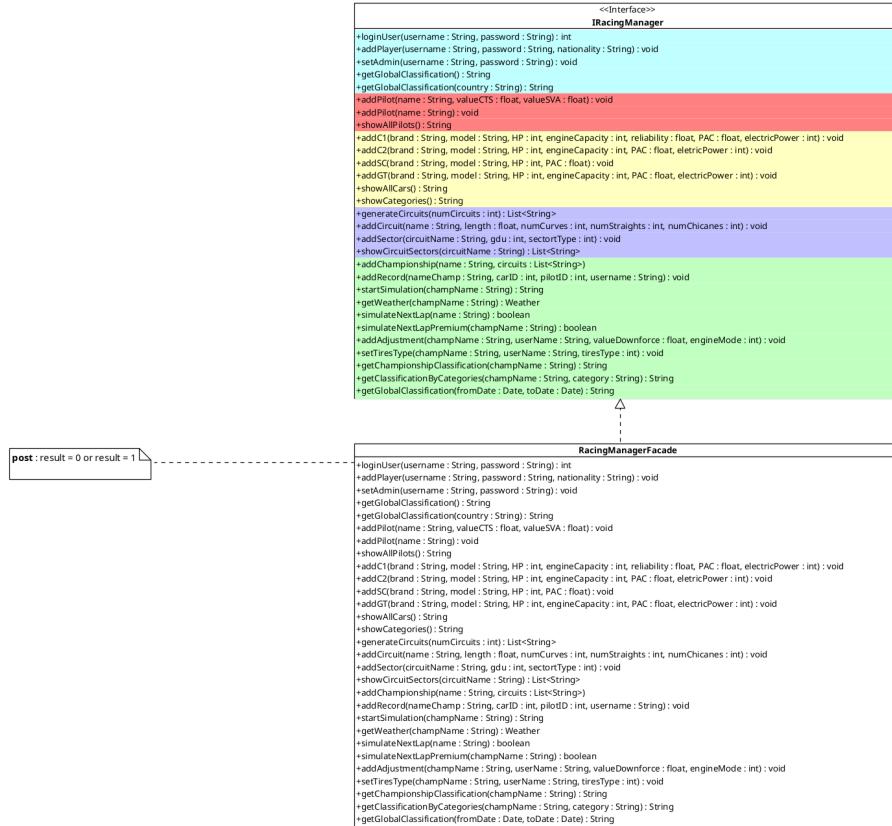


Figura 15: Diagrama de classes relativo à fachada da lógica de negócios.

### 11.1.2 SubUsers

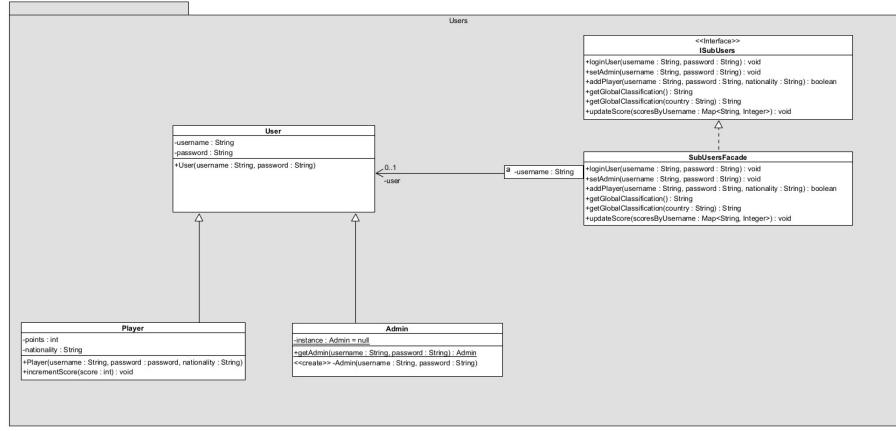


Figura 16: Diagrama de classes relativo ao subsistema dos utilizadores.

**Nota:** a classe “Admin“ é um *singleton* porque só admite uma única instância.

### 11.1.3 SubCircuits

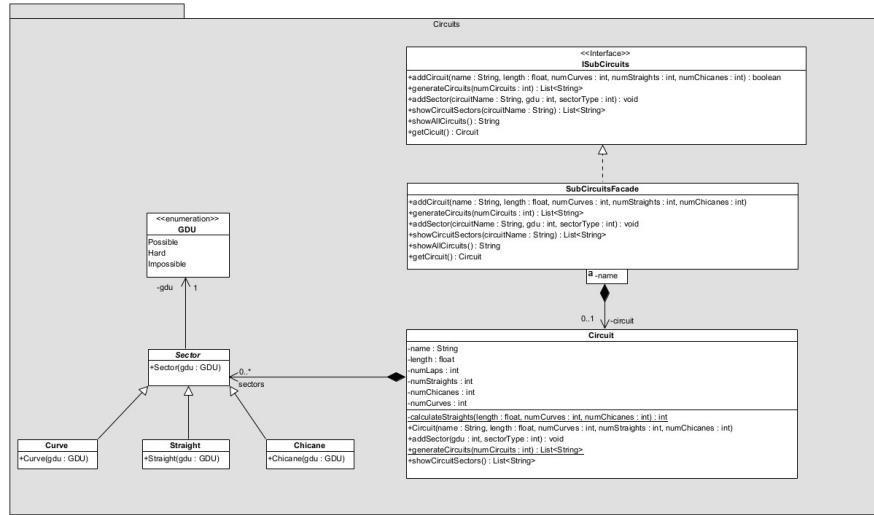


Figura 17: Diagrama de classes relativo ao subsistema dos circuitos.

#### 11.1.4 SubCars

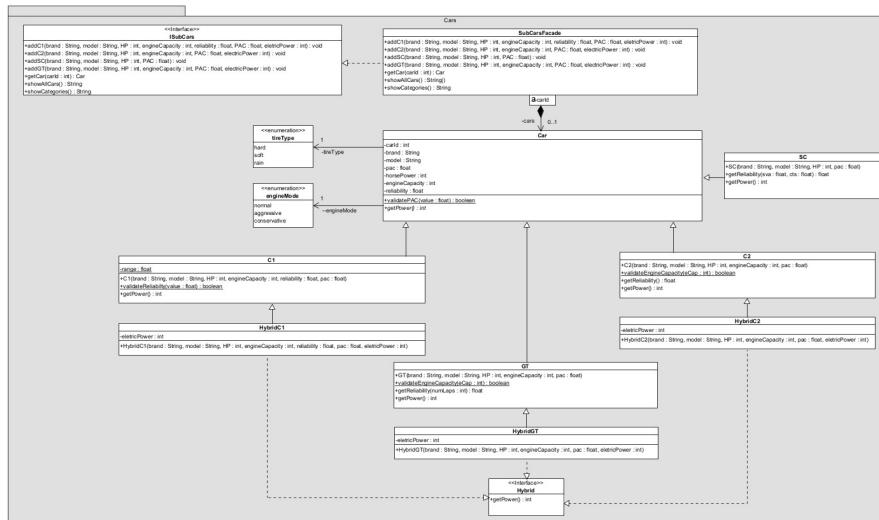


Figura 18: Diagrama de classes relativo ao subsistema dos carros.

### 11.1.5 SubPilots

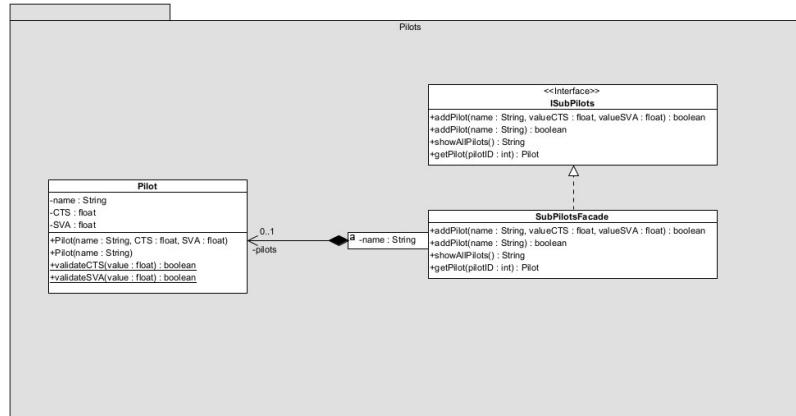


Figura 19: Diagrama de classes relativo ao subsistema dos pilotos.

### 11.1.6 SubChampionships

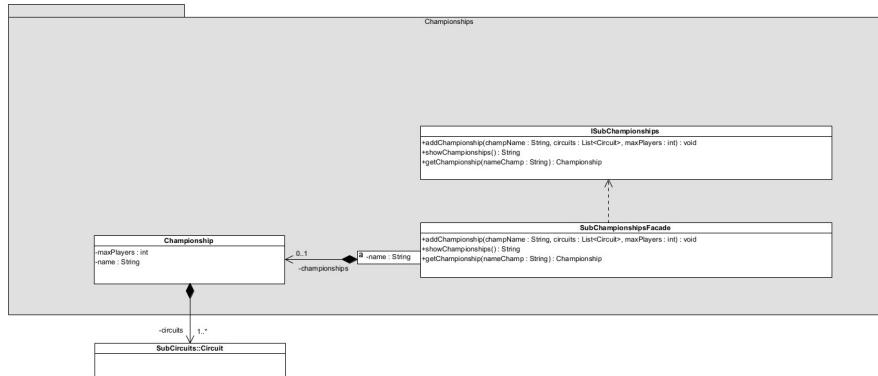


Figura 20: Diagrama de classes relativo ao subsistema dos campeonatos.

### 11.1.7 SubSimulations

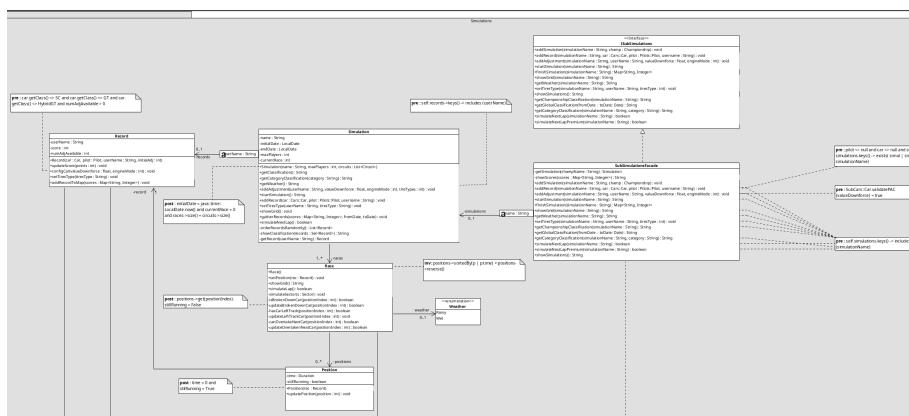


Figura 21: Diagrama de classes relativo ao subsistema das simulações.

## 11.2 Diagrama de componentes

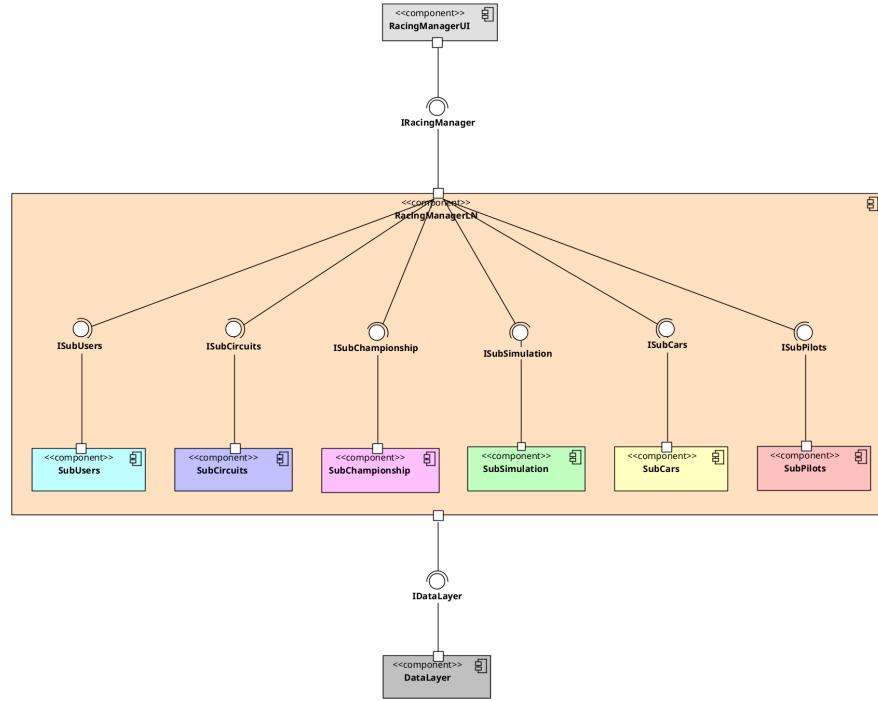


Figura 22: Diagrama de componentes.

### 11.3 Diagrama de *packages*

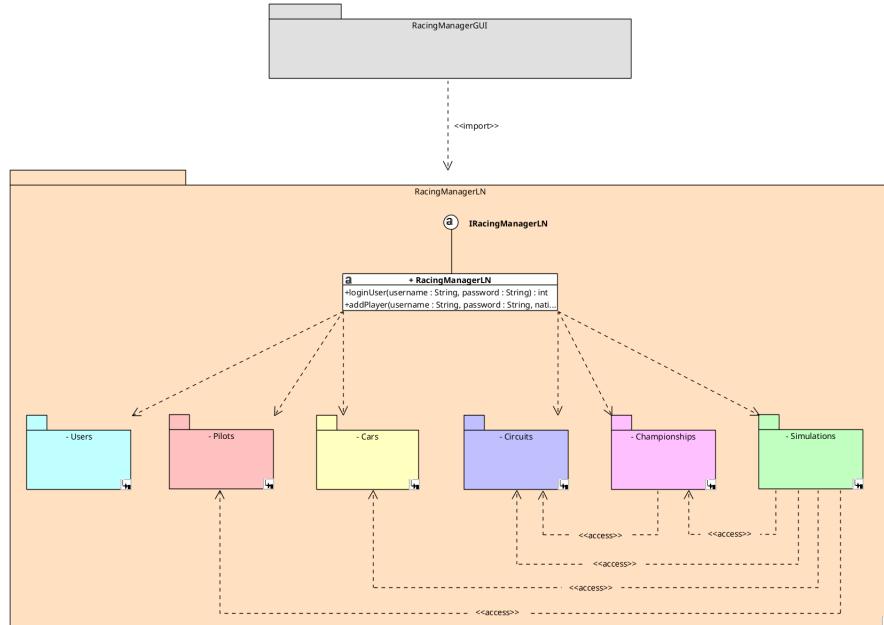


Figura 23: Diagrama de *packages*.

### 11.4 Diagramas de sequênciа

#### 11.4.1 “addSimulation“

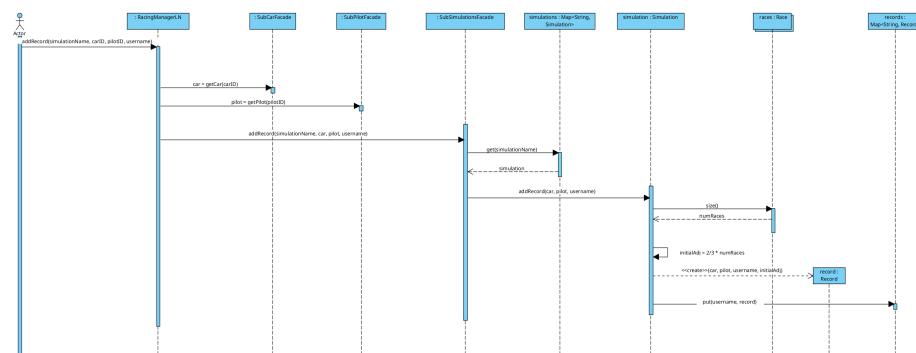


Figura 24: Diagrama de sequênciа da operação “addSimulation“.

#### 11.4.2 “addRecord“

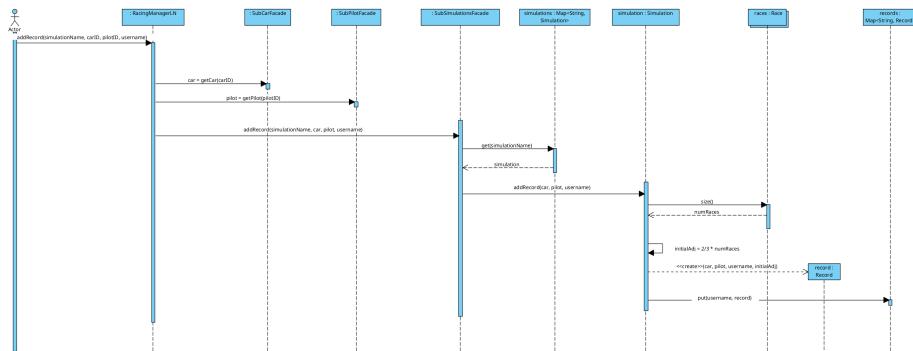


Figura 25: Diagrama de sequência da operação “addRecord”.

### 11.4.3 “startSimulation“

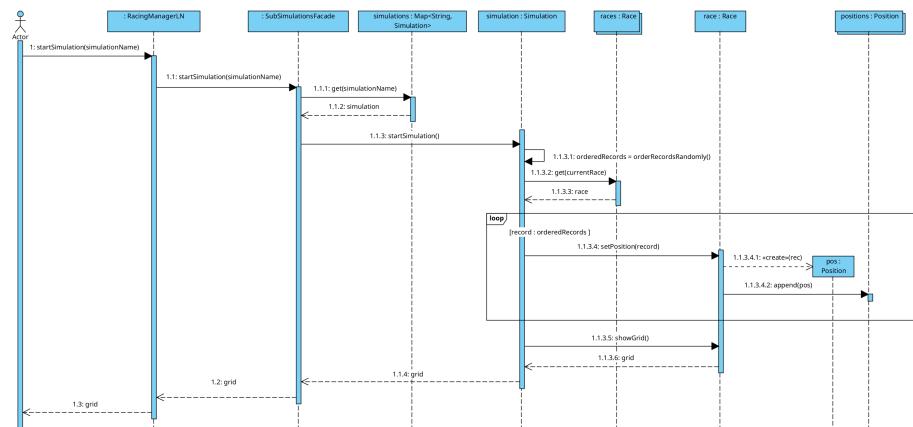


Figura 26: Diagrama de sequência da operação “startSimulation“.

#### 11.4.4 “getWeather“

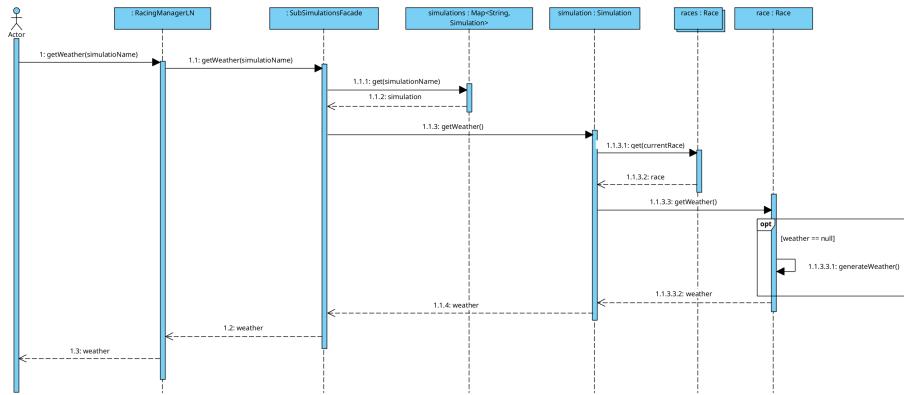


Figura 27: Diagrama de sequência da operação “getWeather“.

#### 11.4.5 “addAdjustment“

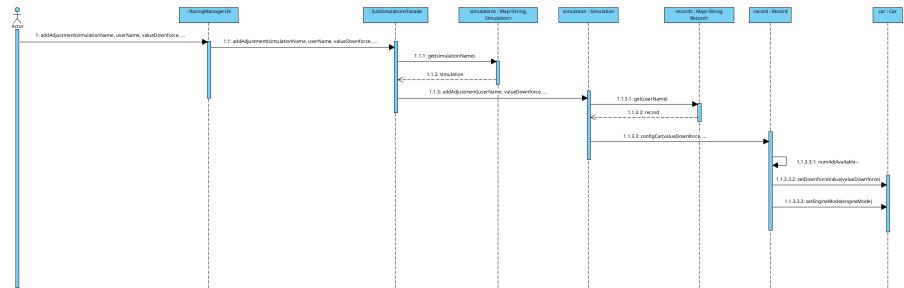


Figura 28: Diagrama de sequência da operação “addAdjustment“.

#### 11.4.6 “simulateNextLap“

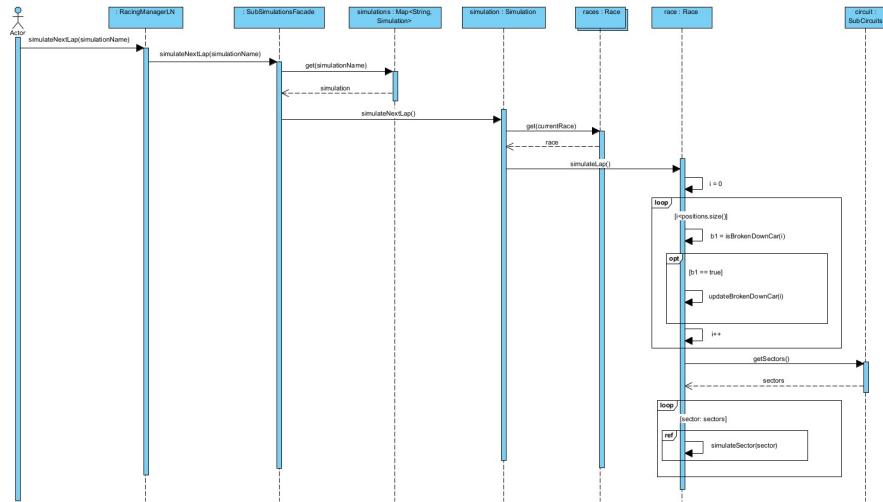


Figura 29: Diagrama de sequência da operação “simulateNextLap“.

#### 11.4.7 “simulateSector“

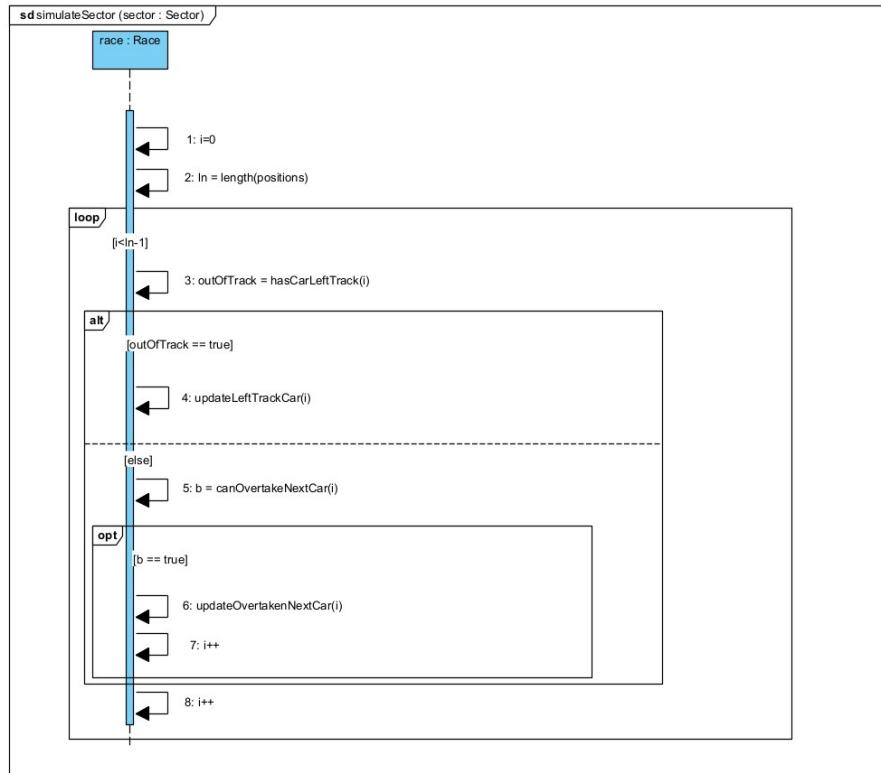


Figura 30: Diagrama de sequênciā da operação “simulateSector“.

### 11.4.8 “showGrid“

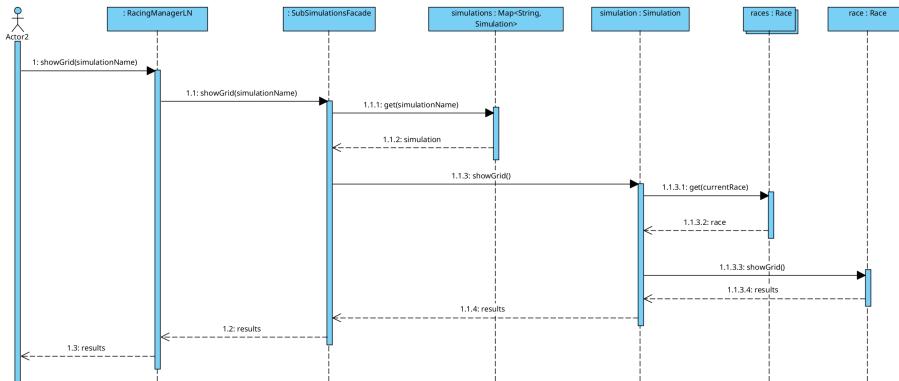


Figura 31: Diagrama de sequência da operação “showGrid“.

### 11.4.9 “finishSimulation“

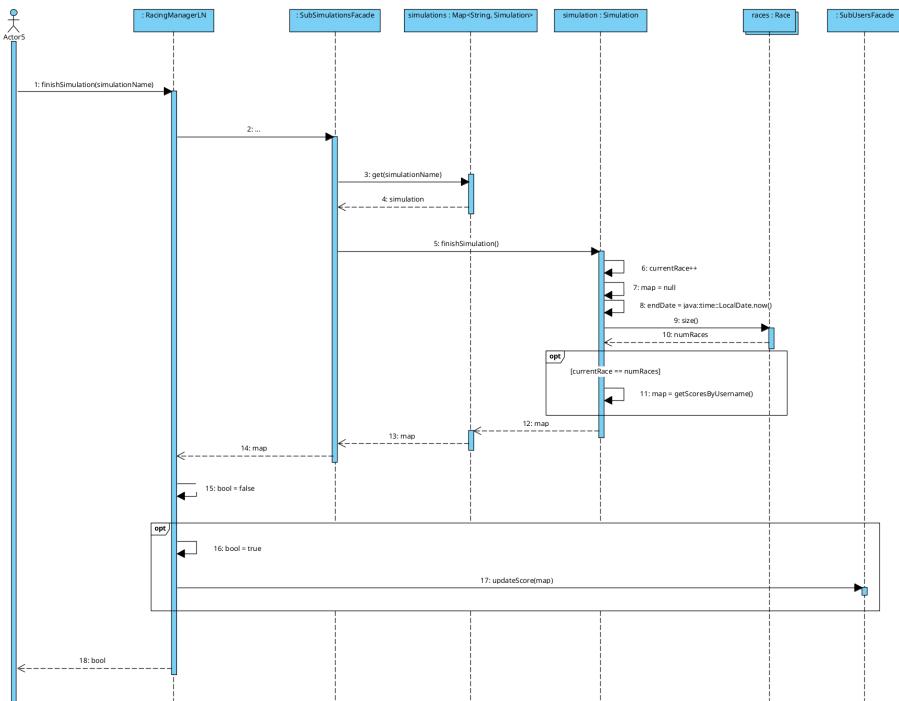


Figura 32: Diagrama de sequência da operação “finishSimulation“.

#### 11.4.10 “getCategoryClassification“

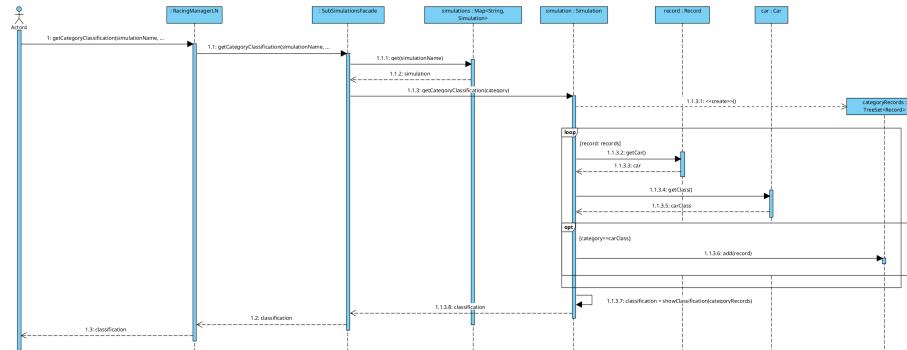


Figura 33: Diagrama de sequência da operação “getCategoryClassification“.

#### 11.4.11 “getGlobalClassification“

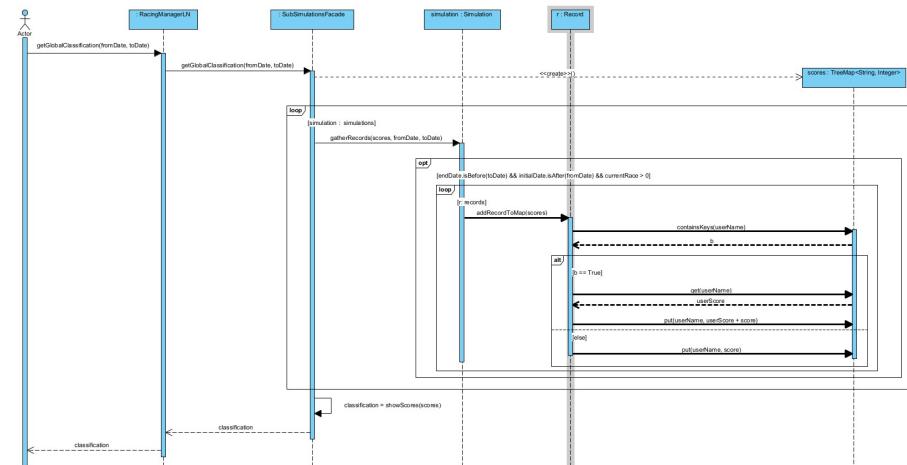


Figura 34: Diagrama de sequência da operação “getGlobalClassification“.

## 11.5 Diagrama de atividades

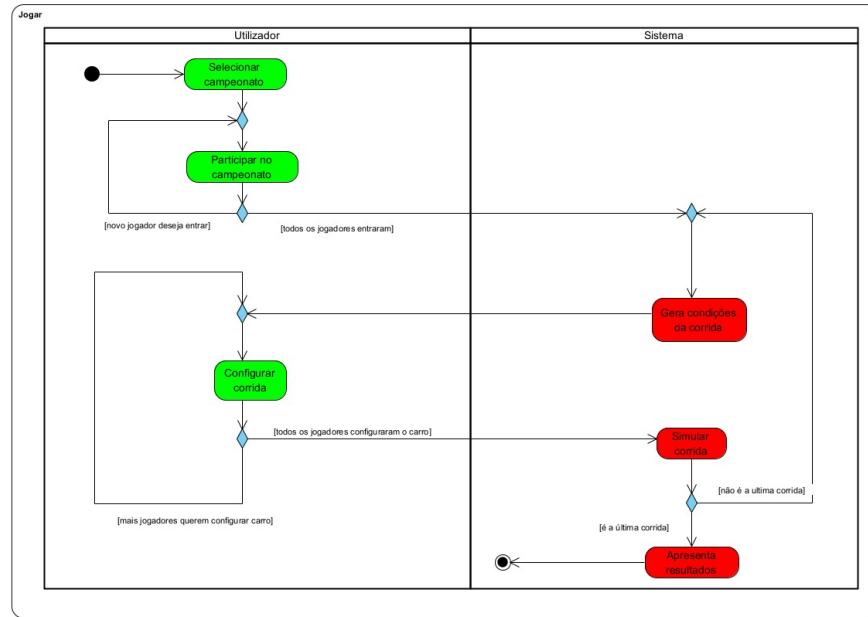


Figura 35: Diagrama de atividades do processo jogar.