



UNIVERSIDADE DO MINHO  
Departamento de Informática

DESENVOLVIMENTO DE SISTEMAS DE SOFTWARE

## Simulador de Corridas: 3<sup>a</sup> Fase

Grupo 7

### Feito por:

Dinis Gonçalves Estrada (A97503)

Eduardo Miguel Pacheco Silva (A95345)

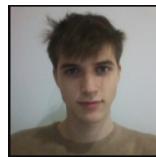
Emanuel Lopes Monteiro da Silva (A95114)

João Gonçalo de Faria Melo (A95085)

João Miguel da Silva Pinto Ferreira (A89497)



(a) Dinis Estrada



(b) Eduardo Silva



(c) Emanuel Silva



(d) João Melo



(e) João Ferreira

GitHub  
January 7, 2023  
Ano Letivo 2022/23

# Conteúdo

<b>1</b>	<b>Introdução</b>	<b>3</b>
<b>2</b>	<b>Objetivo do trabalho</b>	<b>3</b>
<b>3</b>	<b>Alterações realizadas</b>	<b>4</b>
3.1	Modelo de Domínio . . . . .	4
3.2	Modelo de Use Cases . . . . .	5
<b>4</b>	<b>Use Cases</b>	<b>6</b>
4.1	RegistarComoAdministrador . . . . .	6
4.2	RegistarComoJogador . . . . .	6
4.3	EfetuarAutenticacao . . . . .	7
4.4	AdicionarCampeonato . . . . .	8
4.5	AdicionarCircuito . . . . .	9
4.6	AdicionarCarro-C1 . . . . .	10
4.7	AdicionarCarro-C2 . . . . .	11
4.8	AdicionarCarro-GT . . . . .	12
4.9	AdicionarCarro-SC . . . . .	13
4.10	AdicionarPiloto . . . . .	14
4.11	Registar Jogador no Campeonato . . . . .	15
4.12	Preparar Corrida . . . . .	16
4.13	Classificação Corrida . . . . .	17
4.14	Classificação Campeonato . . . . .	17
<b>5</b>	<b>Diagrama de Componentes</b>	<b>18</b>
<b>6</b>	<b>Diagrama de Classes</b>	<b>19</b>
6.1	O que foi aproveitado do código legado? . . . . .	19
6.1.1	Classe Campeonato . . . . .	19
6.1.2	Classe Corrida . . . . .	19
6.1.3	Classe Circuito . . . . .	19
6.1.4	Classe Carros . . . . .	19
6.1.5	Classe Piloto . . . . .	19
6.2	SubSistema Utilizadores . . . . .	20
6.3	SubSistema Campeonatos . . . . .	20
6.4	SubSistema Circuitos . . . . .	21
6.5	SubSistema Carros . . . . .	21
<b>7</b>	<b>SubSistema Pilotos</b>	<b>22</b>
<b>8</b>	<b>Diagramas de Sequência</b>	<b>23</b>
8.1	validaAfinacoes() . . . . .	23
8.2	alteraAfinacao() . . . . .	24
8.3	getResultsdos() . . . . .	24
8.4	getResultsdosCorrida() . . . . .	25
8.5	prepararCorrida() . . . . .	25
8.6	registarJogador() . . . . .	26
<b>9</b>	<b>Implementação</b>	<b>26</b>
9.1	Arquitetura . . . . .	26
9.2	Base de Dados . . . . .	26
9.2.1	Modelo lógico . . . . .	26
9.2.2	Povoamento . . . . .	26

9.2.3	ORM e DAOs . . . . .	27
<b>10</b>	<b>Análise Crítica</b>	<b>28</b>
<b>11</b>	<b>Conclusão</b>	<b>28</b>

## **1 Introdução**

Esta segunda fase do projeto da cadeira de Desenvolvimento de Sistemas de Software, consistiu no desenvolvimento de um modelo conceptual que seja capaz de suportar os Use Cases definidos na fase anterior. Visto que todo o processo é incremental e iterativo, esta fase consistiu numa continuação da primeira, pelo que foi necessária uma reavaliação do modelo de domínio e a alteração de alguns Use Cases. Esta análise dos detalhes é de extrema importância, uma vez que garante a coerência global do procedimento e facilita a resolução de fases futuras.

## **2 Objetivo do trabalho**

Ao longo desta segunda fase foram-nos apresentados vários desafios. Num momento inicial, foi necessário, tendo em conta os Use Cases definidos previamente e o modelo de Domínio, o levantamento das responsabilidades do sistema e a definição dos respetivos métodos (API da Lógica de Negócio), sendo necessária a criação de vários subsistemas pelos quais estes métodos seriam distribuídos. De seguida, passamos para a criação de um possível diagrama de classes, definindo atributos e classes bem como as relações entre essas mesmas classes. O facto deste diagrama ser navegável, facilitou o processo de desenvolvimento de diagramas de sequência, que por sua vez representam as trocas de mensagens e a “interação” entre os vários objetos. Foi definido um diagrama de sequência para cada método levantado. Por último, criámos o diagrama de componentes no qual foram representados os vários subsistemas que o grupo considerou apropriados. Todo este processo referido, serviu para desenvolver um modelo conceptual coerente com a primeira fase desenvolvida. Todos estes diagramas foram desenvolvidos usando a ferramenta Visual Paradigm.

### 3 Alterações realizadas

### 3.1 Modelo de Domínio

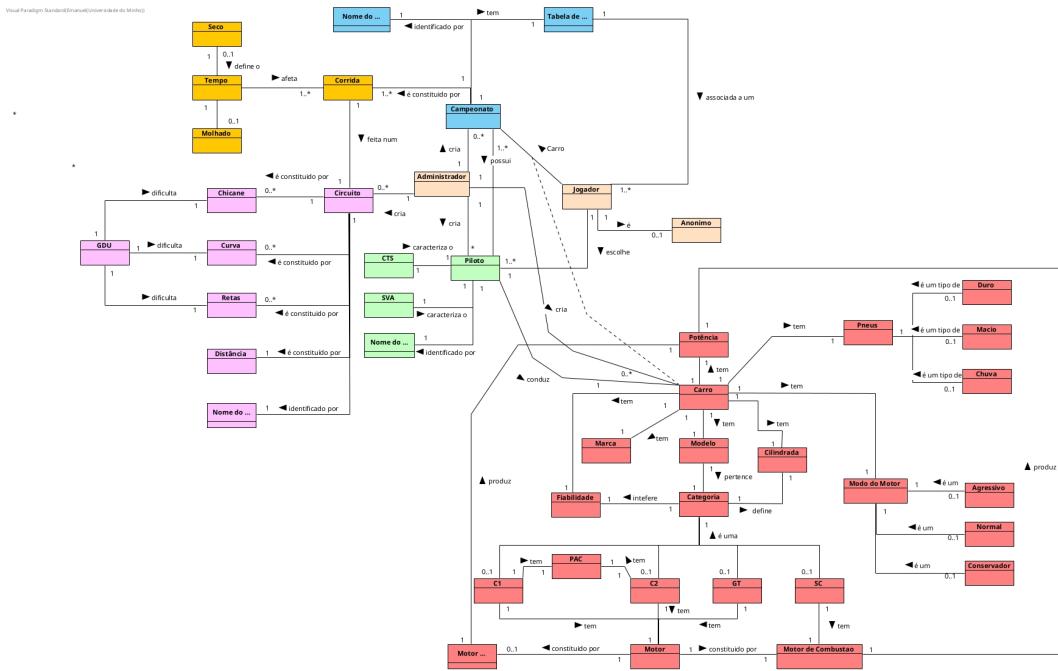


Figure 2: Modelo de Domínio

Em relação ao modelo de Domínio realizado na 1<sup>a</sup> fase, apenas foi adicionado uma ligação entre a classe Piloto e a classe Campeonato pois um Campeonato também possui pilotos.

### 3.2 Modelo de Use Cases

Nesta 2<sup>a</sup> fase alteramos o modelo de use cases da seguinte forma: foi removido o configurar corrida e configurar campeonato e foi adicionado o use case preparar Corrida, consultar classificação da corrida, consultar classificação do campeonato e registar o jogador no campeonato como se pode observar na imagem abaixo.

O Modelo de Use Cases permite identificar e descrever os requisitos funcionais do nosso sistema.

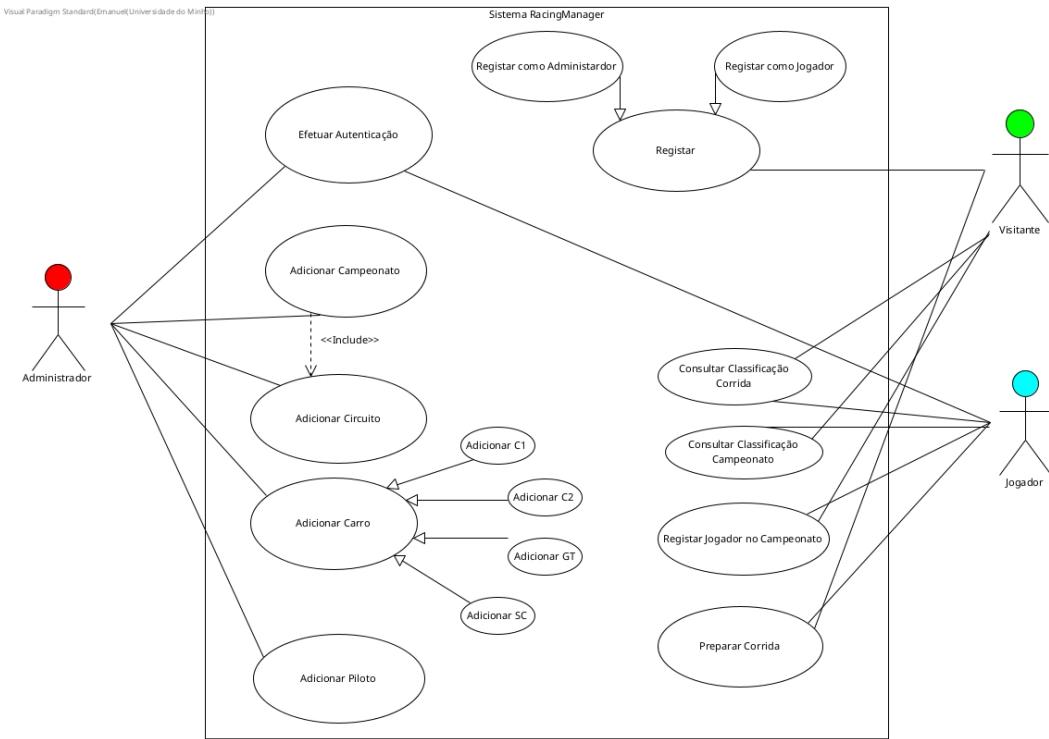


Figure 3: Modelo de Use Cases

## 4 Use Cases

### 4.1 RegistarComoAdministrador

Use Case:	Registar como Administrador			
Descrição:	Utilizador regista-se na aplicação como administrador.			
Cenário:	O José abre a aplicação e indica que pretende criar uma conta como administrador. O José indica as credenciais que pretende utilizar e regista-se como administrador.			
Pré-Condição:	True			
Pós-Condição:	Utilizador regista-se com sucesso.			
		1. Identificar responsabilidades da LN	2. definir API (identificar métodos)	3. identificar subsistemas (agrupar métodos)
Fluxo Normal				
	1. Utilizador indica que pretende criar uma conta como administrador.			
	2. Utilizador insere as credenciais para a sua nova conta.			
	3. Sistema cria e regista a nova conta.	criar e registar a nova conta	registarAdministrador( nomeAdmin:String, passAdmin:String):	Utilizadores

Table 1: Use Case "Registar como Administrador"

### 4.2 RegistarComoJogador

Use Case:	Registar como Jogador			
Descrição:	Utilizador regista-se na aplicação como jogador.			
Cenário:	O Jorge abre a aplicação e indica que pretende criar uma conta como jogador. O Jorge indica as credenciais que pretende utilizar e regista-se como jogador.			
Pré-Condição:	True			
Pós-Condição:	Utilizador regista-se com sucesso.			
		1. Identificar responsabilidades da LN	2. definir API (identificar métodos)	3. identificar subsistemas (agrupar métodos)
Fluxo Normal				
	1. Utilizador indica que pretende criar uma conta como jogador.			
	2. Utilizador insere as credenciais para a sua nova conta.			
	3. Sistema cria e regista a nova conta.	criar e registar a nova conta	registarJogador(nomeJog:String, passJog:String):	Utilizadores

Table 2: Use Case "Registar como Jogador"

### 4.3 Efetuar Autenticacao

Use Case:	Efetuar Autenticação			
Descrição:	Utilizador autentica-se na aplicação.			
Cenário:				
Pré-Condição:	O utilizador está registado como administrador ou jogador.			
Pós-Condição:	O utilizador autentica-se com sucesso, ficando disponíveis as funcionalidades da aplicação.			
		1. Identificar responsabilidades da LN	2. definir API (identificar métodos)	3. identificar subsistemas (agrupar métodos)
Fluxo Normal				
	1.	Utilizador apresenta as suas credenciais.		
	2.	Sistema valida as credenciais.	verificar que o utilizador está registado	autenticaUtilizador(nome:String, pass:String):boolean
	3.	Sistema permite o acesso, apresentando as funcionalidades disponíveis para o jogador.		Utilizadores
Fluxo Excepção(1)		[credenciais incorretas](passo 2)		
	2.1.	Sistema verifica que as credenciais não correspondem a uma conta existente.	verificar que o utilizador não está registado	autenticaUtilizador(nome:String, pass:String):boolean
	2.2.	Sistema nega acesso ao utilizador.		Utilizadores

Table 3

#### 4.4 AdicionarCampeonato

Use Case:	Adicionar Campeonato			
Descrição:	Administrador adiciona um campeonato à lista de campeonatos.			
Cenário:	O administrador decide adicionar um campeonato. Dá-lhe um nome e escolhe os circuitos.			
Pré-Condição:	Utilizador autenticou-se como administrador.			
Pós-Condição:	Campeonato é adicionado à lista de campeonatos e fica imediatamente disponível para ser jogar.			
		1. Identificar responsabilidades da LN	2. definir API (identificar métodos)	3. identificar subsistemas (agrupar métodos)
Fluxo Normal				
	1.	Administrador indica que pretende criar um campeonato.		
	2.	Administrador atribui um nome ao campeonato.		
	3.	Escolhe os circuitos a adicionar ao campeonato.		
	4.	O campeonato é adicionado, ficando disponível para ser jogado.	adicionar novo campeonato à lista de campeonatos	adicionarCampeonato(campeonato : Campeonato)
Fluxo Alternativo(1)	[administrador escolhe criar circuito] (passo 3)			Campeonatos
	3.1	Escolhe criar circuito.		
	3.2	<<include>> Adicionar Circuito		
	3.3	Retorna a (passo 3)		

Table 4: Use Case ”Adicionar Campeonato”

## 4.5 AdicionarCircuito

Use Case:	Adicionar Circuito			
Descrição:	O jogador cria um circuito.			
Cenário:	O administrador decide adicionar um circuito novo. Dá-lhe um nome, indica a distância e o número de curvas, de retas e de chicanes, atribuindo um GDU a cada uma.			
Pré-Condição:	Utilizador autenticou-se como administrador.			
Pós-Condição:	Círculo é adicionado à lista de círculos e fica imediatamente disponível para ser jogado.			
Fluxo Normal		1. Identificar responsabilidades da LN	2. definir API (identificar métodos)	3. identificar subsistemas (agrupar métodos)
	1.	Administrador indica que pretende criar um círculo.		
	2.	Sistema pede o nome do círculo, a distância, o número de curvas e o número de chicanes.		
	3.	Administrador atribui um nome, uma distância, um número de curvas e um número de chicanes ao círculo.		
	4.	Sistema calcula o número de retas.	calcular o número de retas do círculo calcularRetas(distancia:int, nCurvas:int, nChicanes:int):int	Círculos
	5.	Sistema pede o grau de dificuldade de ultrapassagem (GDU) das curvas, chicanes e retas.		
	6.	Administrador atribui um GDU a cada curva e reta do círculo.		
	7.	Sistema verifica que os GDU's atribuídos são válidos.	validar os GDU's validarGDU(gdu: ENUM(GDU)):boolean	Círculos
	8.	Sistema pede o número de voltas que o círculo terá.		
	9.	Administrador atribui um número de voltas.		
	10.	O círculo é adicionado, ficando disponível para ser jogado.	adicionar novo círculo à lista de círculos adicionarCírculo(círculo: Círculo)	Círculos

Table 5: Use Case "Adicionar Circuito"

## 4.6 AdicionarCarro-C1

Use Case:	Adicionar Carro - C1			
Descrição:	Administrador adiciona um carro da categoria C1 à lista de carros. Indica marca, modelo, potência e cilindrada. Para finalizar indica o PAC.			
Cenário:	Um administrador opta por adicionar um novo carro da categoria C1.			
Pré-Condição:	Utilizador autenticou-se como administrador.			
Pós-Condição:	Carro é adicionado à lista de carros e fica imediatamente disponível para ser utilizado.			
		1. Identificar responsabilidades da LN	2. definir API (identificar métodos)	3. identificar subsistemas (agrupar métodos)
Fluxo Normal				
	1.	Sistema apresenta características necessárias para C1 (Marca, Modelo,Cilindrada, Potência).		
	2.	Administrador indica marca, modelo, cilindrada e potência.		
	3.	Sistema valida características.	valida os valores atribuídos ao carro	validaCarro(marca:String, modelo:String, cilindrada:int, potencia:int):boolean
	4.	Sistema ajusta fiabilidade para um valor de aproximadamente 95%.	calcular a fiabilidade para o carro	calculaFiabilidade():float
	5.	Sistema pede PAC e indica que o $0 < \text{PAC} \leq 1$ .		
	6.	Administrador indica PAC.		
	7.	Administrador indica que carro não é híbrido.		
	8.	Sistema regista carro e este fica disponível para jogar.	adicionar carro à lista de carros	adicionarCarro(carro: Carro)
Fluxo Alternativo(1)		[administrador indica que carro é híbrido] (passo 8)		Carros
	7.1	Sistema pede potência de motor elétrico.		
	7.2	Administrador indica valor de potência do motor elétrico.		Carros
	7.3	Sistema ajusta fiabilidade para um novo valor (<95%).	calcular a fiabilidade para o carro	calculaFiabilidade():float
	7.4	(retorna a passo 8)		
Fluxo Exceção(2)		[sistema verifica que as características são inválidas] (passo 3)		
	3.1	Sistema indica que características são inválidas.	validar os valores atribuídos ao carro.	validaCarro(marca:String, modelo:String, cilindrada:int, potencia:int):boolean
	3.1	Sistema interrompe a criação do carro		Carros

Table 6

## 4.7 AdicionarCarro-C2

Use Case:	Adicionar Carro - C2			
Descrição:	Administrador adiciona um carro da categoria C2 à lista de carros.			
Cenário:	Um administrador opta por adicionar um novo carro da categoria C2. Indica marca, modelo, potência e cilindrada. Para finalizar indica o PAC.			
Pré-Condição:	Utilizador autenticou-se como administrador.			
Pós-Condição:	Carro é adicionado à lista de carros e fica imediatamente disponível para ser utilizado.			
		1. Identificar responsabilidades da LN	2. definir API (identificar métodos)	3. identificar subsistemas (agrupar métodos)
Fluxo Normal				
	1.	Sistema apresenta características necessárias para C2 (Marca, Modelo, Cilindrada, Potência).		
	2.	Administrador indica marca, modelo, cilindrada e potência.		
	3.	Sistema valida características.	validaCarro(marca:String, modelo:String, cilindrada:int, potencia:int):boolean	Carros
	4.	Sistema ajusta fiabilidade para um valor próximo de 80%.	calcular a fiabilidade do carro	Carros
	5.	Sistema pede PAC e indica que o $0 < \text{PAC} \leq 1$ .		
	6.	Administrador indica PAC.		
	7.	Administrador indica que carro não é híbrido.		
	8.	Sistema regista carro e este fica disponível para jogar.	adicionar carro à lista de carros	adicionarCarro(carro: Carro)
Fluxo Alternativo(1)		[administrador indica que carro é híbrido] (passo 7)		
	7.1	Sistema pede potência de motor elétrico.		
	7.2	Administrador indica valor de potência do motor elétrico		Carros
	7.3	Sistema ajusta fiabilidade para um novo valor ( $< 80\%$ ). (retorna a passo 8)	calcular a fiabilidade do carro	Carros
Fluxo Exceção(2)		[sistema verifica que as características são inválidas] (passo 3)		
	3.1	Sistema indica que características são inválidas.	validar os valores atribuídos ao carro.	validaCarro(marca:String, modelo:String, cilindrada:int, potencia:int):boolean
	3.1	Sistema interrompe a criação do carro		

Table 7

## 4.8 AdicionarCarro-GT

Use Case:	Adicionar Carro - GT			
Descrição:	Administrador adiciona um carro da categoria GT à lista de carros.			
Cenário:	Um administrador opta por adicionar um novo carro da categoria GT. Indica marca, modelo, potência e cilindrada.			
Pré-Condição:	Utilizador autenticou-se como administrador.			
Pós-Condição:	Carro é adicionado à lista de carros e fica imediatamente disponível para ser utilizado.			
		1. Identificar responsabilidades da LN	2. definir API (identificar métodos)	3. identificar subsistemas (agrupar métodos)
Fluxo Normal				
	1.	Sistema apresenta características necessárias para GT (Marca, Modelo, Cilindrada, Potência).		
	2.	Administrador indica marca, modelo, cilindrada e potência.		
	3.	Sistema verifica que as características são válidas.	validar os valores atribuídos ao carro	validaCarro(marca:String, modelo:String, cilindrada:int, potencia:int):boolean
	4.	Sistema calcula fiabilidade tendo em conta as características do carro.	calcular fiabilidade do carro	calculaFiabilidade(cilindrada:int, potencia:int):float
	7.	Administrador indica que carro não é híbrido.		
	8.	Sistema regista carro e este fica disponível para jogar.	adicionar carro à lista de carros	adicionarCarro(carro: Carro)
Fluxo Alternativo(1)		[administrador indica que carro é híbrido] (passo 7)		Carros
	7.1	Sistema pede potência de motor elétrico.		
	7.2	Administrador indica valor de potência do motor elétrico.		
	7.3	Sistema calcula fiabilidade tendo em conta as características do carro.	calcular a fiabilidade do carro	calculaFiabilidade(cilindrada:int, potencia:int):float
	7.4	(retorna a passo 8)		
Fluxo Exceção(2)		[sistema indica que as características são inválidas] (passo 3)		
	3.1	Sistema indica que características são inválidas.	validar os valores atribuídos ao carro.	validaCarro(marca:String, modelo:String, cilindrada:int, potencia:int):boolean
	3.1	Sistema interrompe a criação do carro		Carros

Table 8

## 4.9 AdicionarCarro-SC

Use Case:	Adicionar Carro - SC			
Descrição:	Administrador adiciona um carro da categoria SC à lista de carros.			
Cenário:	Um administrador opta por adicionar um novo carro da categoria SC. Indica marca, modelo, potência e cilindrada.			
Pré-Condição:	Utilizador autenticou-se como administrador.			
Pós-Condição:	Carro é adicionado à lista de carros e fica imediatamente disponível para ser utilizado.			
		1. Identificar responsabilidades da LN	2. definir API (identificar métodos)	3. identificar subsistemas (agrupar métodos)
Fluxo Normal				
	1.	Sistema apresenta características necessárias para SC (Marca, Modelo, Cilindrada, Potência).		
	2.	Administrador indica marca, modelo, cilindrada e potência.		
	3.	Sistema verifica que as características são válidas.	validar os valores atribuídos ao carro	validaCarro(marca:String, modelo:String, cilindrada:int, potencia:int):boolean
	4.	Sistema calcula fiabilidade do carro.		calculaFabilidade(cilindrada:int, potencia:int):float
	8.	Sistema regista carro e este fica disponível para jogar.	adicionar carro à lista de carros	adicionarCarro(carro:Carro)
Fluxo Exceção(1)		[sistema verifica que as características são inválidas] (passo 3)		
	3.1	Sistema indica que características são inválidas.	validar os valores atribuídos ao carro.	validaCarro(marca:String, modelo:String, cilindrada:int, potencia:int):boolean
	3.1	Sistema interrompe a criação do carro		

Table 9

## 4.10 AdicionarPiloto

Use Case:	<b>Adicionar Piloto</b>			
Descrição:	Administrador adiciona um piloto à lista de pilotos.			
Cenário:	Cenário 4			
Pré-Condição:	Utilizador autenticou-se como administrador.			
Pós-Condição:	Piloto é adicionado à lista de campeonatos.			
		1. Identificar responsabilidades da LN	2. definir API (identificar métodos)	3. identificar subsistemas (agrupar métodos)
Fluxo Normal				
	1.	Administrador indica que pretende criar um piloto.		
	2.	Sistema pede o nome do piloto e os respetivos níveis de perícia.		
	3.	Administrador atribui nome e níveis de perícia (CTS e SVA) ao piloto.		
	4.	Faz a confirmação, regista e adiciona piloto.	registar o piloto adicionaPiloto(nome:String, cts:double, sva:double):	Pilotos

Table 10: Use Case "Adicionar Piloto"

## 4.11 Registar Jogador no Campeonato

Use Case:	Registrar Jogador no Campeonato			
Descrição:	Jogador inscreve-se num campeonato da lista de campeonatos.			
Cenário:	Um jogador decide inscrever-se num campeonato.			
Pré-Condição:	Utilizador autenticou-se como jogador e existe pelo menos um campeonato para jogar, um carro e um piloto.			
Pós-Condição:	Jogador fica inscrito no campeonato.			
		1. Identificar responsabilidades da LN	2. definir API (identificar métodos)	3. identificar subsistemas (agrupar métodos)
Fluxo Normal				
	1. Sistema procura e apresenta os de campeonatos	apresentar os campeonatos disponíveis para jogar	getCampeonatos(): HashMap<String, Campeonato>	Campeonato
	2. Jogador escolhe o campeonato que se pretende inscrever			
	3. Sistema apresenta campeonato	apresentar os pilotos disponíveis		
	4. Sistema procura e apresenta as categorias de carros disponíveis	apresentar as categorias de carros disponíveis para jogar	getCategorias(): List<Categoria>	Carro
	5. Utilizador escolhe uma categoria			
	6. Sistema procura e apresenta carros disponíveis para a categoria escolhida	apresentar carros disponíveis de uma certa categoria para jogar	getCarros(c: Categoria):List<Carro>	Carro
	7. Utilizador escolhe um carro			
	8. Sistema procura e apresenta pilotos disponíveis	apresentar pilotos disponíveis para jogar	getPilotos():List<Piloto>	Piloto
	9. Utilizador escolhe um piloto			
	10. Sistema regista a inscrição	registar a inscrição do jogador	registaJogadorCampeonato (jogador: Jogador,nomecampeonato: Campeonato, carro: Carro, piloto: Piloto):	Campeonato

Table 11

## 4.12 Preparar Corrida

Use Case:	Preparar Corrida				
Descrição:	Jogadores colocam se prontos para iniciar a corrida				
Cenário:	<p>Quando todos estão registados, dá-se início ao campeonato. As condições da primeira corrida são apresentadas: o circuito é o “Gualtar Campus” e a situação meteorológica é de tempo seco (a outra possibilidade seria chuva). Cada um dos jogadores decide se pretende alterar a afinação do carro, tendo em consideração que, por se tratar de uma campeonato com três corridas, apenas poderão fazer duas afinações ao longo do mesmo. Após considerar as características do circuito, do carro e do piloto, o Francisco decide alterar a afinação (possível por se tratar de um C2) e aumenta a downforce de 0.5 (valor neutro) para 0.7. Ao aumentar a downforce, sacrifica alguma velocidade para ter maior estabilidade em curva. Deste modo, troca alguma capacidade de ultrapassar em reta por capacidade de ultrapassar em curva, compensando a menor propensão para o risco do piloto. Finalmente, todos os jogadores devem escolher os pneus e modo do motor a usar na corrida. Dos três tipos de pneu disponíveis, neste momento, no jogo (macio, duro e chuva) o Francisco escolhe pneus macios, o que permite ter melhor desempenho no início da corrida, à custa do desempenho no final. Dos três modos de funcionamento do motor (conservador, normal ou agressivo), o Francisco escolhe o agressivo, aumentando o desempenho do carro à custa de maior probabilidade de o motor ter uma avaria.</p>				
Pré-Condição:	Todos os jogadores estão registados no campeonato				
Pós-Condição:	Todos os jogadores estão prontos				
Fluxo Normal		1. Identificar responsabilidades da LN	2. definir API (identificar métodos)	3. identificar subsistemas (agrupar métodos)	
	1. O sistema apresenta o circuito e a situação meteorológica.				
	2. Sistema verifica se o jogador fez 2/3 ou menos das alterações de afinação possíveis.	verificar se o número de afinações feitas no campeonato são inferiores a 2/3 do número de corridas total	validarAfinacao(nomeJogador: String, nomeCampeonato:String):boolean	Campeonato	
	2. Jogador altera ou não os pneus, o funcionamento do moto e o perfil aerodinâmico do carro	Alterar o tipo de pneus do carro, o modo de motor e o perfil aerodinâmico	alteraAfincacao(nomeCampeonato: String, nomeJogador: String, pneu : ENUM(PNEU), motor : ENUM(MOTOR), pac:float)	Campeonato	
	3. O Jogador, no fim de fazer alterações, indica que está pronto.	Alterar estado de jogador para pronto e verificar se pode começar a simulação	prepararCorrida(nomeCampeonato: String, indiceCorrida: int, jogador: Jogador)	Campeonato	

Table 12

## 4.13 Classificação Corrida

Use Case:	Consultar Classificação Corrida			
Descrição:	Visualização dos resultados de uma corrida			
Pré-Condição:	O Jogador tem de estar autenticado e registado no campeonato a interagir			
Pós-Condição:	True			
		1. Identificar responsabilidades da LN	2. definir API (identificar métodos)	3. identificar subsistemas (agrupar métodos)
Fluxo Normal				
	1.	O Jogador escolhe o campeonato e a corrida à qual quer consultar os resultados		
	2.	O sistema devolve os resultados da corrida	getResultadosCorrida(nomeCampeonato: String, corrida: int)	Campeonato
Fluxo Exceção		[Não existem resultados]		
	2.1	O sistema indica que a corrida não tem os resultados disponíveis		

Table 13

## 4.14 Classificação Campeonato

Use Case:	Consultar Classificação Campeonato			
Descrição:	Visualização dos resultados de um Campeonato			
Pré-Condição:	O Jogador tem de estar autenticado e registo no campeonato a interagir			
Pós-Condição:	True			
		1. Identificar responsabilidades da LN	2. definir API (identificar métodos)	3. identificar subsistemas (agrupar métodos)
Fluxo Normal				
	1.	O Jogador escolhe o campeonato ao qual quer consultar os resultados		
	2.	O sistema devolve os resultados do campeonato	getResultados(nomeCampeonato: String)	Campeonato
Fluxo Exceção		[Não existem resultados]		
	2.1	O sistema indica que o campeonato não tem os resultados disponíveis		

Table 14

## 5 Diagrama de Componentes

Decidimos organizar o nosso diagrama de componentes conforme é apresentado na figura 4. Esta organização visa mostrar o relacionamento entre os diferentes componentes do sistema. Para além dos subsistemas e interfaces necessárias de UI e DL, sub-dividimos o nosso sistema nos seguintes sub-sistemas: "Utilizadores", "Campeonatos", "Circuitos", "Carros" e "Pilotos" que implementam, respetivamente, as interfaces "IUUtilizadores", "ICampeonatos", "ICircuitos", "ICarros" e "IPilotos". Podemos verificar que a definição destes sub-sistemas corresponde diretamente às diferentes entidades que são representadas nos cenários do enunciado. Tomamos, também, a decisão de incluir a corrida dentro do sub-sistema dos "Campeonatos". Consideramos que esta pode ser uma boa abordagem, visto que as corridas são jogadas a partir de um determinado campeonato e, quando simulamos um jogo, estas duas entidades são as mais utilizadas e são manipuladas em conjunto. Esta decisão permite-nos implementar na interface "ICampeonatos" métodos relativos à simulação do jogo em si.

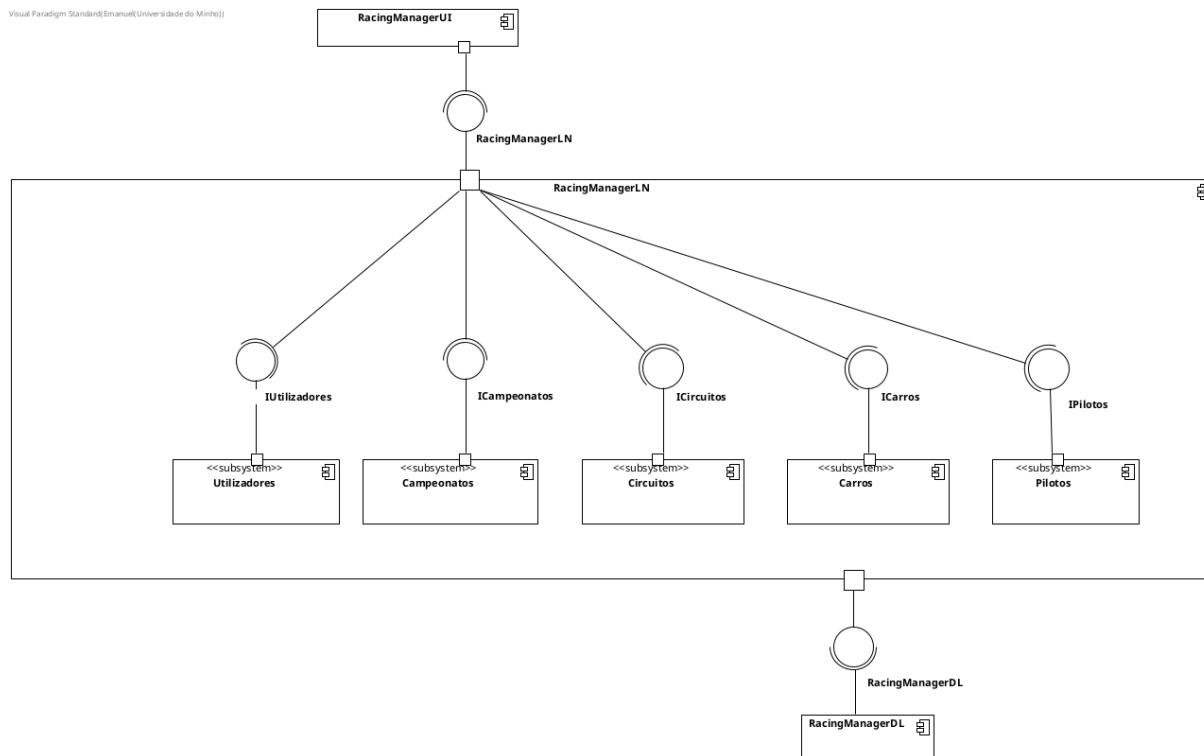


Figure 4: Diagrama de Componentes

## 6 Diagrama de Classes

### 6.1 O que foi aproveitado do código legado?

As classes que representamos nos nossos subsistemas refletem, em parte, o código legado disponibilizado pelos docentes. Antes de observar de forma mais ampla os diagramas de classes para os diferentes subsistemas, vamos abordar em específico a implementação de algumas das classes, e com especial atenção às principais similaridades entre a arquitetura que definimos e o código.

#### 6.1.1 Classe Campeonato

O campeonato contém uma lista de corridas.

#### 6.1.2 Classe Corrida

A corrida contém um circuito e um clima. Uma diferença é o facto de termos usado uma classe de enumeração para definir as opções climatéricas, em vez de um inteiro. Para além disso, associamos a cada corrida um resultado para cada jogador.

#### 6.1.3 Classe Circuito

O circuito apresenta, para além do nome, distancia e número de voltas presentes no código legado, atributos como: número de chicane, curvas e retas e os respetivos graus de dificuldade de ultrapassagem. Adicionamos estes atributos porque são necessários para a simulação da corrida por parte do sistema.

#### 6.1.4 Classe Carros

Em relação ao carro, mantivemos maior parte dos atributos definidos no código legado. Mantivemos, também, a implementação do carro como uma classe abstrata. No entanto, visto que o nosso problema faz referência à existência de mais categorias, implementamos os C1, C2, GT e SC, que são todos tipos de carros diferentes, com limitações de cilindrada e potência distintas. Para além disso, cada C1, C2 e GT pode ainda ser híbrido, ou seja, C1H, C2H e GTH, respetivamente.

#### 6.1.5 Classe Piloto

Para a definição do piloto singimo-nos aos requisitos por parte do enunciado, pelo que este tem como atributos apenas o seu nome e níveis de perícia.

## 6.2 SubSistema Utilizadores

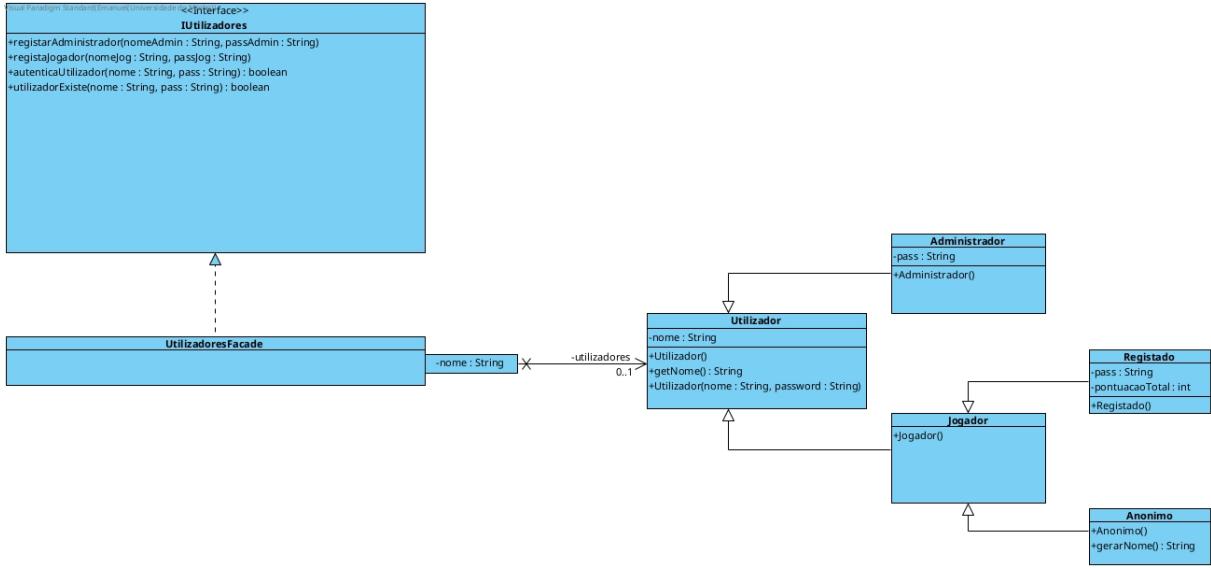


Figure 5: Diagrama de Classes dos Utilizadores

Neste subsistema, para além da classe UtilizadoresFacade, podemos encontrar a classe abstrata Utilizador. No nosso sistema, um utilizador pode ser um administrador ou um jogador. Caso este seja um jogador, pode ainda estar registrado na aplicação ou ser apenas anónimo. Neste subsistema estão incluídos os métodos relativos ao registo de utilizadores, bem como a sua autenticação.

## 6.3 SubSistema Campeonatos

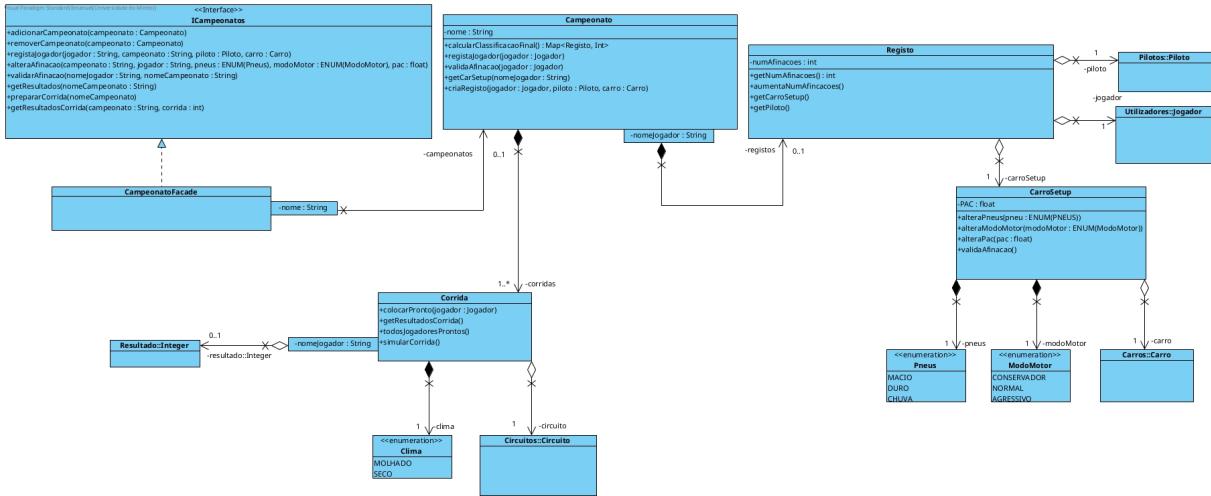


Figure 6: Diagrama de Classes do Campeonatos

O subsistema "Campeonatos" é o mais complexo da nossa arquitetura. Como já foi dito anteriormente este subsistema inclui não só a classe Campeonato, mas também a classe Corrida. Neste subsistema podemos encontrar todos métodos relativos ao registo de jogadores nos campeonatos, simulação de corridas, realização de afinações, obtenção das classificações, etc.

## 6.4 SubSistema Circuitos

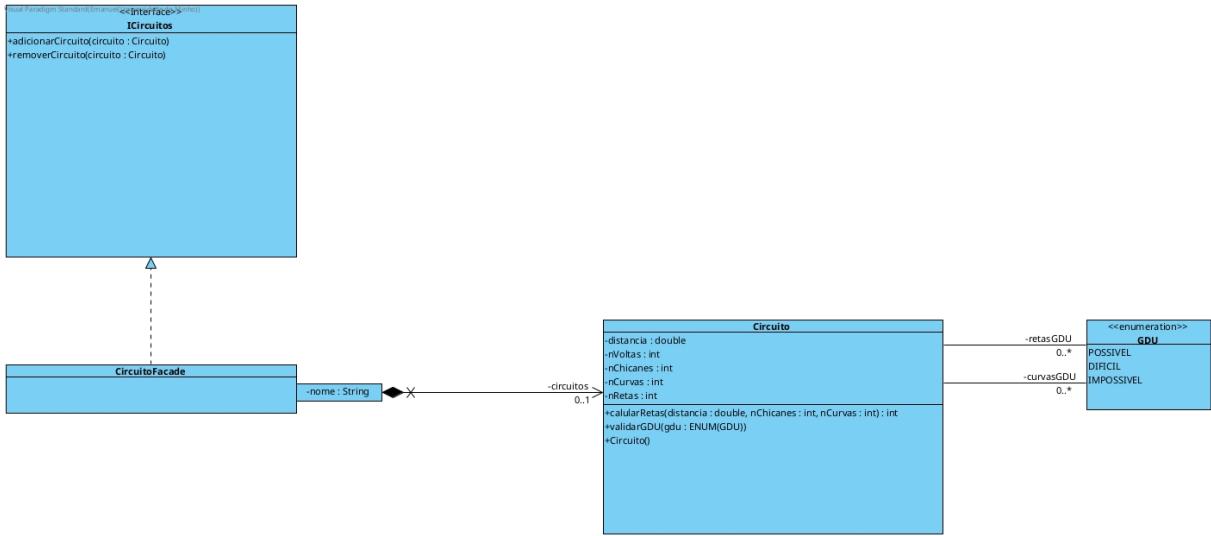


Figure 7: Diagrama de Classes do Circuitos

Neste subsistema, para além da classe CircuitosFacade, podemos encontrar a classe Circuito que diz respeito à entidade do circuito e as suas características, como as curvas, chicane, retas e os respectivos graus de dificuldade de ultrapassagem.

## 6.5 SubSistema Carros

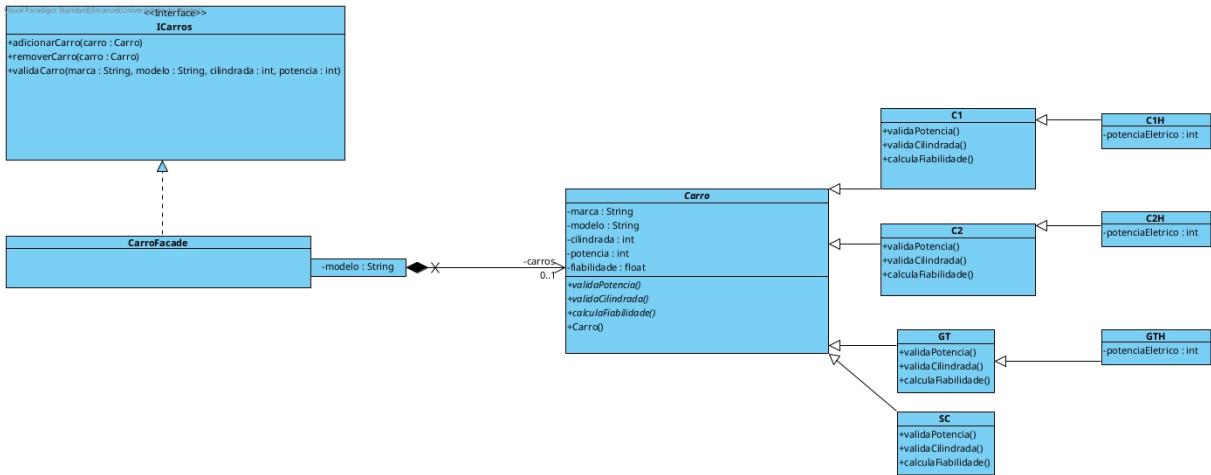


Figure 8: Diagrama de Classes dos Carros

Neste subsistema, para além da classe CarrosFacade, podemos encontrar a classe abstrata Carro. Esta classe indica os atributos que todos os carros devem ter. Como já foi dito acima, cada carro é de uma determinada categoria, podendo este ser um C1, C2, GT ou SC.

## 7 SubSistema Pilotos

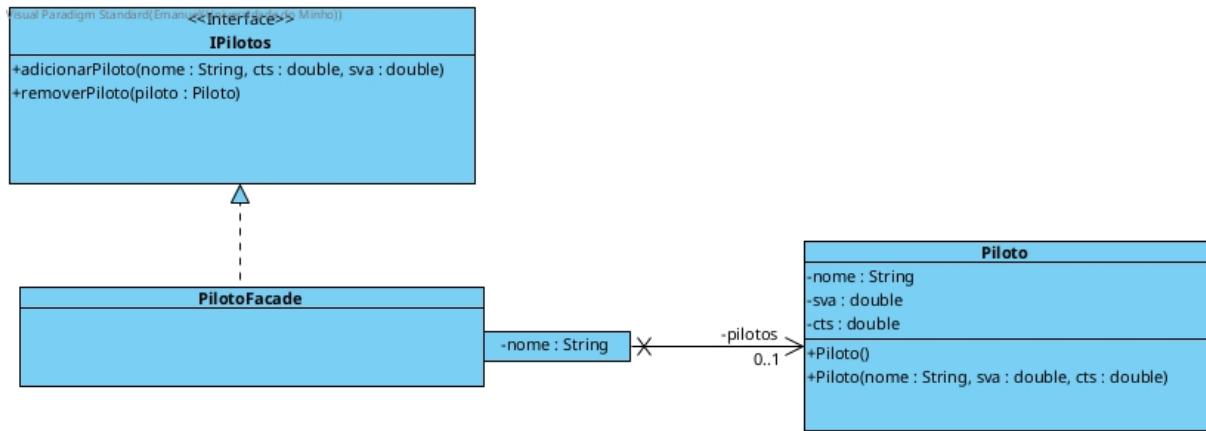


Figure 9: Diagrama de Classes do Pilotos

Neste subsistema, para além da classe PilotosFacade, podemos encontrar a classe Piloto que indica alguns atributos como o seu nome os seus níveis de pericia.

## 8 Diagramas de Sequência

Os diagramas de sequência para os métodos que apresentamos de seguida visam mostrar a interação entre os diferentes objetos do sistema.

### 8.1 validaAfinacoes()

Este método *validaAfinacoes()* faz parte da interface "ICampeonatos" e é implementado pelo Campeonato-Facade. Permite verificar se é possível ou não um determinado jogador fazer uma afinação ao seu carro no campeonato em questão. Não é possível um jogador fazer mais afinações do que 2/3 do número de corridas do campeonato. A partir do seguinte diagrama de sequência, conseguimos observar os passos até obter um valor booleano ("validacao"). Este valor depende dos valores obtidos com a ajuda das funções *getNumCorridas()* da classe Campeonato e a função *length* para as listas.

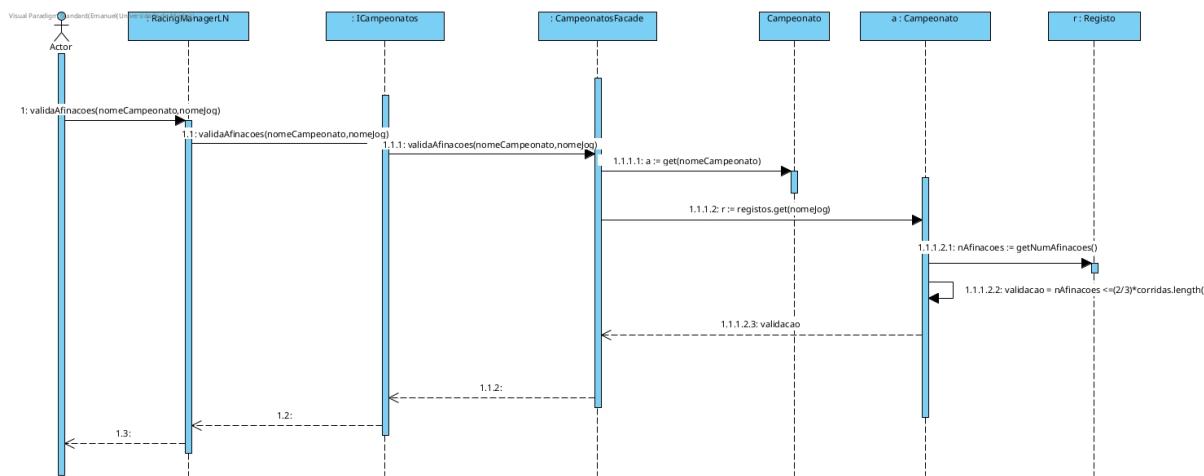


Figure 10: Diagrama de Sequência *validaAfinacoes()*

## 8.2 alteraAfinacao()

Este método *alteraAfinacao()* faz parte da interface "ICampeonatos" e é implementado pelo Campeonato-Facade. Permite alterar o modo de motor, o perfil aerodinâmico e o tipo de pneus no carro.

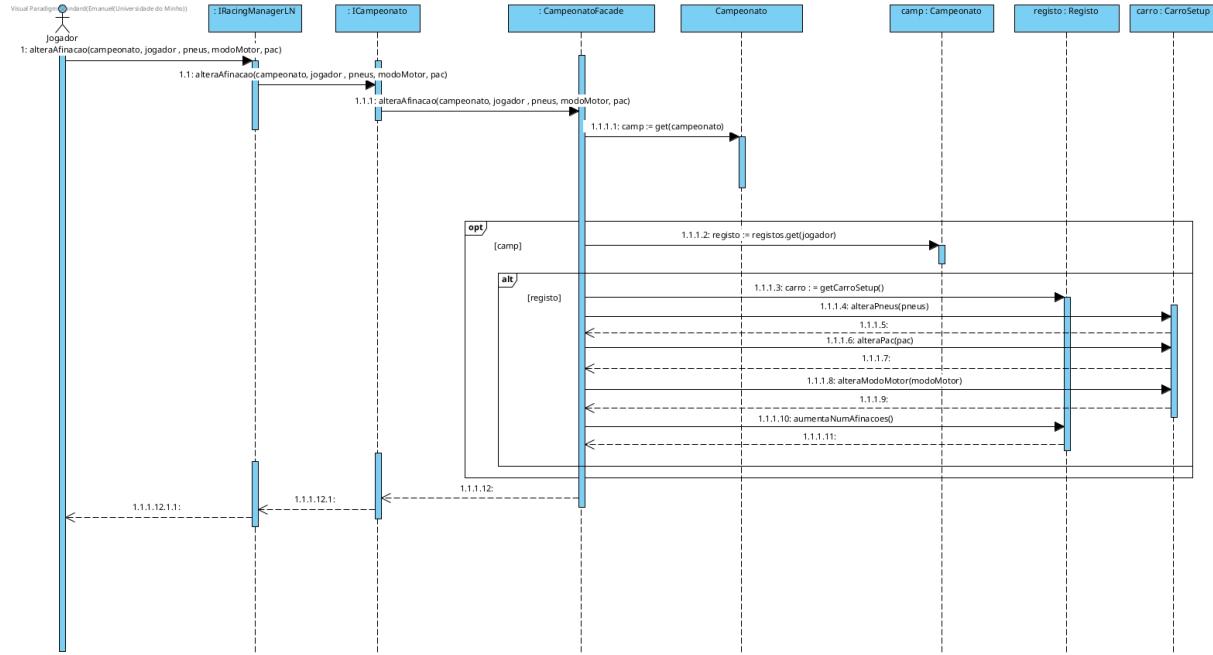


Figure 11: Diagrama de Sequência *alteraAfinacao()*

## 8.3 getResultados()

O método *getResultados()* faz parte da interface "ICampeonatos" e é implementado pelo Campeonato-Facade. Acede ao campeonato a partir do nome do campeonato dado como argumento e devolve os resultados obtidos a partir da função *calcularClassificacaoFinal()* da classe Campeonato.

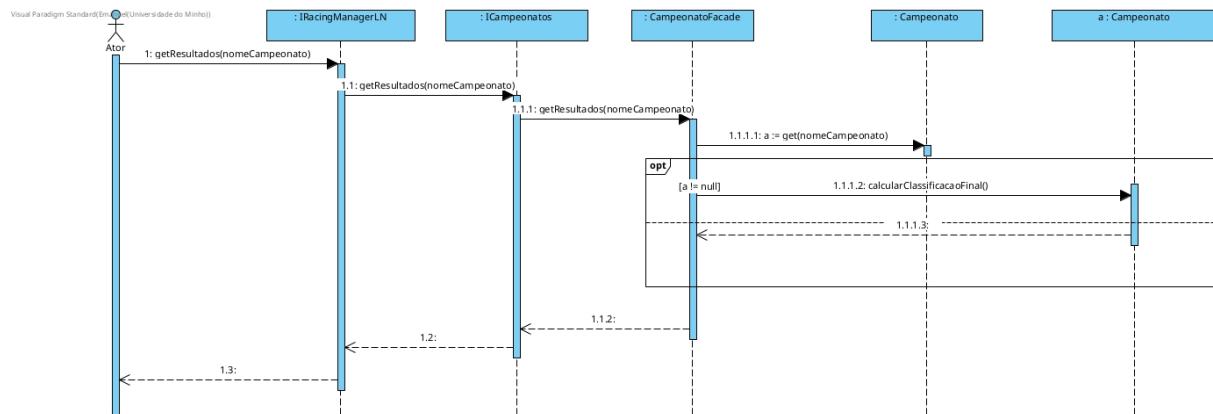


Figure 12: Diagrama de Sequência *getResultados()*

## 8.4 getResultsosCorrida()

O método *getResultsosCorrida()* faz parte da interface "ICampeonatos" e é implementado pelo CampeonadoFacade. A partir de um nome de campeonato e de um índice para aceder à corrida que pretende, é devolvido os resultados de uma determinada corrida, obtidos a partir da função *getResultsosCorrida()* da classe Corrida.

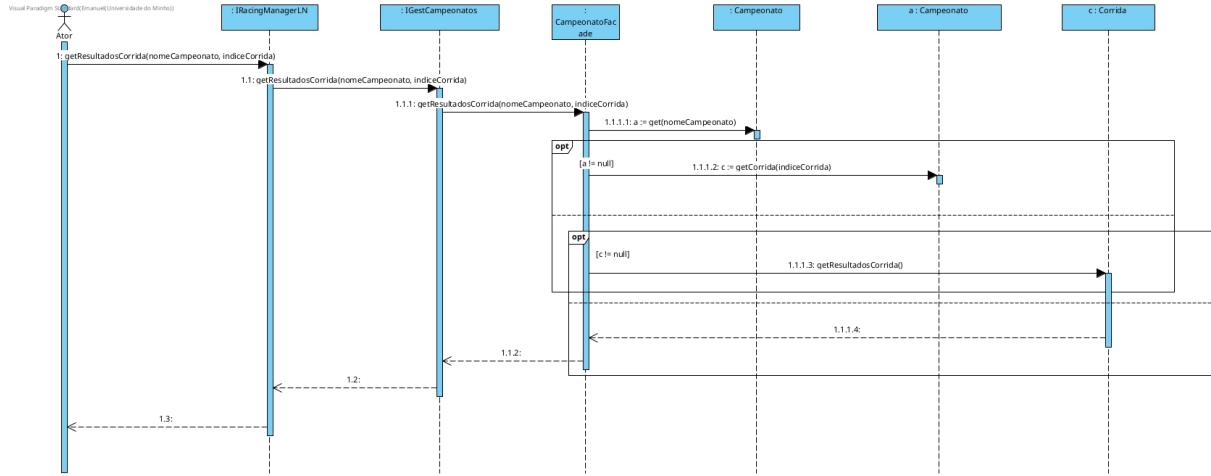


Figure 13: Diagrama de Sequência *getResultsosCorrida()*

## 8.5 prepararCorrida()

O método *prepararCorrida()* faz parte da interface "ICampeonatos" e é implementado pelo Campeonado-Facade. Coloca o jogador a pronto e verifica se pode começar a corrida verificando se todos estão prontos, caso estejam começa a simulação

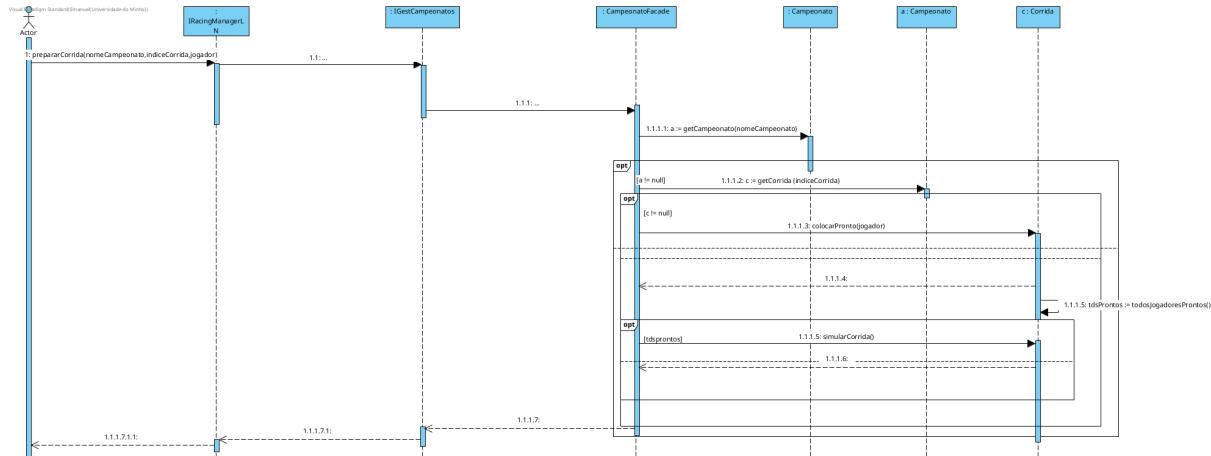


Figure 14: Diagrama de Sequência *prepararCorrida()*

## 8.6 registrarJogador()

O método *registrarJogador()* faz parte da interface "ICampeonatos" e é implementado pelo Campeonato-Facade. Cria um registo num determinado campeonato com as informações do jogador e do carro e piloto escolhidos.

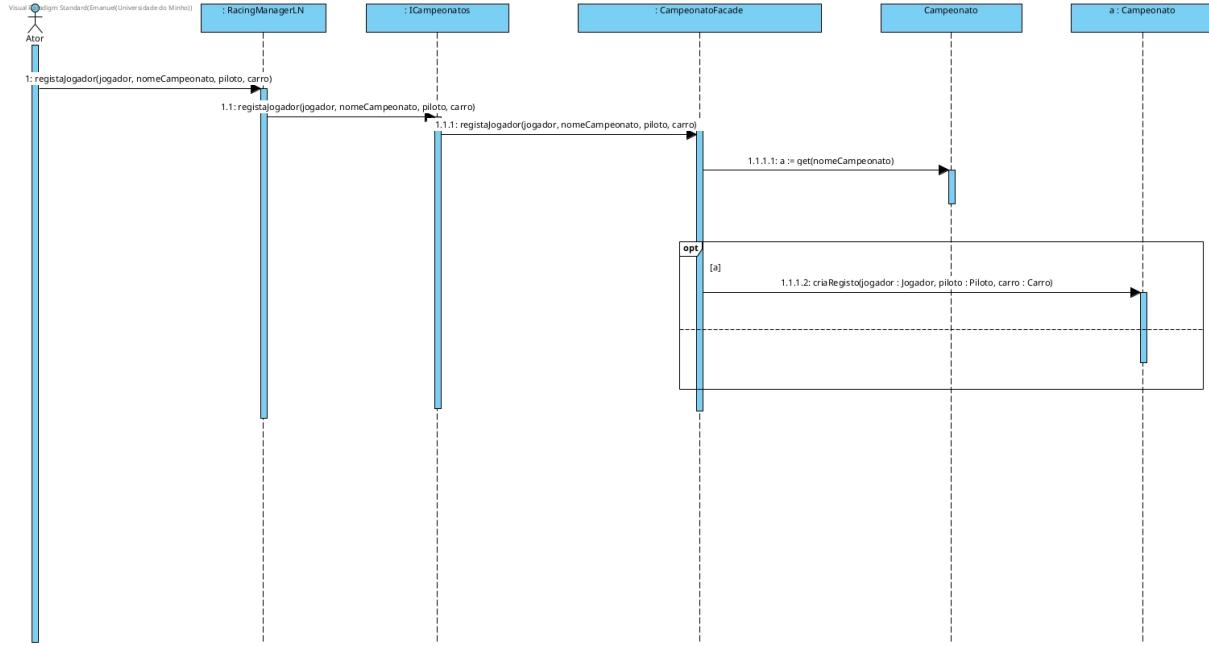


Figure 15: Diagrama de Sequência *registrarJogador()*

## 9 Implementação

### 9.1 Arquitetura

Para a implementação do nosso programa decidimos utilizar a arquitetura MVC (*Model-View-Controller*) de modo a separar o código em camada de dados, camada de vista e interação com o utilizador e camada de controlo do fluxo do programa. Apesar dos módulos definidos estarem de acordo com o Diagrama de Classes da fase 2, foram ajustados de forma a incluir persistência de dados com DAOs.

### 9.2 Base de Dados

Recorremos a JDBC e MariaDB para fazer a ligação entre o nosso código Java e uma base de dados.

#### 9.2.1 Modelo lógico

A partir da imagem abaixo é possível identificar quais as tabelas que decidimos incluir no modelo lógico da nossa base de dados, tendo em conta o contexto da nossa arquitetura e das estratégias usadas. É, ainda, visível os diferentes atributos de cada entidade e os relacionamentos entre as entidades.

#### 9.2.2 Povoamento

As tabelas mostradas na figura foram povoadas na base de dados a partir da *User Interface*, que permite aos administradores adicionar entidades e aos jogadores registarem-se em campeonatos, etc.

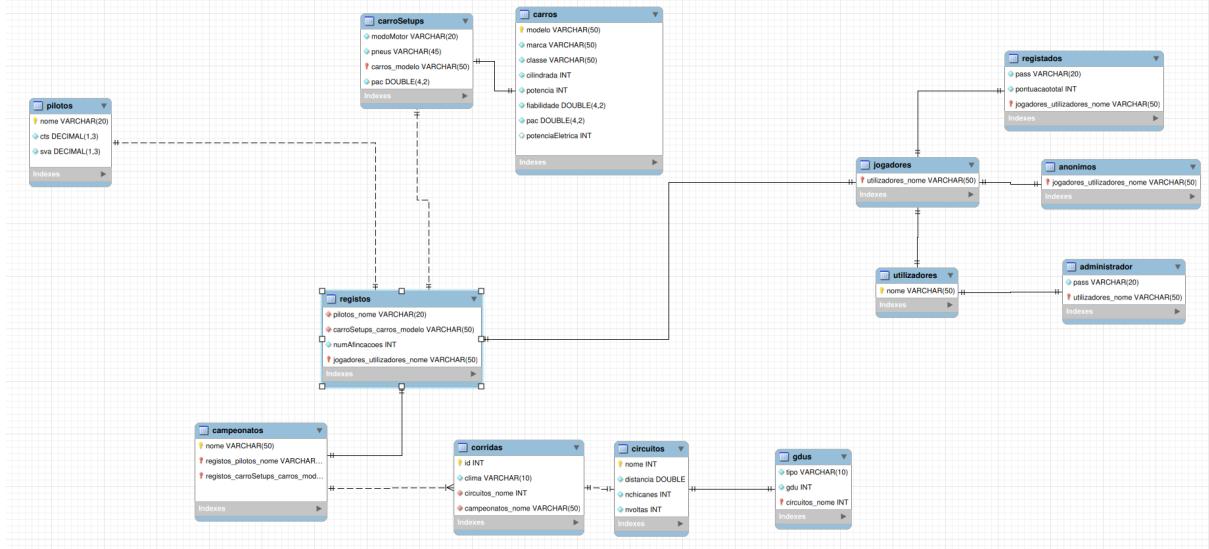


Figure 16: Modelo lógico da Base de Dados

### 9.2.3 ORM e DAOs

Decidimos tirar partido do uso de conceitos de ORM (*Object-Relational Mapping*) e de DAOs (*Data Access Objects*) em conjunto de modo a separar a lógica de acesso à base de dados da lógica de negócios do nosso programa. Isto permite mapear objetos Java para entidades e vice-versa, facilitando e oficializando o processo de manipulação de dados. O DAO fornece uma interface para o acesso às entidades da base de dados. Todos estes fatores contribuem para a persistência dos dados da nossa aplicação.

## **10 Análise Crítica**

Durante a realização deste trabalho deparamo-nos com várias indecisões no que toca à estruturação da arquitetura do nosso sistema e desenho dos nossos diagramas. Uma das maiores dificuldades foi realizar o diagrama de classes para o subsistema dos Campeonatos, visto que tínhamos muitas opções relativas a como organizar a informação.

Outra dificuldade que surgiu foi a tradução dos métodos da API identificados nos Use Cases para o diagrama de classes. Reparamos que alterações tinham que ser feitas aos nossos Use Cases da fase 1 de forma a encontrar alguma concordância.

Na realização da terceira fase do projeto, o grupo encontrou dificuldades, principalmente, na implementação de DAOs. O grande nível de abstração das classes Utilizador e Carro revelou-se desafiante no que toca à sua definição em tabelas no modelo lógico de base de dados e à implementação da interface para DAOs respetivos.

## **11 Conclusão**

Concluindo, o grupo considera que, apesar de ter conhecimento de que não atingiu todos os objetivos pretendidos, adquiriu e aplicou conceitos importantes relativos à terceira fase, principalmente a implementação dos DAOs e a redefinição da arquitetura adaptada.