Universidade do Minho
Escola de Engenharia
Licenciatura em Engenharia Informática

Ano Letivo de 2002/2023

Repositório Github

https://github.com/cgustavop/DSS-PL

Racing Manager

7 de janeiro de 2023

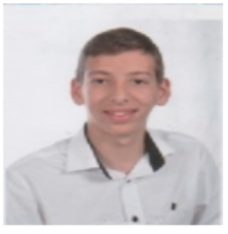
DSS

Vasco Manuel Araújo Andrade de Oliveira
96361

Carlos Gustavo Silva Pereira
96867

Cláudio Alexandre Freitas Bessa
97063

Carlos Emanuel Leite Machado
97114



Tiago André Mendes Oliveira
97254

Índice

Lista de Figuras

1 Introdução

Racing Manager é um simulador de corridas local, onde um ou mais jogadores competem em campeonatos de forma simulada pelo sistema. Nesta aplicação existem três atores principais, os gestores, os administradores e por fim os jogadores. Os administradores são utilizadores escolhidos pelos gestor para adicionar *features* ao simulador. Dentro dessas *features* temos como exemplo, criação de campeonatos, adição de circuitos, adição de carros e adição de pilotos entre outros. Os jogadores, por sua vez, são capazes de competirem entre si localmente em campeonatos disponíveis para os mesmos. Possuem a opção de escolherem o seu carro, piloto e adaptarem-se, afinando os carros, consoante as condições dos circuitos geradas pelo sistema. Ao fim de cada corrida os jogadores presentes, são submetidos a um sistema de *ranking*, caso estejam autenticados, para um *ranking* final no fim do campeonato.

Tal como outros sistemas de *software* na indústria, possui duas versões do jogo, sendo uma delas a versão base e a versão *premium*. A sua grande diferença é possível ser visualizada durante a simulação da corrida, onde as posições relativas entre cada competidor e outras situações, ultrapassagens, despistes, entre outros, em vez de serem atualizadas ao fim de cada volta ou no fim de cada segmento, são atualizadas em tempo real.

Área de Aplicação: Desenho e arquitetura de sistemas de *software*.
Palavras-Chave: Base de Dados, Jogos, Aplicação, Simulação

2 Objetivos

Nesta fase já não tanto introdutória do projeto *Racing Manager* é pretendido a execução de algumas tarefas cruciais para todo o sistema de *software*, tarefas essas acrescentadas a tarefas já efetuadas, e agora melhoradas, na primeira fase:

- Compreensão do solicitado no enunciado.
- Criação modelo de domínio com as entidades relevantes
- Criação do modelo de *use cases*, diagrama mais as especificações, com as funcionalidades propostas
- Criação de diagrama de componentes e respetivos subcomponentes com os seus subsistemas
- Criação de diagramas de classes para cada subsistema existente no diagrama de componentes
- Criação de diagramas de sequências e/ou documentação OCL para cada método de operador lógico por subsistema

3 Modelo de Domínio

O Modelo de Domínio descreve as entidades do contexto em que o sistema deve ser implementado. Um modelo como este é importante para estabelecer algumas regras sobre as entidades e pensar como estas vão funcionar no respetivo sistema.

Para a resolução do Modelo de Domínios começamos por identificar, através do enunciado apresentado, as entidades do sistema e de seguida as suas associações, com as suas respetivas multiplicidades. É de apontar que o desenvolvimento não foi linear e, como tal, foi sofrendo alterações ao longo do tempo.

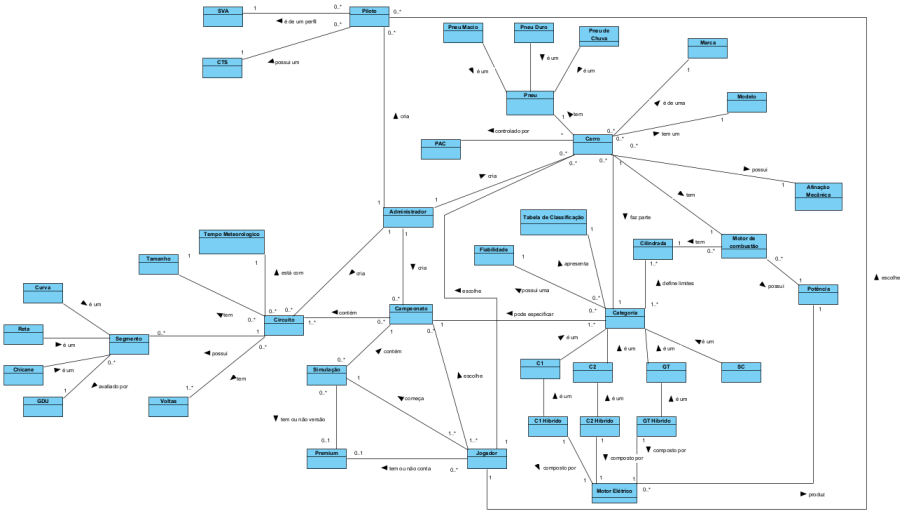


Figura 3.1: Modelo de domínio

3.1 Entidades presentes no Modelo

- Administrador, Piloto, SVA, CTS, Campeonato, Circuito, Voltas, Tempo Metereológico, Tamanho, Voltas, Segmento(Curva, Reta, Chicane, GDU),
- Carro, Marca, Modelo, Cilindrada, Pneu(Macio, Duro, de Chuva), Motor de Combustão, Motor Elétrico, Potência, Fiabilidade, Afinação Mecânica, Categoria(C1 (Híbrido), C2(Híbrido), GT(Híbrido), SC),
- Jogador, Tabela de Classificação, Simulação, Premium.

3.2 Leitura do Modelo de Domínios

O **Administrador** pode criar:

- Um **Campeonato**, com a respetiva lista de **Circuitos** pretendidos;
- Um **Circuito**, com os seus componentes: **Voltas, Tempo Metereológico, Tamanho, Voltas, Segmento (Curva, Reta, Chicane, GDU)**;
- Um **Carro**, com os seus componentes: **Marca, Modelo, Cilindrada, Motor de Combustão, Potência, Fiabilidade, PAC, Categoria(C1(Híbrido), C2(Híbrido), GT(Híbrido), SC)**, Motor Elétrico (só é preciso fornecer se o carro for híbrido);
- Um **Piloto**, com o seu **SVA** e **CTS**.

Nota: As entidades **Campeonato**, **Circuito** e **Piloto** precisam de um nome, portanto nós interpretamos que as entidades, neste caso, são o próprio nome.

O **Jogador**, para começar a jogar, precisa primeiro de escolher o **Campeonato** e de seguida o **Piloto** e o **Carro** pretendidos. Ainda antes de começar a jogar, o **Jogador** pode escolher mudar a **Afinação Mecânica**, os **Pneus(Macio, Duro, de Chuva)** e o **PAC** do **Carro**. Depois dessa fase começa a **Simulação** do **Campeonato**, em que poderá ser **Premium** ou não, dependendo se o **Jogador** também o é. Ao fim de realização de cada **Circuito** o **Jogador** pode escolher mudar a **Afinação Mecânica** do **Carro** outra vez. No fim da **Simulação** do **Campeonato**, os lugares de cada **Jogador** vão ser apresentados na **Tabela de classificação** da respetiva **Categoria** do **Carro**.

4 Use case

O modelo de Use Cases descreve as interações entre o sistema e o seu ambiente, fazendo parte deste os utilizadores e outros sistemas externos. Com este modelo podemos melhor guiar as funcionalidades requiridas do programa a desenvolver.

Na construção deste modelo analisámos os cenários no enunciado e identificamos os atores e as funcionalidades que estes requiriam.

Nota: Neste relatório não mostramos os cenários de *use cases* utilizados, mas eles estão presentes nas especificações dos use cases enviado em anexo.

4.1 Diagrama de use cases

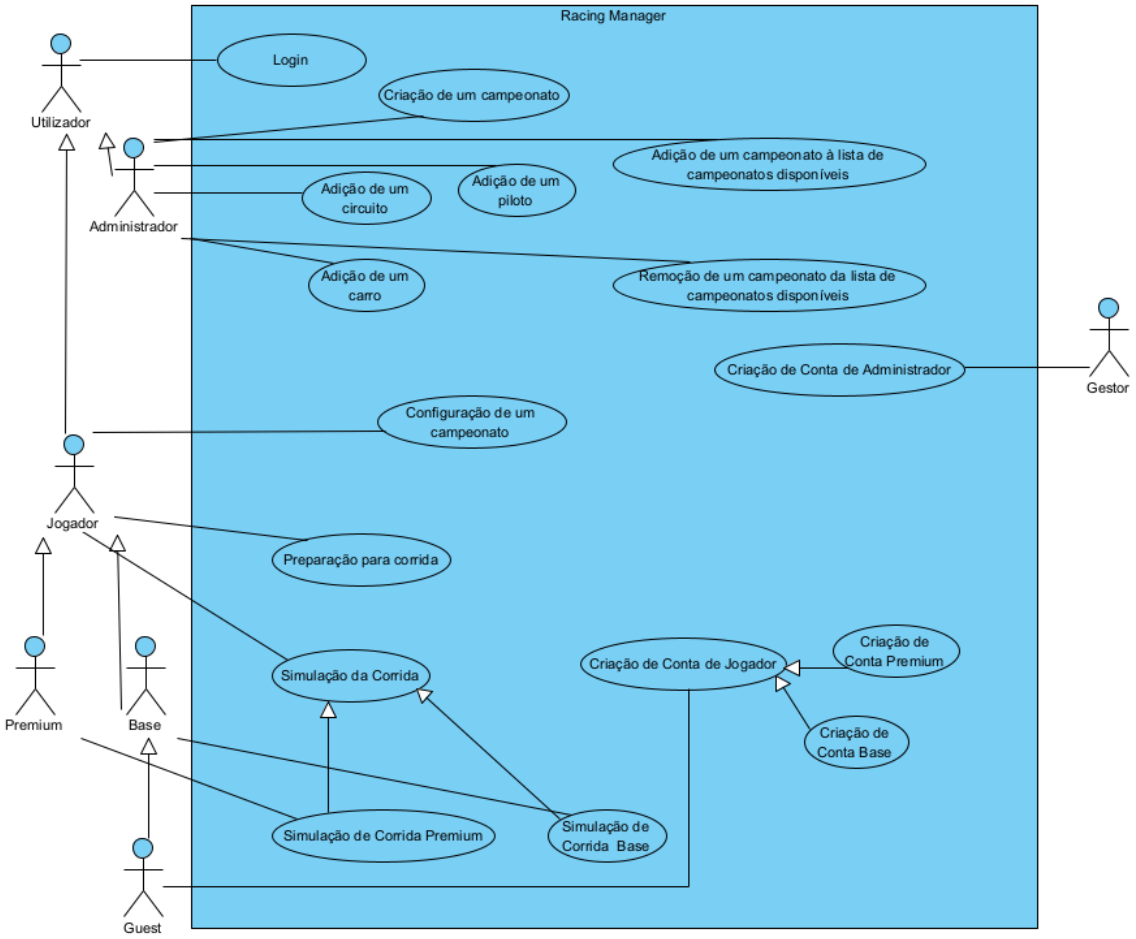


Figura 4.1: Diagrama dos *use cases*

4.2 Leitura do Diagrama de use cases

O ator **Utilizador** serve para mostrar que qualquer ator, tirando o **Gestor**, poderá efetuar o Login. Decidimos esta solução pois achamos que qualquer pessoa poderia querer trocar de conta a qualquer momento ou poderia ainda não ter efetuado o login assim que ligou o jogo.

O ator **Administrador** poderá criar campeonatos e adicionar circuitos, carros e pilotos, e também adicionar ou remover campeonatos na lista de campeonatos disponíveis, para o caso do **Administrador** achar que o campeonato deva ou não dar para jogar pelos **Jogadores**.

O ator **Jogador** poderá então configurar um campeonato (escolhendo o campeonato o carro e o piloto para poder começar a jogar), preparar-se para a corrida (mudando as afinações mecânicas, os pneus e o PAC do carro) e, depois de se preparar, começar a simulação da corrida, em que, dependedo se ele for um jogador **Base** ou **Premium**, terá direito á simulação de corrida base ou premium.

O ator **Guest** representa as pessoas que não têm conta ou que ainda não fizeram Login, podendo, assim, criar uma conta premium ou base, e tendo as mesmas funcionalidades que um **Jogador Base**.

Para se poder criar uma conta de administrador é preciso que o ator que tenha controlo dos registos das contas lhe adicione a conta, sendo esse ator o **Gestor**.

4.3 Atores e use cases

Atores	Use Cases
Gestor	Criação da conta de Administrador
Utilizador	Login
Guest	Criação da conta de Jogador Base
	Criação da conta de Jogador Premium
Administrador	Criação de um campeonato
	Adição de um campeonato à lista de campeonatos disponíveis
	Remoção de um campeonato da lista de campeonatos disponíveis
	Adição de um circuito
	Adição de um carro
	Adição de um piloto
Jogador	Configuração de um campeonato
	Preparação para a corrida
	Simulação da Corrida Base
	Simulação da Corrida Premium

Tabela 4.1: Atores e *use cases*

4.4 Especificações de use case

Use Case	Criação da conta de Administrador	
Ator	Gestor	
Pré-condição	True	
Pós-condição	Nova conta de Administrador adicionada ao sistema	
	Ator	**Sistema**
Fluxo Normal	1. Gestor fornece nome e palavra-passe	
		2. Sistema verifica disponibilidade do nome
		3. Sistema regista a conta como "administrador"
Fluxo alternativo (1) [Nome indisponível](passo 2)		1. Sistema informa que o nome não está disponível
		2. Regressa a 1

Tabela 4.2: Especificação da Criação conta de Administrador

Use Case	Login	
Ator	Guest (Administrador ou Jogador)	
Pré-condição	True	
Pós-condição	Utilizador autenticado (como "administrador"ou "jogador")	
	Ator	**Sistema**
Fluxo Normal	1. Utilizador fornece um nome e uma palavra passe	
		2. Sistema verifica validade dos dados
		3. Sistema verifica que o utilizador é um "jogador"
		4.Sistema autentica o login do utilizador como "jogador"
Fluxo alternativo (1) [Dados de login de administrador] (passo 3)		1. Sistema verifica que o utilizador é um "administrador"
		2. Sistema autentica o login do utilizador como "administrador"

Tabela 4.3: Especificação Login

Use Case	Criação da conta de Jogador Base	
Ator	Guest	
Pré-condição	True	
Pós-condição	Nova conta de Jogador Base adicionada ao jogo	
	Ator	**Sistema**
Fluxo Normal	1.Guest define um nome e uma palavra passe	
		2. Sistema verifica disponibilidade do nome
		3. Sistema regista a conta como "jogador base"
Fluxo alternativo (1) [Nome indisponível](passo 2)		1. Sistema informa que o nome não está disponível
		2. Regressa a 1

Tabela 4.4: Especificação Criação da conta de Jogador Base

Use Case	Criação da conta de Jogador Premium	
Ator	Guest	
Pré-condição	True	
Pós-condição	Nova conta de Jogador Premium adicionada ao jogo	
	Ator	**Sistema**
Fluxo Normal	1.Guest define um nome e uma palavra passe	
		2. Sistema verifica disponibilidade do nome
		3. Sistema regista a conta como "jogador premium"
Fluxo alternativo (1) [Nome indisponível](passo 2)		1. Sistema informa que o nome não está disponível
		2. Regressa a 1

Tabela 4.5: Especificação Criação da conta de Jogador Premium

Use Case	Criação de um campeonato	
Ator	Administrador	
Pré-condição	1. Administrador autenticado 2. Existência de circuitos	
Pós-condição	Novo campeonato adicionado ao jogo	
	Ator	**Sistema**
Fluxo Normal	1. Administrador fornece nome do campeonato	
		2. Sistema verifica disponibilidade do nome
		3. Sistema apresenta circuitos existentes
	4. Admin escolhe circuito(s) a adicionar	
		5. Sistema regista o campeonato
Fluxo alternativo (1) [Nome indisponível](passo 2)		1. Sistema informa que o nome não está disponível
		2. Regressa a 1

Tabela 4.6: Especificação da Criação de um campeonato

Use Case	Adição de um campeonato à lista de campeonatos disponíveis	
Ator	Administrador	
Pré-condição	1.Administrador autenticado 2. Existência de campeonatos indisponíveis	
Pós-condição	Novo campeonato disponível aos jogadores	
	Ator	**Sistema**
Fluxo Normal		1. Sistema apresenta campeonatos indisponíveis
	2. Administrador seleciona um campeonato a disponibilizar	
		3. Sistema atualiza campeonato como disponível

Tabela 4.7: Especificação da Adição de um campeonato à lista de campeonatos disponíveis

Use Case	Remoção de um campeonato da lista de campeonatos disponíveis	
Ator	Administrador	
Pré-condição	1.Administrador autenticado 2. Existência de campeonatos disponíveis	
Pós-condição	Novo campeonato indisponível aos jogadores	
	Ator	**Sistema**
Fluxo Normal		1. Sistema apresenta campeonatos disponíveis
	2. Administrador seleciona um campeonato a indisponibilizar	
		3. Sistema atualiza campeonato como indisponível

Tabela 4.8: Especificação da Remoção de um campeonato à lista de campeonatos disponíveis

Use Case	Adição de um circuito	
Ator	Administrador	
Pré-condição	Administrador autenticado	
Pós-condição	Novo circuito adicionado ao jogo	
	Ator	**Sistema**
Fluxo Normal	1. Administrador fornece nome, tamanho em km, nº de curvas, nº de chicanes e nº voltas	
		2. Sistema verifica disponibilidade do nome do circuito
		3. Sistema calcula nº de retas
		4. Sistema apresenta lista de curvas e retas (segmentos da pista)
	5. Administrador fornece GDU's para cada segmento	
		6. Sistema regista novo circuito
Fluxo alternativo (1) [Nome indisponível](passo 2)		1. Sistema informa que o nome não está disponível
		2. Regressa a 1

Tabela 4.9: Especificação da Adição de um circuito

Use Case	Adição de um carro	
Ator	Administrador	
Pré-condição	Administrador autenticado	
Pós-condição	Novo carro adicionado ao jogo	
	Ator	**Sistema**
Fluxo Normal		1. Sistema apresenta as categorias disponíveis
	2. Administrador escolhe categoria, marca, modelo, cilindrada e potência	
		3. Sistema verifica que o carro é C1 e necessita de fiabilidade (e pode ser híbrido)
	4. Administrador indica que carro não é híbrido	
	5. Administrador indica fiabilidade	
		6. Sistema verifica fiabilidade
	7. Administrador indica PAC	
		8. Sistema regista carro e este fica disponível para jogar
Fluxo alternativo (1) [Carro é SC](passo 3)		1. Sistema verifica que o carro é SC
		2. Regressa a 5
Fluxo alternativo (2) [Carro é C2](passo 3)		1. Sistema verifica que o carro é C2
		2. Regressa a 4
Fluxo alternativo (3) [Carro é GT](passo 3)		1. Sistema verifica que o carro é GT
		2. Regressa a 4
Fluxo alternativo (4) [Carro é C2 híbrido](passo 4)	1. Administrador indica que carro é híbrido	
	2. Administrador indica potência do motor elétrico	
		3. Regressa a 5
Fluxo alternativo (5)[Cilindrada não se enquadra na categoria](passo 2)	1. Sistema verifica que a cilindrada não se enquadra na categoria.	
		2. Regressa a 4

Tabela 4.10: Especificação da Adição de um carro

Use Case	Adição de um piloto	
Ator	Administrador	
Pré-condição	Administrador autenticado	
Pós-condição	Novo piloto adicionado ao jogo	
	Ator	**Sistema**
Fluxo Normal	1. Administrador indica o nome do piloto	
		2. Sistema verifica disponibilidade do nome do piloto
	3. Administrador indica os níveis de perícia nos critérios de CTS ("Chuva vs. Tempo Seco") e SVA ("Segurança vs Agressividade")	
		4. Sistema verifica que os dados de níveis de perícia estão válidos
		5. Sistema regista novo piloto
Fluxo alternativo (1) [Nome indisponível](passo 2)		1. Sistema informa que o nome não está disponível
		2. Regressa a 1
Fluxo alternativo (2) [Níveis de perícia inválidos](passo 4)		1. Sistema informa que os dados são inválidos
		2. Regressa a 3

Tabela 4.11: Especificação da Adição de um piloto

Use Case	Configuração de um campeonato	
Ator	Jogador	
Pré-condição	True	
Pós-condição	Jogador registado	
	Ator	**Sistema**
Fluxo Normal	1. Jogador seleciona campeonato	
	2. Jogador seleciona carro e piloto pretendido	
		3. Sistema verifica dados
		4. Sistema regista o jogador no campeonato
		5. Sistema regista novo piloto
	6. Jogador decide começar campeonato	
Fluxo alternativo (1) [Adicionar outro jogador](passo 5)	1. Jogador escolhe adicionar outro jogador	
		2. Regressa a 2

Tabela 4.12: Especificação da Configuração de um campeonato

Use Case	Preparação para a corrida	
Ator	Jogador	
Pré-condição	Campeonato configurado	
Pós-condição	Jogador registado como pronto para corrida	
	Ator	**Sistema**
Fluxo Normal	1. Jogador escolhe fazer afinações ao seu carro	
	2. Jogador faz afinações	
	3. Jogador escolhe o tipo de pneu que pretende e se motor é híbrido ou não	
		4. Sistema regista jogador como pronto
Fluxo alternativo (1) [Não faz afinações](passo 1)	1. Jogador escolhe não fazer afinações ao seu carro	
		2. Regressa para 3
Fluxo alternativo (2) [Escolha de motor](passo 3):	1. Jogador não consegue escolher tipo de motor se o carro for de categoria SC	
		2. Regressa para 4

Tabela 4.13: Especificação da Preparação para a corrida

Use Case	Jogo simula a corrida	
Ator	Jogador	
Pré-condição	1. Corrida com todos os jogadores prontos 2. Jogador registado no sistema como "jogador base"ou não ter efetuado login	
Pós-condição	Corrida simulada	
	Ator	**Sistema**
Fluxo Normal		1. Sistema inicia a simulação da corrida
		2. Após cada segmento (curva/reta/chicane) o sistema atualiza situações do mesmo
		3. Sistema indica posições após cada volta
		4. Sistema apresenta resultados no fim da corrida
		5. Sistema atribui pontos a cada jogador, por posições e categorias somando para corridas futuras no mesmo campeonato
		6. Sistema simula próximas corridas até serem feitas todas as existentes do campeonato
		7. Sistema verifica que jogador não está autenticado
	8. Jogador escolhe fazer login	
		9. Sistema contabiliza os pontos obtidos no ranking da sua conta
		10. Sistema mostra os rankings
Fluxo alternativo (1) **[Jogador já está autenticado](passo 7)**		1. Sistema verifica que jogador já está autenticado
		2. Regressa a 9
Fluxo alternativo (2) **[Jogador não faz login](passo 8)**	1. Jogador não faz login	
		2. Regressa a 10

Tabela 4.14: Especificação da Simulação da Corrida Base

Use Case	Simulação da Corrida Premium	
Ator	Jogador	
Pré-condição	1. Corrida com todos os jogadores prontos 2. Jogador registado no sistema como "jogador premium"	
Pós-condição	Corrida simulada	
	Ator	**Sistema**
Fluxo Normal		1. Sistema inicia a simulação da corrida
		2. Sistema atualiza as situações do mesmo em tempo real
		3. Sistema indica posições em tempo real
		4. Sistema apresenta resultados no fim da corrida
		5. Sistema atribui pontos a cada jogador, por posições e categorias somando para corridas futuras no mesmo campeonato
		6. Sistema simula próximas corridas até serem feitas todas as existentes do campeonato
		7.Sistema contabiliza os pontos obtidos no ranking da sua conta
		8. Sistema mostra os rankings

Tabela 4.15: Especificação da Simulação da Corrida Premium

5 Diagrama de Componentes

Os diagramas de componentes têm como objetivo visualizar a organização dos componentes do sistema e os relacionamentos de dependência entre eles. Neste caso usamos o diagrama de componentes com o principal objetivo de dividir a lógica de negócio do nosso projeto, dividindo as responsabiliades do sistema, identificadas anteriormente, em 5 subsistemas distintos. Cada subsistema contêm o seu respetivo diagrama de classes com os seus métodos, identificados nas responsabilidades do sistema, e as suas classes. Cada responsabilidade do sistema tem o seu respetivo diagrama de sequência. Uma vez que dividimos em 5 subsistemas no diagrama de componentes, não há um grande nível de complexidade na recolha e definição de informação pelos diferentes métodos. O diagrama de componenetes com os seus diagramas de classes e de sequência serão enviados em anexo.

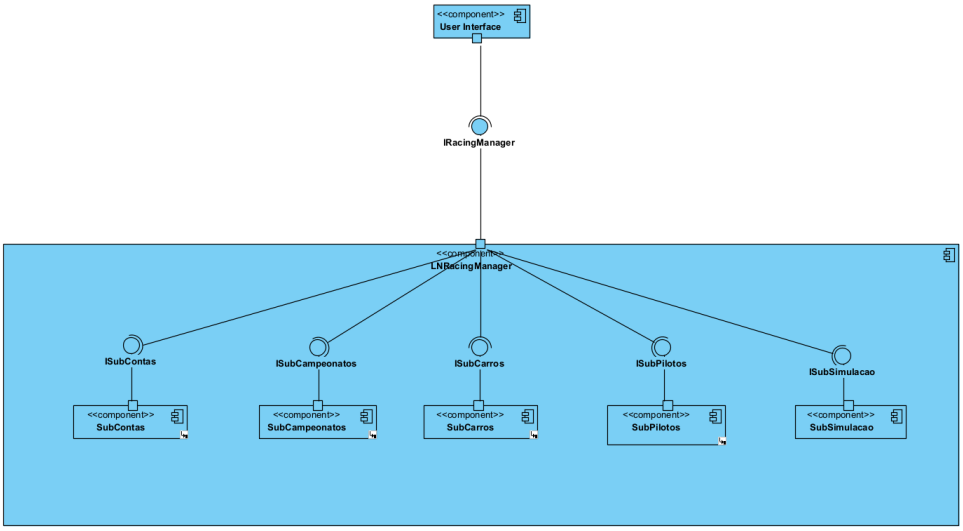


Figura 5.1: Diagrama de Componentes

Nota: Decidimos não fazer um diagrama de packages pois achamos que não acrescentaria informação ao nosso trabalho, tendo isto em conta simplesmente assumimos que cada subsistema, representado no diagrama de componentes, corresponde a um package.

5.1 Identificação das responsabilidades do sistema

Através dos use cases, realizados na primeira fase deste trabalho, identificamos as responsabilidades do sistema e associamos cada responsabilidade a um respetivo subsistema, identificando, assim, 5 subsistemas distintos em que cada subsistema tem as suas respetivas responsabilidades: SubContas, SubCampeonatos, SubCarros, SubPilotos, SubSimulacao.

As identificações das responsabilidades do sistema podem ser visualizadas no excel enviado em anexo.

5.2 Subsistemas

5.2.1 SubContas

O subsistema de Contas é fundamental no projeto, pois guardar os dados de um Utilizador, o seu nome, a sua password, o seu tipo (Administrador,JogadorPremium,JogadorBase) e os seus pontos.

Diagrama de Classes

Este sistema, para além da sua própria interface *ISubContas* e o seu *facade SubContasFacade*, constitui uma classe *Conta*, sendo os seus atributos a informação que pretendemos guardar do utilizador já referida, e uma *enumeration* para representar o tipo de utilizador, também já referido. A classe *Conta* é implementada pelo *facade* com uma estrutura de dados *HashMap(String,Conta)* em que a chave é o nome presente na respetiva *Conta*. Deste modo encontra-se, na figura seguinte, o diagrama de classes desta parte do projeto, contendo os seguintes métodos referentes às responsabilidades do sistema:

- nomeDisponível(nome : String) : Boolean
- registarConta(nome : String, password : String, type : userType)
- validarConta(nome : String, password : String) : Boolean
- autenticarConta(nome : String) : userType
- atribuirPontos(nome : String, pontos : Integer)

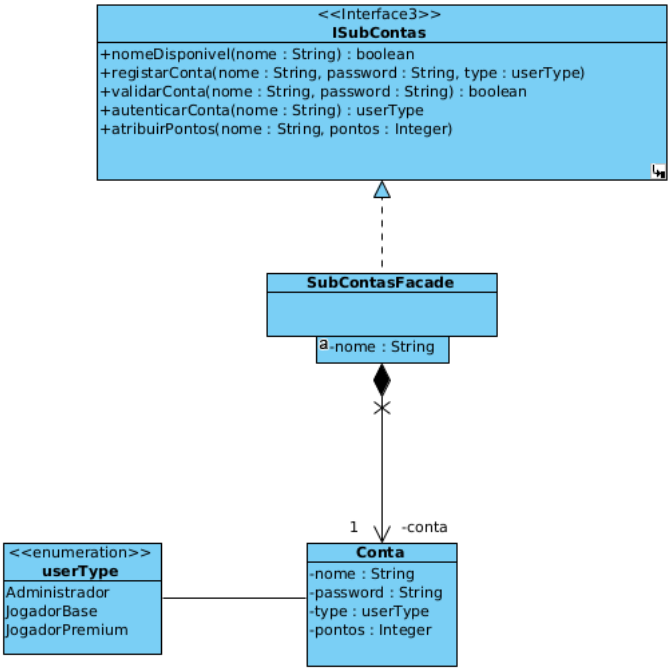


Figura 5.2: Diagrama de Classes do SubContas

Diagrama de Sequência

Após realizado o diagrama de classes para o subsistema, o próximo passo são os diagramas de sequência referentes aos métodos definidos nas classes.
Em primeiro lugar, temos o método "nomeDisponivel"que recebe um nome(*String*) e percorre o *HashMap* das contas. É retornado um *booleano* dependendo se o nome já existe no *Map*(*true*) ou, se ainda se encontra disponivel(*false*).

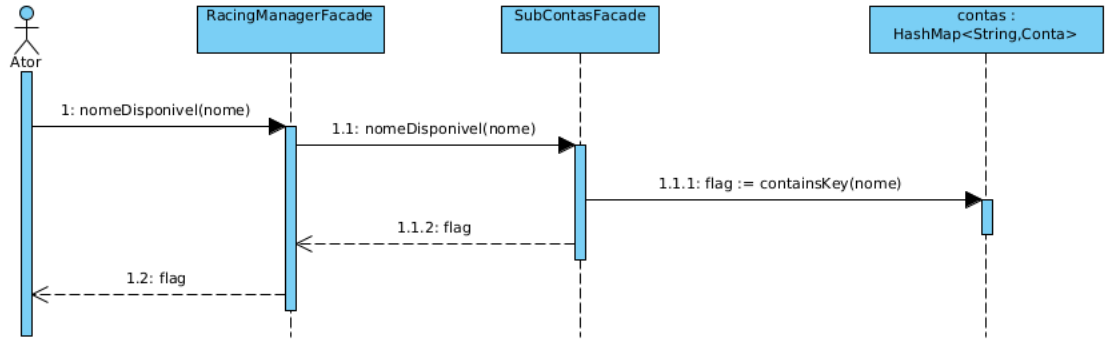


Figura 5.3: Diagrama de Sequência do nomeDisponivel

De seguida, a função "registarConta"tem como argumentos um nome, uma *password* e um *enum(type)*. Com estes dados, usa o construtor da Conta e posteriormente adiciona essa mesma conta ao *HashMap*.

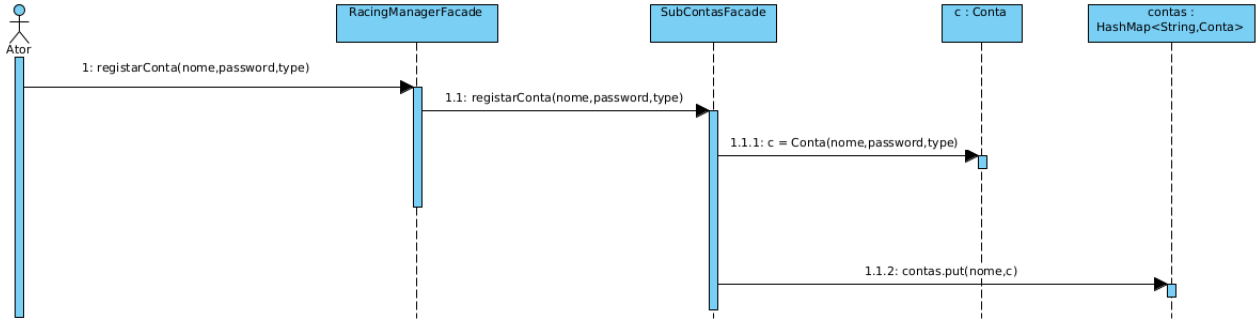


Figura 5.4: Diagrama de Sequência do registarConta

Adicionalmente, outro método presente neste subsistema é o "validarConta"que tem como finalidade validar os dados (nome e *password*) fornecidos por um utilizador referentes a uma conta. Primeiramente, obtém-se a conta associada ao nome dado pelo utilizador. Se essa conta existir, comparamos a password fornecida com a que se encontra na informação da conta. Por fim, retorna-se um *booleano* com o valor *true* se os dados são válidos.

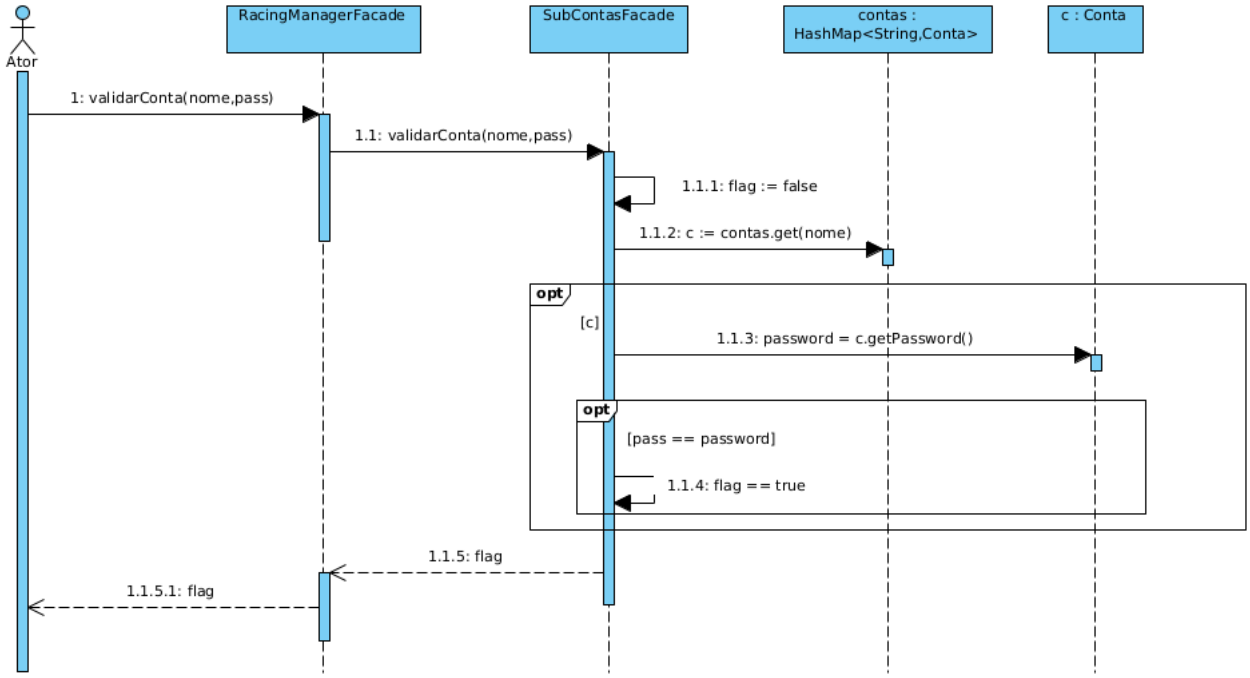


Figura 5.5: Diagrama de Sequência do validarConta

Doutro modo, o "autenticarConta", apenas recebe um nome que é utilizado para conseguirmos saber as informações relativas à conta associada ao mesmo, e retorna um *enum* com o seu tipo(*type*).

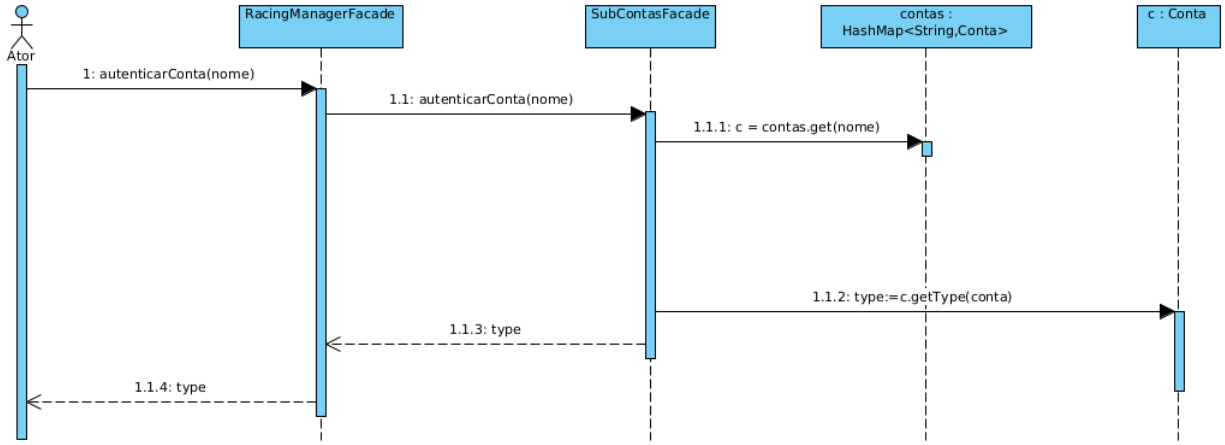


Figura 5.6: Diagrama de Sequência do autenticarConta

Por fim, a função de "atribuirPontos" recebe dois argumentos(nome e pontos) e tem como objetivo adicionar pontos a uma certa conta. Através do nome da conta acrescenta-se ao número de pontos já existentes, os pontos dados como argumento.

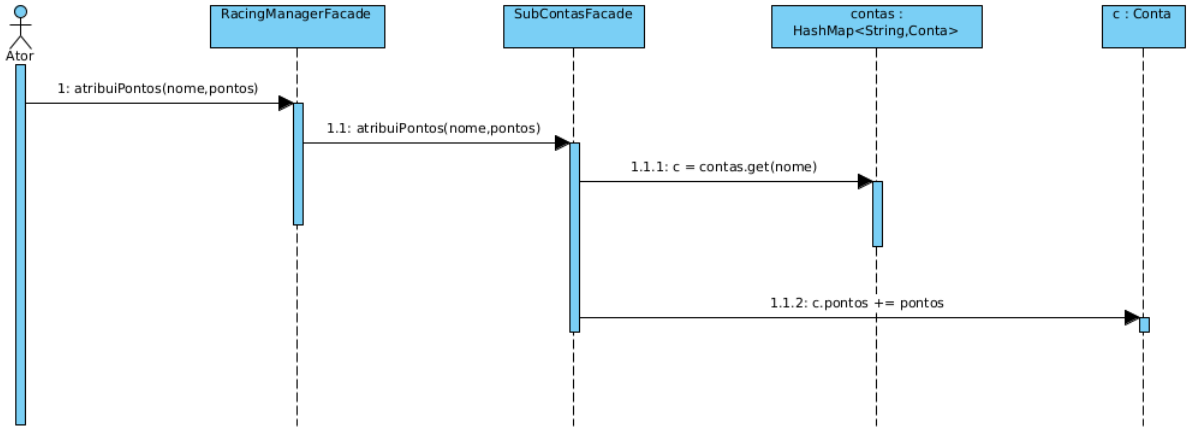


Figura 5.7: Diagrama de Sequência do atribuirPontos

5.2.2 SubCampeonatos

O subsistema de Campeonatos conta com os módulos de *Campeonato*, *Circuito* e *Segmento*, tendo, assim, como principal função suportar a funcionalidade de criação de campeonato e circuito, mas também em manipular a lista de campeatos que estão disponiveis para serem jogados.

Diagrama de Classes

Como podemos visualizar na figura abaixo, do diagrama de classes, temos uma *interface* titulada de *ISubCampeonatos*, um *facade* definido como *SubCampeonatosFacade*, as

respetivas classes mencionadas acima, uma *enumeration* para o tempo metereológico do circuito e as seguintes estruturas de dados:

- -campeonatos: HashMap<String,Campeonato> (em que a chave é o nome do respetivo Campeonato)
- -campeonatosDisponíveis: List<Campeonato>
- -campeonatosIndisponíveis: List<Campeonato>
- -circuitos: HashMap<String,Circuito> (em que a chave é o nome do respetivo Campeonato)
- -segmentos: List<Segmentos> (Ligada a cada Circuito)

Os métodos definidos são:

- nomeCampeonatoDisponível(nome : String) : Boolean
- nomeCircuitoDisponivel(nome: String) : Boolean
- registarCircuito(circuito: Circuito)
- circuitosExistentes() : List<Circuito>
- registarCampeonato(campeonato : Campeonato)
- campeonatosIndisponiveis() : List<Cameponato>
- disponibilizarCampeonato(campeonato : Cameponato)
- campeonatosDisponiveis() : List<Campeonato>
- indisponibilizarCampeonato(campeonato : Campeonato)
- calcularRetas(nrCurvas : Integer, nrChicanes : Integer) : Integer

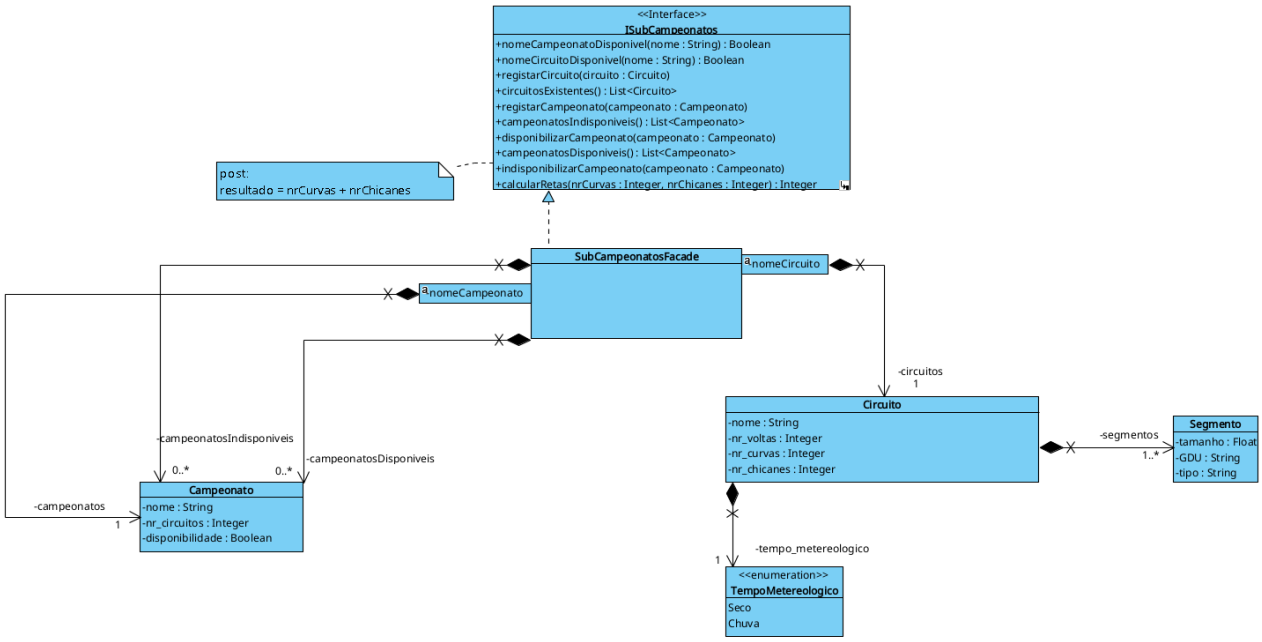


Figura 5.8: Diagrama de Classes SubCampeonatos

Diagrama de Sequências

Abaixo é visível, primeiramente, o diagrama de sequências do método "nomeCampeonatoDisponivel"onde o administrador indica determinado nome pretendido para o campeonato e o *facade* verifica se na variável de instância de estrutura *HashMap* possui alguma chave desse mesmo valor, retornando um *booleano*.

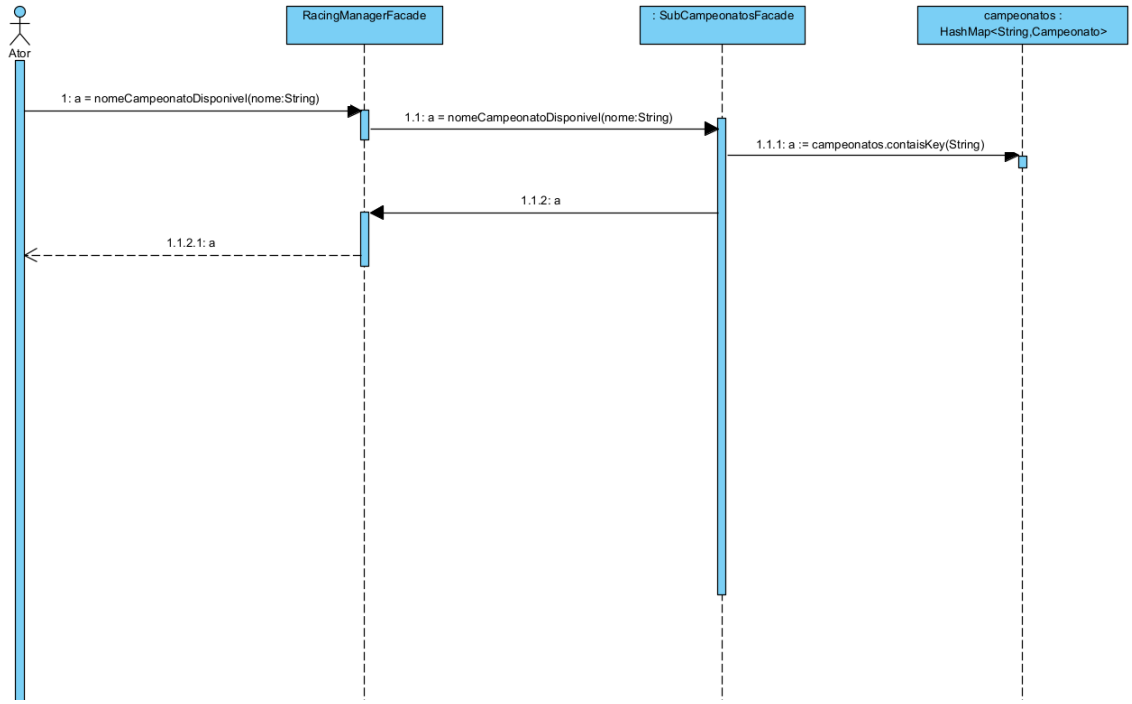


Figura 5.9: Diagrama de Sequências nomeCampeonatoDisponivel

Seguidamente, na função "campeonatosDisponiveis"podemos verificar o processo de iterar pelos valores guardados no *HashMap* e verificar quais deles tem a variável disponibilidade é falsa. Os que forem sinalizados por esse estado, iram ser removidos dessa lista de valores, sendo que posteriormente será devolvido essa lista.

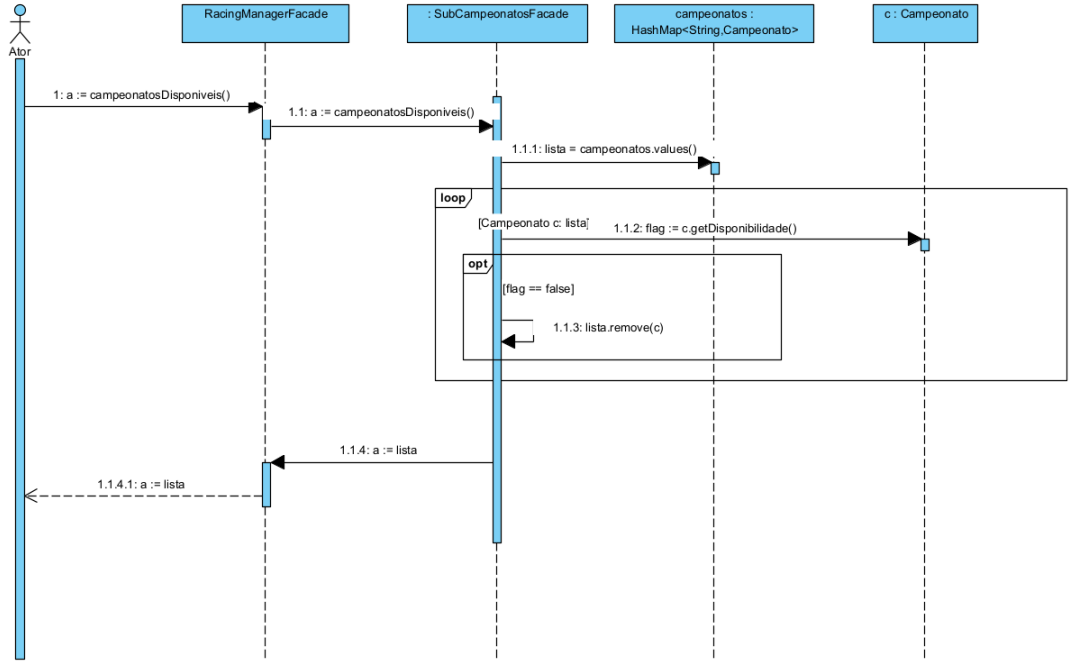


Figura 5.10: Diagrama de Sequências campeonatosDisponiveis

Já no diagrama de sequências de "registarCampeonato", é visível que recebendo um objeto *Campeonato*, faz o seu clone e insere no *HashMap* existente como variável de instância no *facade*.

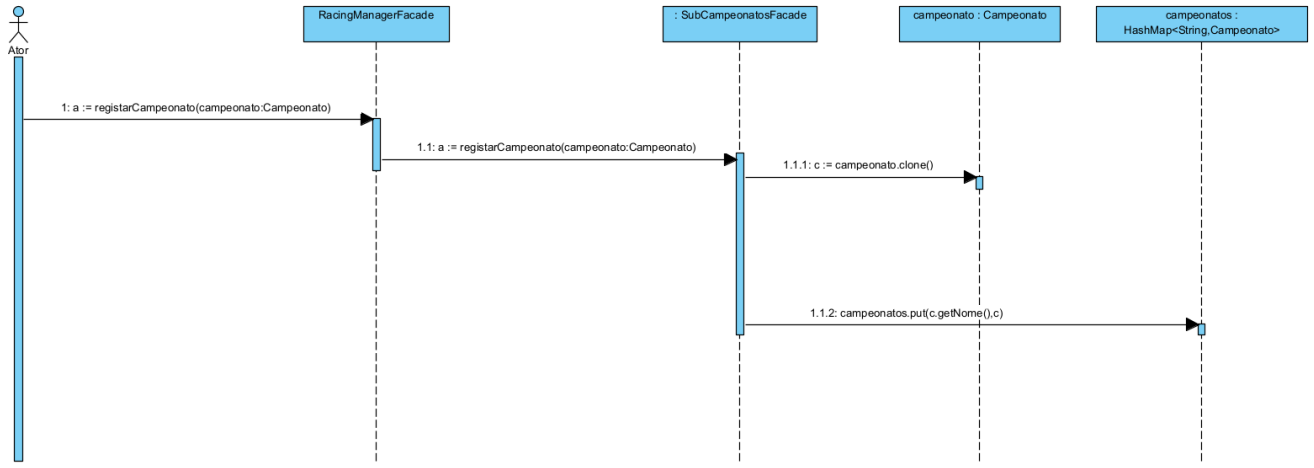


Figura 5.11: Diagrama de Sequências registarCampeonato

Em "disponibilizarCampeonato"verifica-se que, recebendo determinado objeto *Campeonato*, remove esse mesmo objeto da lista de campeonatos indisponiveis. Seguidamente troca a sua variável disponibilidade para *true* e adiciona à lista de campeonatos disponiveis, que é outra variável do *facade*.

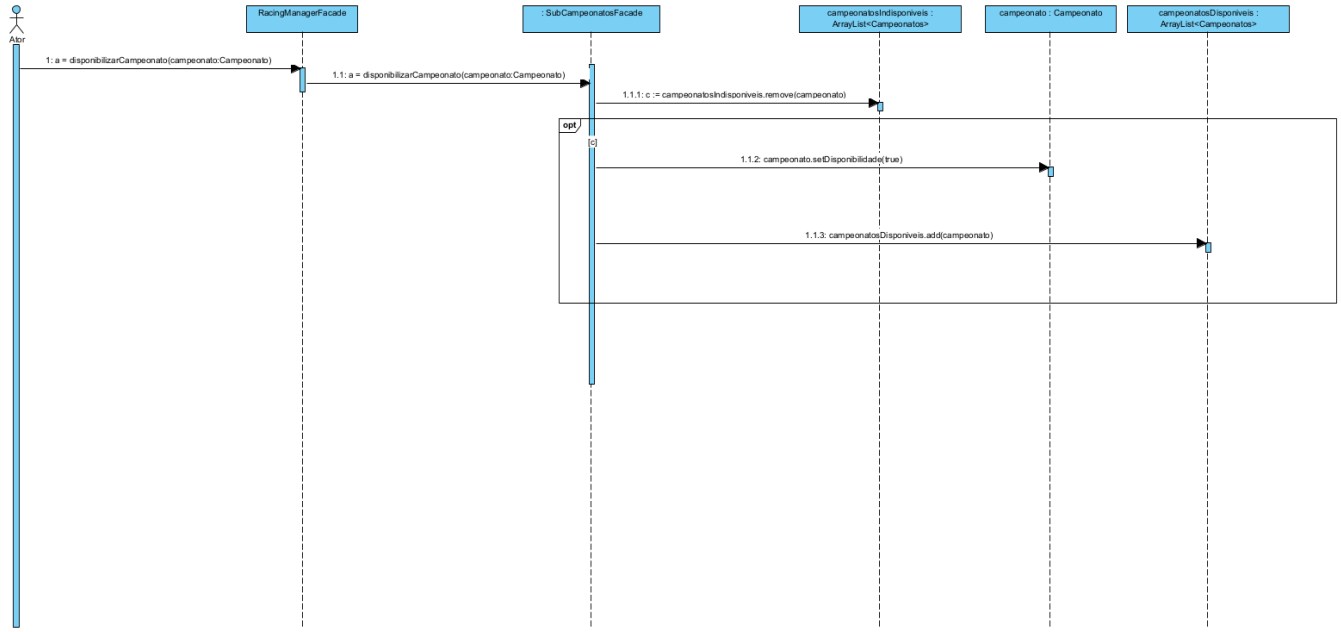


Figura 5.12: Diagrama de Sequências disponibilizarCampeonato

Por fim "circuitosExistentes"é similar ao "campeonatosDisponiveis"e "campeonatosIndiponiveis"onde se obtem uma lista de valores guardados no *Map* de circuitos, definido pela chave ser o nome do circuito e o valor ser o objeto de determinado circuito com esse nome.

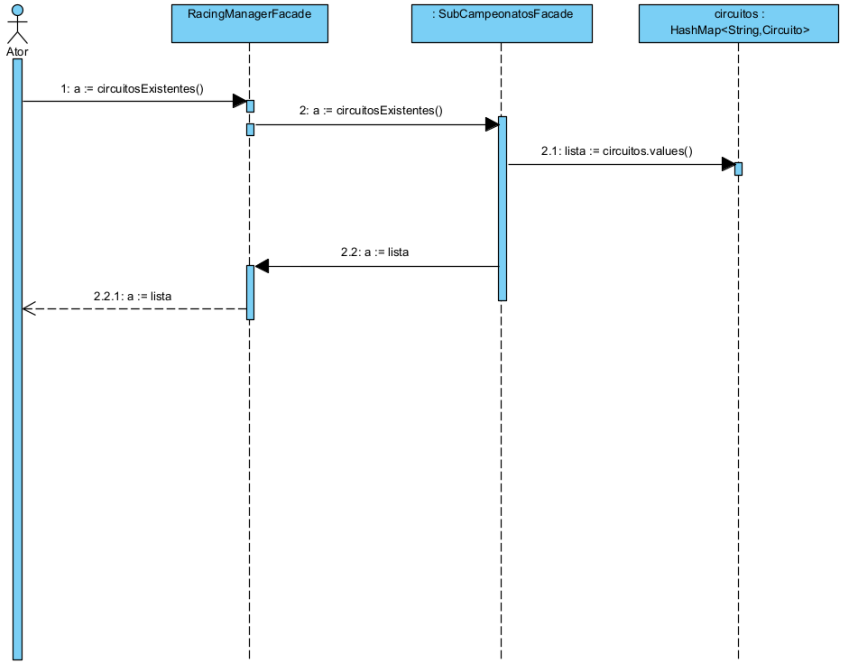


Figura 5.13: Diagrama de Sequências circuitosExistentes

5.2.3 SubCarros

Mesmo sendo os carros um dos pontos principais do programa, acaba por ser um componente com uma grande relevância a nível independente, tendo, assim, o seu próprio subsistema, onde se encontra todas as responsabilidades do sistema referentes a criação do carro.

Diagrama de Classes

Este Diagrama, para além das responsabilidades do sistema referente à criação do carro, também contém mais 9 classes (*Carro,C1,C2,GT,SC,C1H,C2H,GTH*), a *interface Híbrido* e uma *HashMap(String,Carro)*. Os métodos definidos são:

- categoriaValida(c : String) : Boolean
- fiabilidadeValida(f : int) : Boolean
- registarCarro(carro : Carro)
- cilindradaValida(cilindrada : int, categoria : String) : Boolean

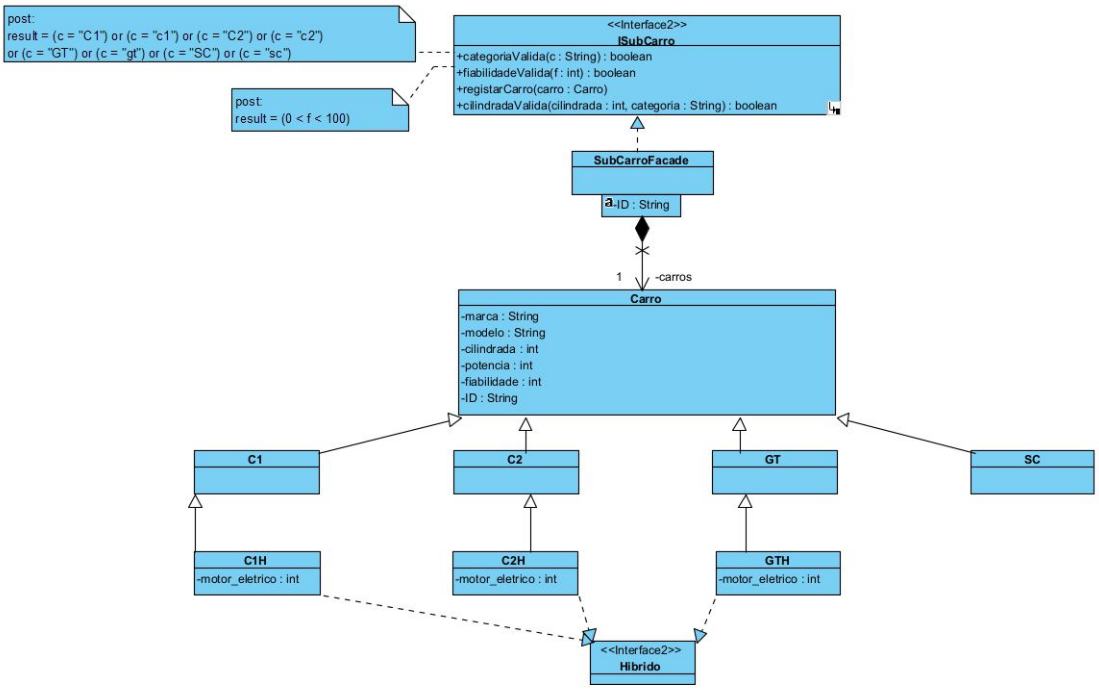


Figura 5.14: Diagrama de Classes do SubCarros

Diagrama de Sequência

O método "categoriaValida"por se tratar dum método simples que vê se a *String* recebida corresponde a uma das *Strings* correspondentes às categorias do nosso programa decidimos fazer em OCL em vez dum diagrama de sequências, onde o resultado do método é *true* se a *String* for igual a uma das *Strings* referidas na figura e *false* caso contrário.

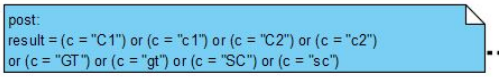


Figura 5.15: Diagrama de Sequências categoriaValida

O método "fiabilidadeValida"também foi feito em OCL, pela mesma razão de ser simples, em que o resultado do método é *true* se o número dado estiver entre 0 e 1 e *false* caso contrário.

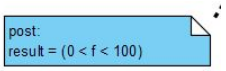


Figura 5.16: Diagrama de Sequências fiabilidadeValida

O método "registarCarro"simplesmente recebe um objeto do tipo *Carro* e adiciona-o ao *HashMap* carros, sendo a sua chave o tamanho da *HashMap*.

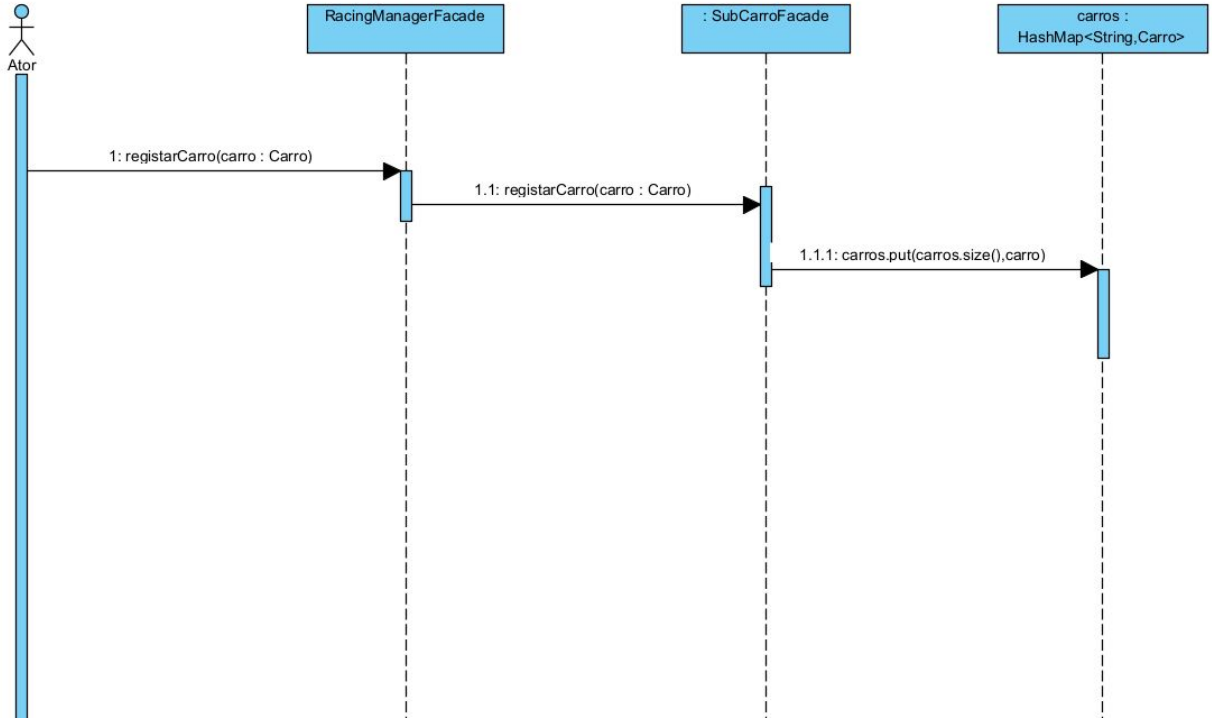


Figura 5.17: Diagrama de Sequências registarCarro

O método "cilindradaValida"verifica se a cilindrada pretendida se enquadra na categoria pretendida devolvendo *true* caso seja e *false* caso contrário.

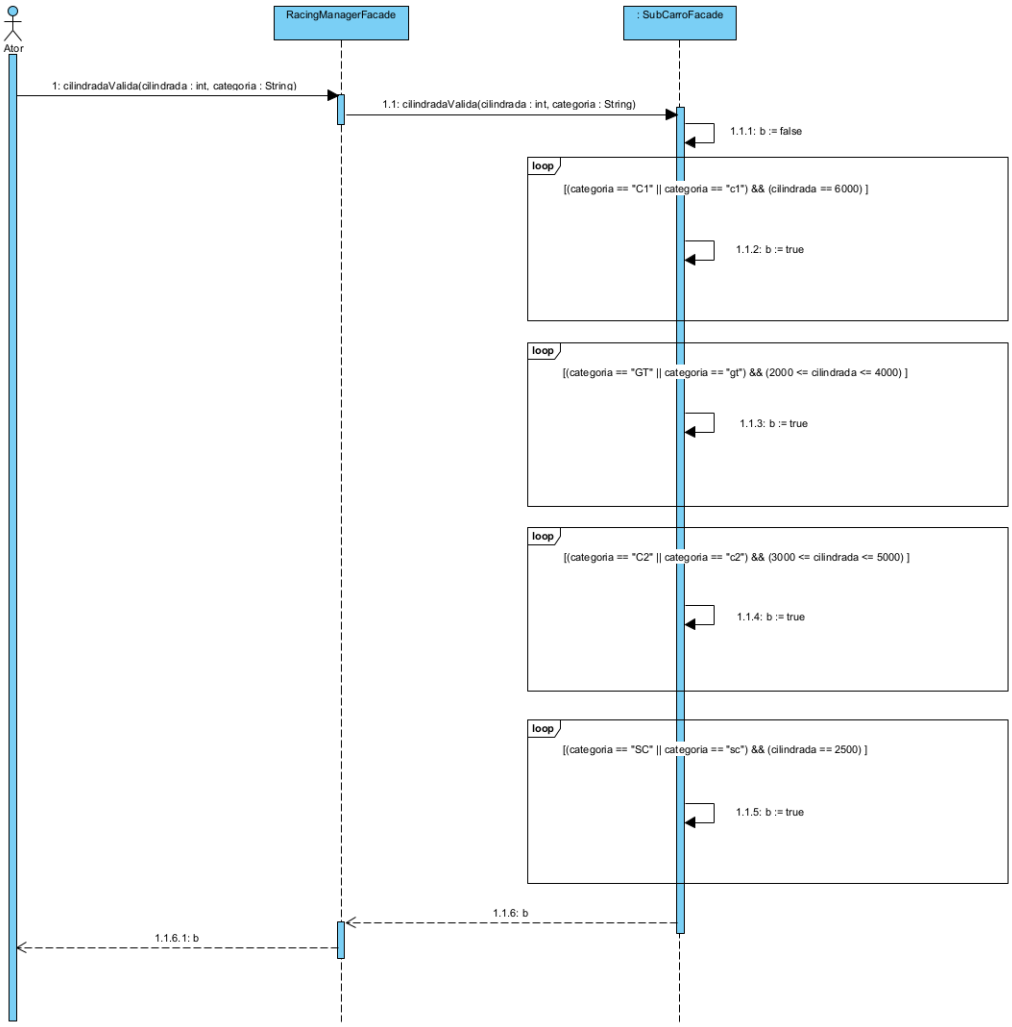


Figura 5.18: Diagrama de Sequências cilindradaValida

5.2.4 SubPilotos

Não sendo o ponto central do programa, os pilotos representam uma parte fundamental da arquitetura do mesmo, uma vez que este se encontra num contexto de corridas motorizadas. Assim sendo, toda a responsabilidades de criação de um piloto e inserção na sua respetiva estrutura de dados encontra-se no quadro das responsabilidades do subsistema dos Pilotos.

Diagrama de Classes

Este sistema é composto por três classes simples. A Interface *ISubPiloto* implementa métodos de validação dos atributos e inserção do *Piloto* que se pretende registar no sistema. Esta, por sua vez, é implementada pela classe *SubPilotoFacade*, que mantém a estrutura de dados - uma *HashMap(String,Piloto)* que para cada nome associa, por

composição, um *Piloto*. O *Piloto* trata-se de uma classe cujos atributos registam o seu nome, CTS e SVA. Os métodos definidos são:

- nomePilotoDisponivel(nome : String) : Boolean
- niveisPericiaValidos(cts : float, sva : float) : Boolean
- registarPiloto(piloto : Piloto)

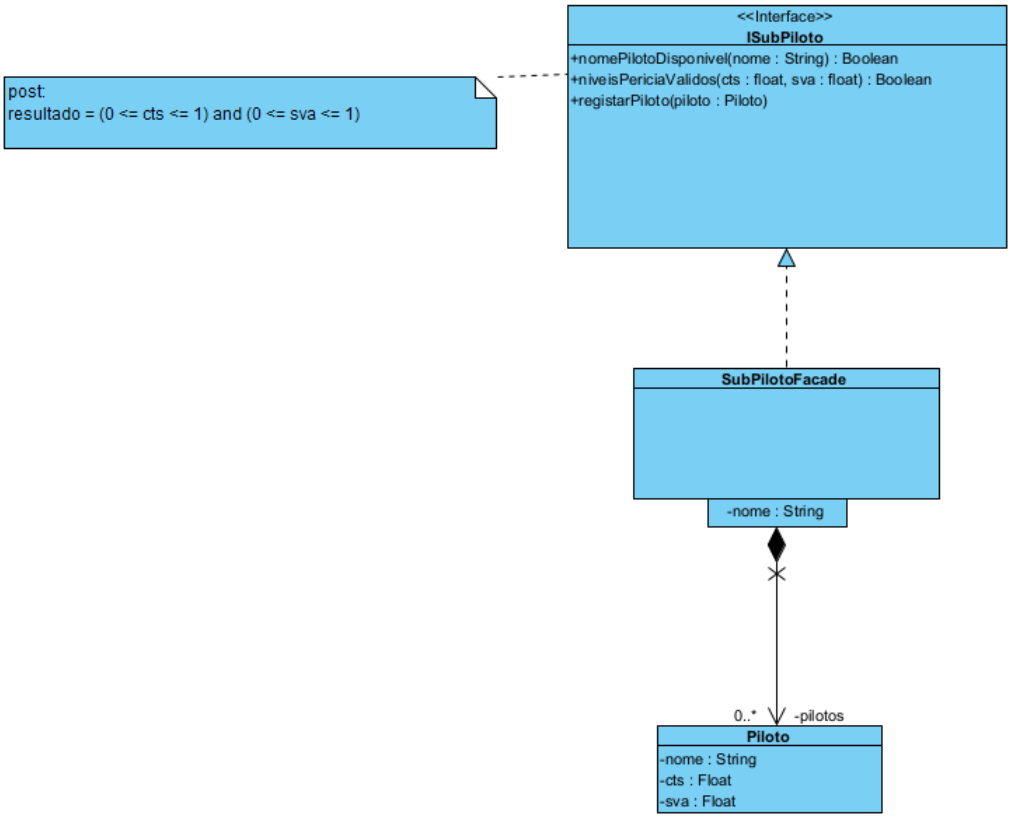


Figura 5.19: Diagrama de classes do subsistema dos pilotos

Diagramas de Sequência

Dado o grau de simplicidade do método "niveisPericiaValidos", que verifica apenas se os valores se encontram dentro dos respetivos limites, optamos por indicar apenas a pós-condição em OCL. Para os restantes métodos desenvolvemos os seus diagramas

de sequência, representando as interações entre o utilizador e o programa e entre os componentes do programa, como se verificam nas figuras 5.20 e 5.21.

O método "nomePilotoDisponivel"verifica se na *HashMap* pertencente ao *SubPilotosFacade* existe alguma chave associada ao valor dado, retornando um *booleano*.

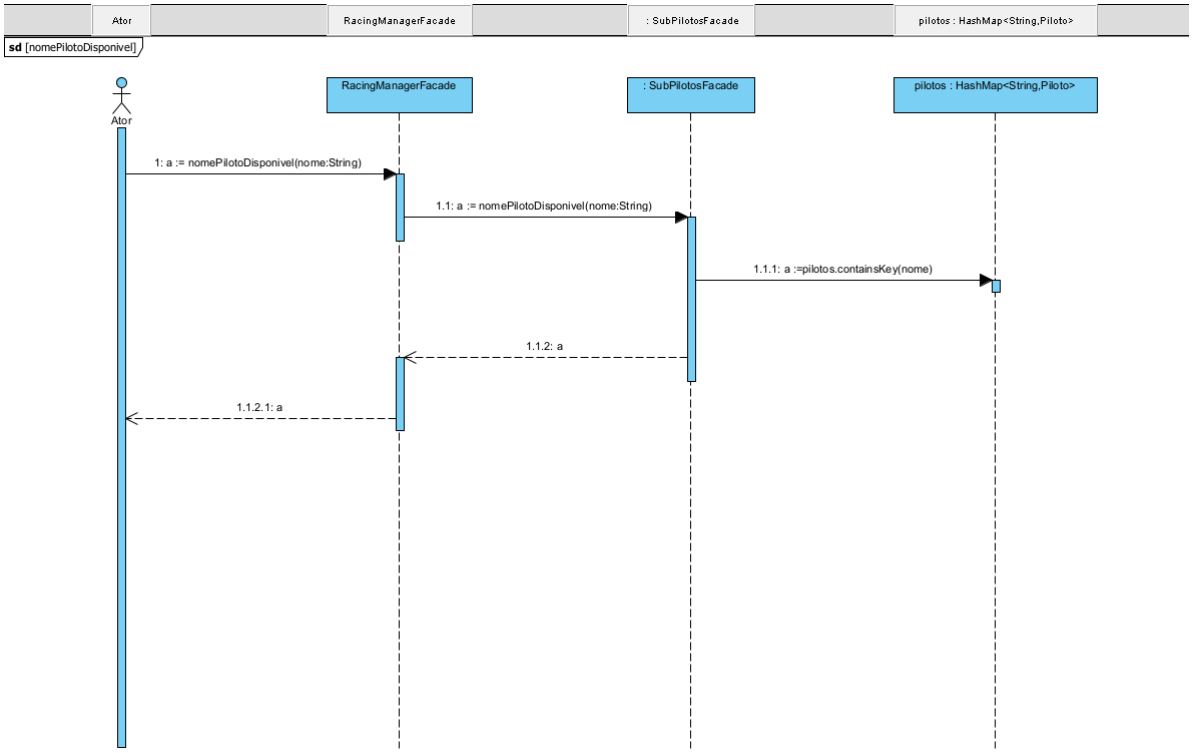


Figura 5.20: Diagrama de sequência do método nomePilotoDisponivel

O método "registarPiloto"recebe um objeto do tipo *Piloto* e adiciona-o à *HashMap* pilotos, sendo a sua chave o nome do *Piloto* recebido.

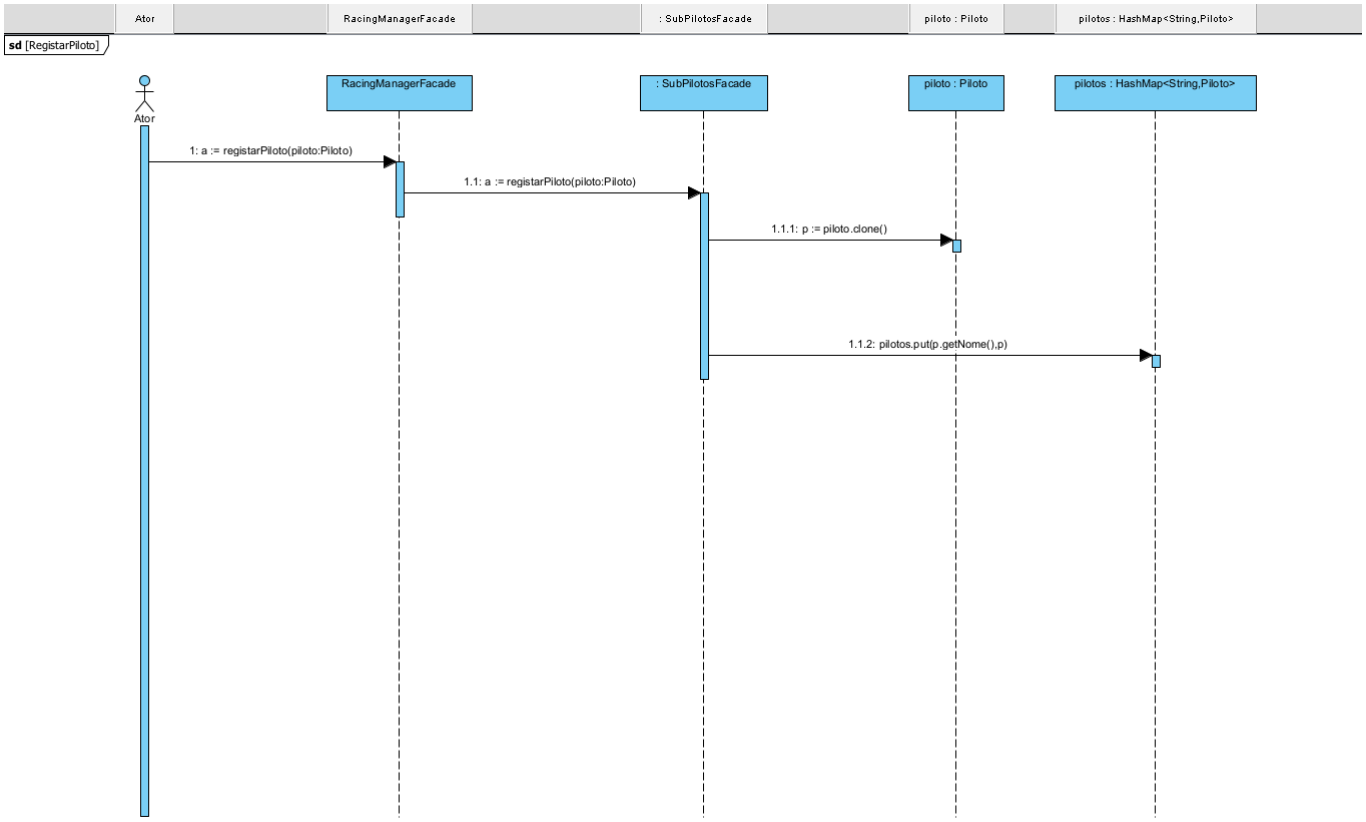


Figura 5.21: Diagrama de sequência do método registarPiloto

5.2.5 SubSimulacao

O subcomponente SubSimulação modela o comportamento do programa relacionado com a simulação do jogo.

Diagrama de Classes

Classes usadas para implementar SubSimulação:

- *ISubSimulaçao* interface que representa o API do componente da Simulação
- *SubSimulaçao* implementa o API do componente da Simulação
- *CampeonatoAtivo* representa um campeonato pronto para começar e simular corridas.
- *JogadorAtivo* representa um jogador participante num campeonato.
- *DadosJogador* associa o id de um jogador a um *Carro* e a um *Piloto*.
- *Corrida* modela funcionalidades comuns a *CorridaBase* e *CorridaPremium*

- *CorridaBase* guarda o conjunto de estados gerados pela simulação Base de um jogo
- *CorridaPremium* guarda o conjunto de estados gerados pela simulação Premium de um jogo
- *EstadoBase* representa o estado da simulação Base num dado momento.
- *EstadoPremium* representa o estado da simulação Premium num dado momento.
- *EstadoJogador* guarda a posição e o estado do carro de um jogador.
- *EstadoJogadorPremium* extende *EstadoJogador*, guardando também o tempo total.

Métodos de SubSimulação:

- registarJogador(campeonato : Campeonato, jogadorID : String, carro : Carro, piloto : Piloto)
- jogadorPronto(campeonato : Campeonato, jogadorID : String)
- simularCorridaBase(campeonato : Campeonato) : CorridaBase
- simularCorridaPremium(campeonato : Campeonato) : CorridaPremium
- ranking(campeonato : Campeonato) : List<DadosJogador>
- afinarCarro(campeonato : Campeonato, jogadorID : String, func : Consumer¡Carro¿)
- temProxCorrida(campeonato : Campeonato) : boolean

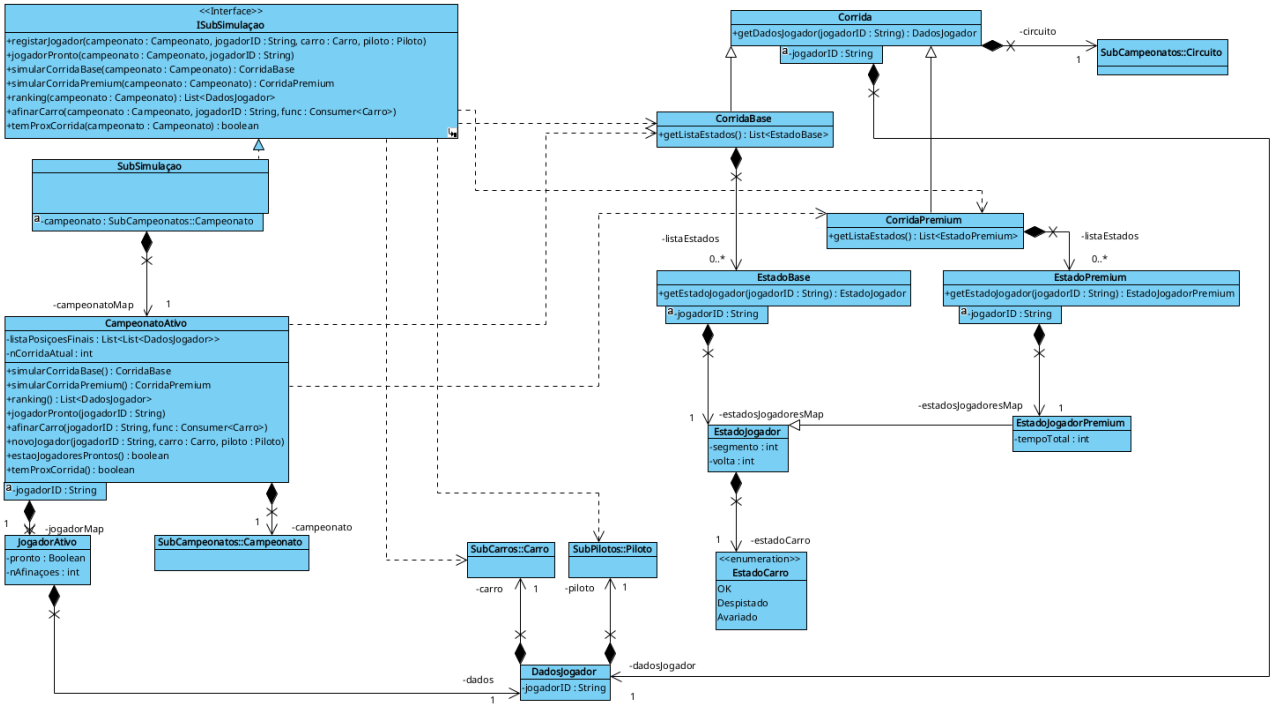


Figura 5.22: Diagrama de classes SubSimulaçao

Diagrama de Sequência

O método "registarJogador" regista o ID de um jogador, o seu *Piloto* e o seu *Carro* escolhido no *CampeonatoAtivo* que representa o *Campeonato* recebido como argumento. Se não existir um *CampeonatoAtivo* que o represente, um novo será criado.

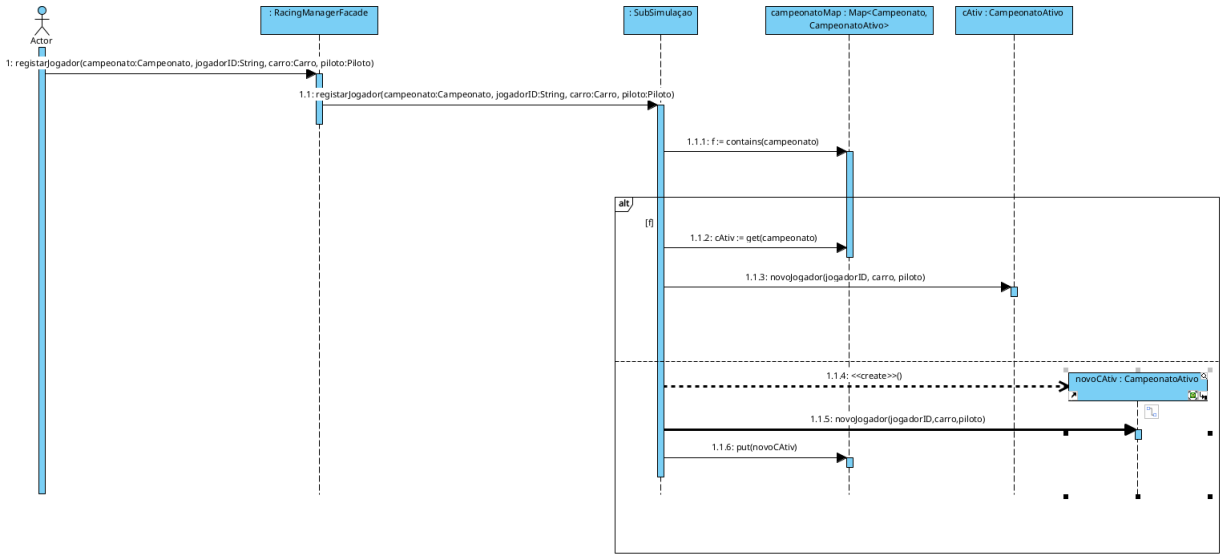


Figura 5.23: Diagrama de sequência do método registarJogador

O método "jogadorPronto"regista um jogador como pronto para começar a próxima corrida no *CampeonatoAtivo* que representa o *Campeonato* recebido como argumento. É levantado um erro caso o *CampeonatoAtivo* equivalente não exista ou caso o jogador não estiver registado no *CampeonatoAtivo*.

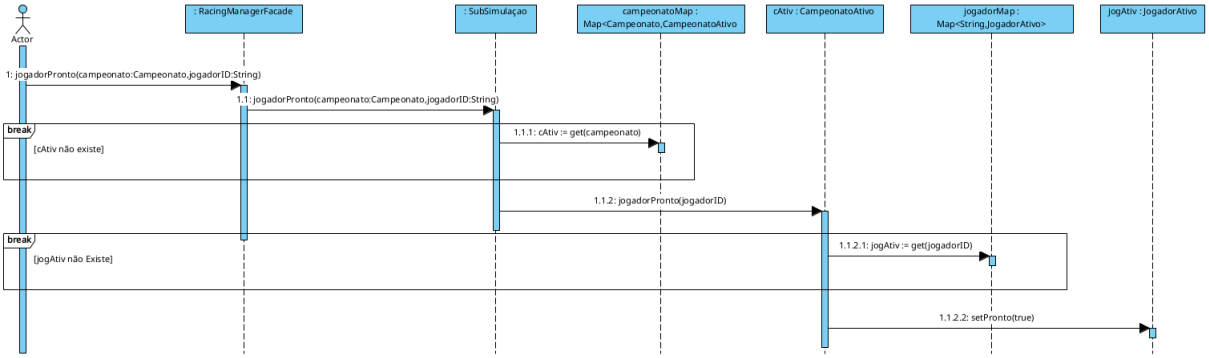


Figura 5.24: Diagrama de sequência do método jogadorPronto

O método "simularCorridaBase"simula usando as regras da simulação base a próxima corrida do *Campeonato* recebido como argumento, e devolve uma *CorridaBase* com a informação do que ocorreu na corrida. É levantado um erro caso o *CampeonatoAtivo* representante não exista, caso algum jogador não esteja pronto, ou caso o *CampeonatoAtivo* não tenha um próximo *Circuito* para começar a simulação.

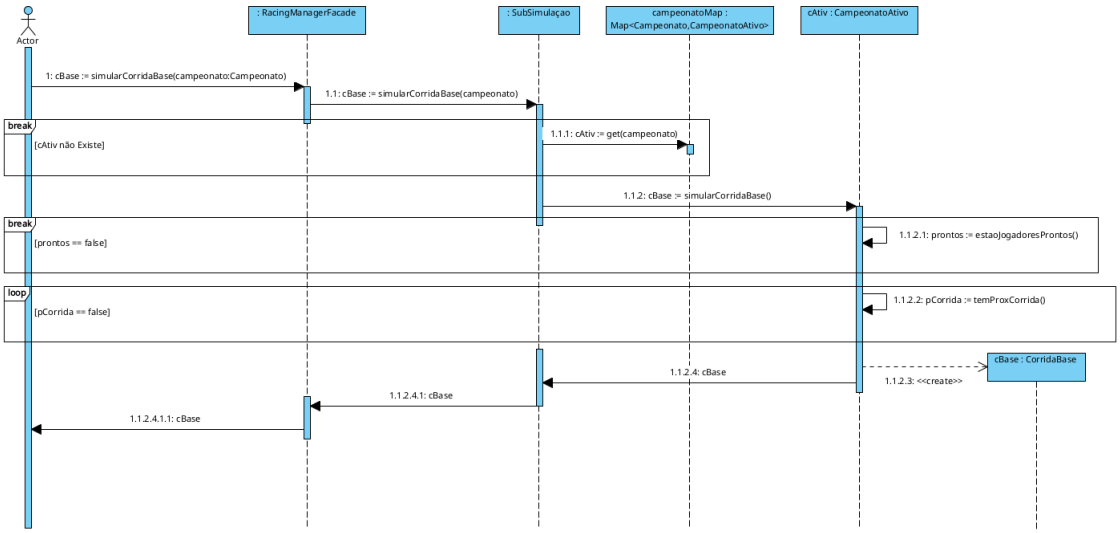


Figura 5.25: Diagrama de sequência do método simularCorridaBase

O método "simularCorridaPremium"simula usando as regras da simulação premium a próxima corrida do *Campeonato* recebido como argumento, e devolve uma *CorridaPremium* com a informação do que ocorreu na corrida. É levantado um erro caso o

CampeonatoAtivo representante não exista, caso algum jogador não esteja pronto, ou caso o *CampeonatoAtivo* não tenha um próximo *Circuito* para começar a simulação.

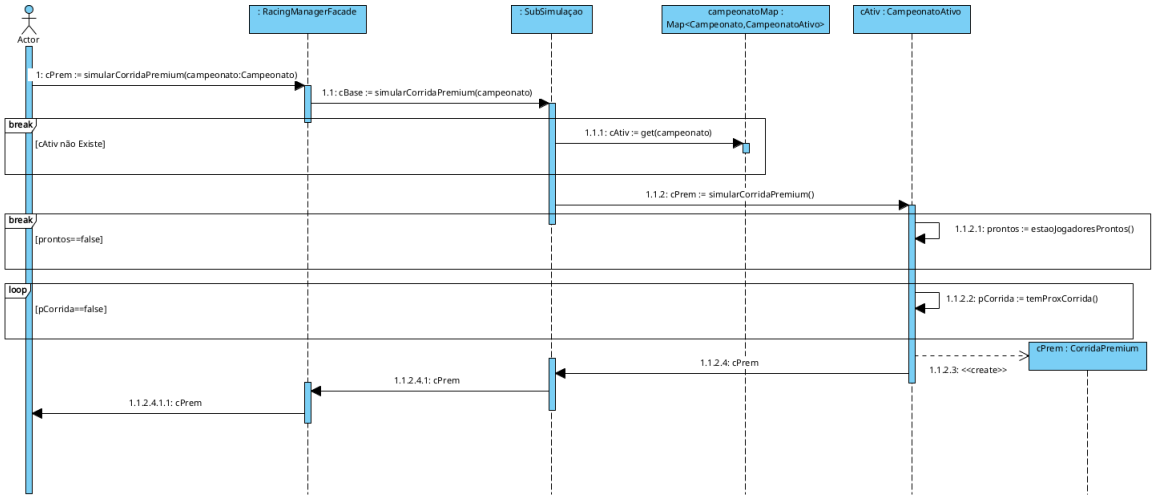


Figura 5.26: Diagrama de sequência do método simularCorridaPremium

O método "ranking" devolve uma lista de *DadosJogador* ordenados pela posição atual no campeonato fornecido como argumento. É levantado um erro caso o *CampeonatoAtivo* representante não exista.

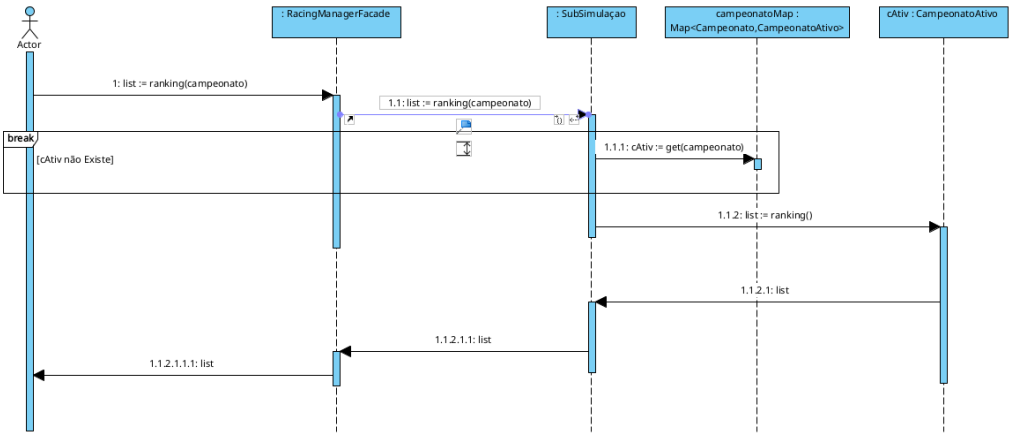


Figura 5.27: Diagrama de sequência do método ranking

O método "afinarCarro" aplica a função recebida que descreve as alterações aos atributos de um *Carro* ao *Carro* do jogadorID registado no *Campeonato* recebido. É levantado um erro caso o *CampeonatoAtivo* representante não exista ou caso o jogador não esteja registado no *CampeonatoAtivo*

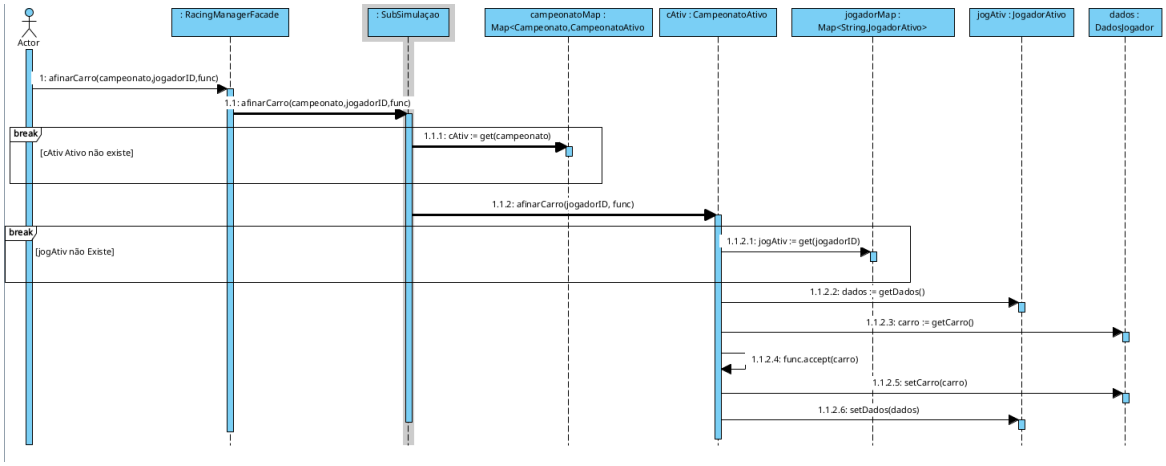


Figura 5.28: Diagrama de sequência do método afinarCarro

O método "temProxCorrida"devolve verdade se o *Campeonato* tem uma próxima corrida ainda por simular e falso caso contrário. É levantado um erro caso o *CampeonatoAtivo* representante não exista.

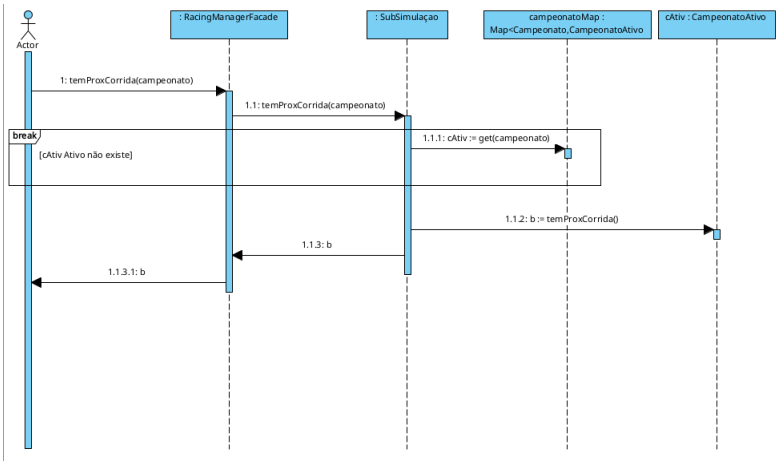


Figura 5.29: Diagrama de sequência do método temProxCorrida

6 Apresentação de DAOs

6.1 Subsistemas

Após a prévia implementação dos diagramas de classes, presente nos subsistemas do diagrama de componentes, agora é necessário a implementação de DAOs que não são nada mais nada menos do que as entidades responsáveis por fazer a ligação com as tabelas da nossa base de dados. Para isso é possível visualizar seguidamente a implementação dos mesmos em cada subsistema respetivamente, com a sua devida explicação. De igual forma, como os diagramas de sequências estão diretamente ligados às entidades presentes neste diagrama, irão sofrer alterações, porém para demonstrar isso apenas está presente um exemplo de um em cada subsistema. Os restantes diagramas de sequência estarão nos ficheiros enviados em anexo.

6.1.1 SubContas

Neste diagrama de classes submetido na última entrega intermédia, é visível agora a entidade **ContasDAO** que implementa o antigo **Map<String,Conta>**.

Nesta tabela estarão identificadas contas pelo **nome**, sendo ele do tipo *varchar*, e contendo os seguintes componentes, **password**, **type** e **pontos**. Estes atributos são do tipo, *varchar varchar* e *integer*, respetivamente.

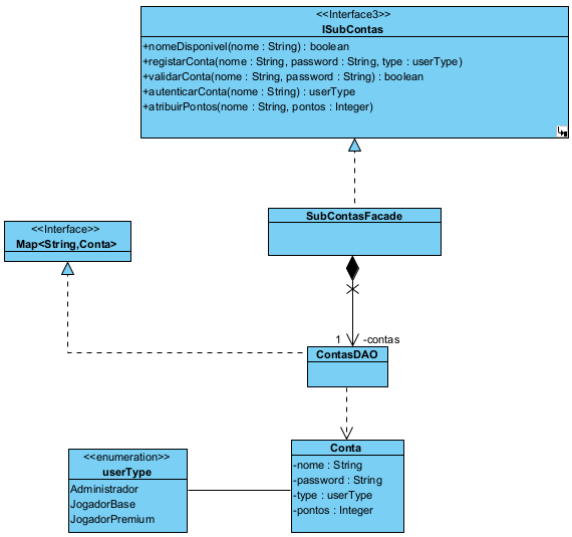


Figura 6.1: Diagrama de Classes SubContas com DAO

Nos diagramas de sequência é visível que o **ContasDAO** substitui o *Map*, neste caso em específico, não efetuando qualquer tipo de alterações nos restantes aspetos do diagrama.

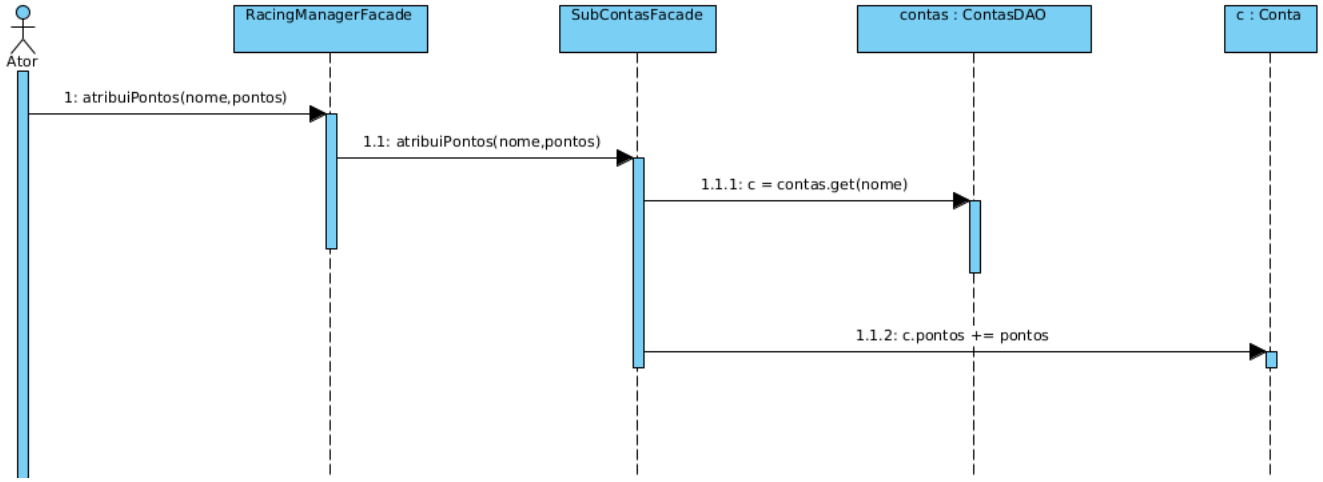


Figura 6.2: Diagrama de Sequências atribuirPontos com DAO

6.1.2 SubCampeonatos

Comparando ao diagrama de classes submetido na última entrega intermédia, é visível agora a entidade **CampeonatoDAO** que implementa o antigo **Map<Integer,Campeonato>** e o **CircuitoDAO** que por sua vez implementa **Map<String,Circuito>**.

Nesta tabela do **CampeonatoDAO** estarão identificadas campeonatos pelo **nome**, sendo ele do tipo *varchar*, e contendo os seguintes componentes, **nrcircuitos** e **disponibilidade**. Estes atributos são do tipo, *integer* e *boolean*, respetivamente. Já na tabela do **CircuitoDAO** estarão de igual forma identificados pelo nome, possuindo de restantes atributos, três inteiros *integers*, que se titulam de **nrvoltas**, **nrcurvas**, **nrchicanes**, um *varchar* referente ao *tempometereologico*. Como conseguimos gerar os segmentos dentro da classe **Circuito** não há necessidade de os armazenar.

Já nos diagramas de sequência é visível que o **CampeonatoDAO** substitui o *Map*, neste caso em específico, não efetuando qualquer tipo de alterações nos restantes aspetos do diagrama.

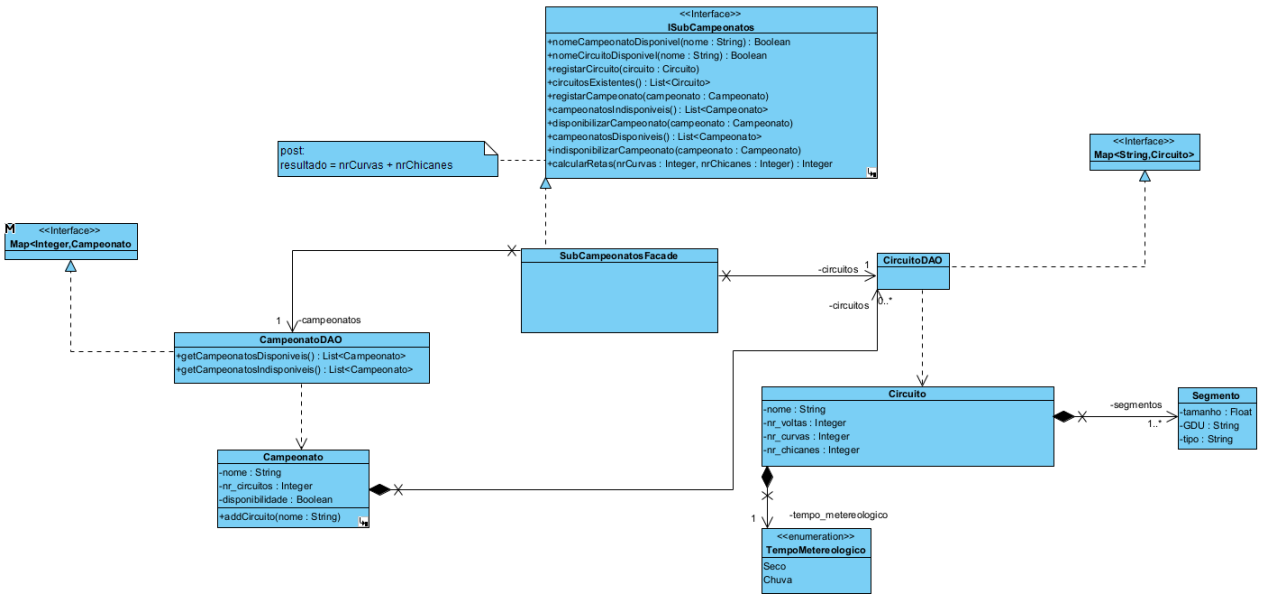


Figura 6.3: Diagrama de Classes SubCampeonatos com DAOs

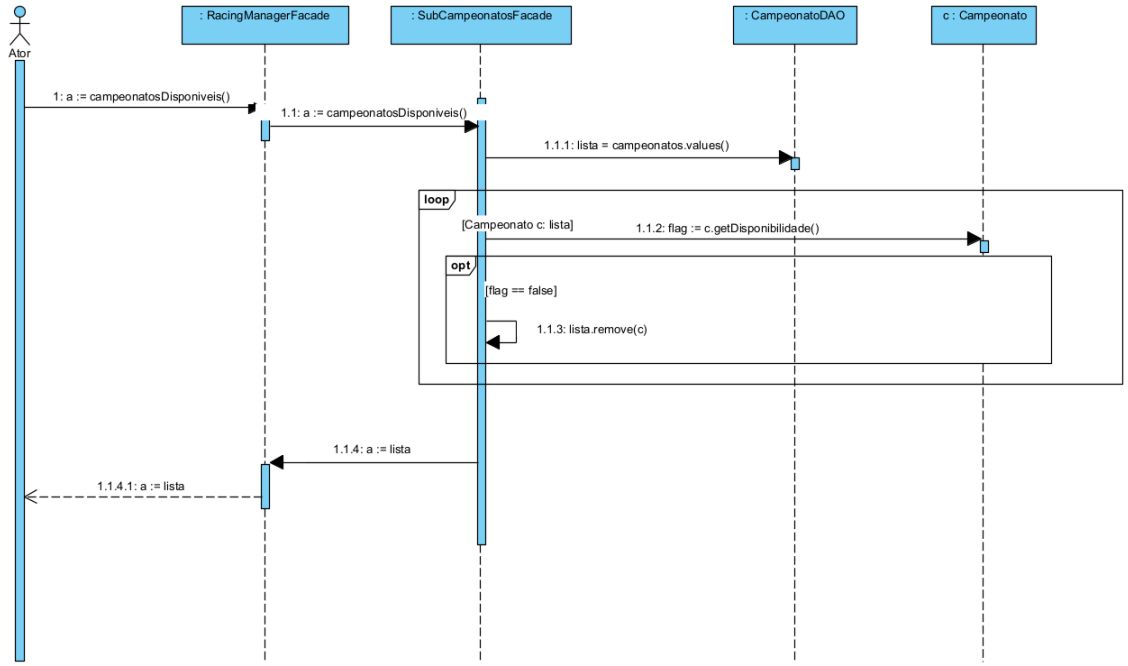


Figura 6.4: Diagrama de Sequências campeonatosDisponíveis com DAOs

6.1.3 Carros

Neste diagrama de classes submetido na última entrega intermédia, é visível agora a entidade **CarroDAO** que implementa o antigo **Map<String,Carro>**.

Nesta tabela do **CarroDAO** estarão identificadas carros pelo **ID**, sendo ele do tipo *varchar*, e contendo os seguintes componentes, **marca**, **modelo**, **cilindrada**, **potencia** e **fiabilidade**. Estas variáveis guardadeas na tabela possui dois tipos, as duas primeiras *varchar* e as restantes *integer* respetivamente. Dependendo do carro, uma vez que este pode ser herdado a primeiro nível por várias outras classes, as que possam possuir **motoreletrico** guardaram tal como *integer*.

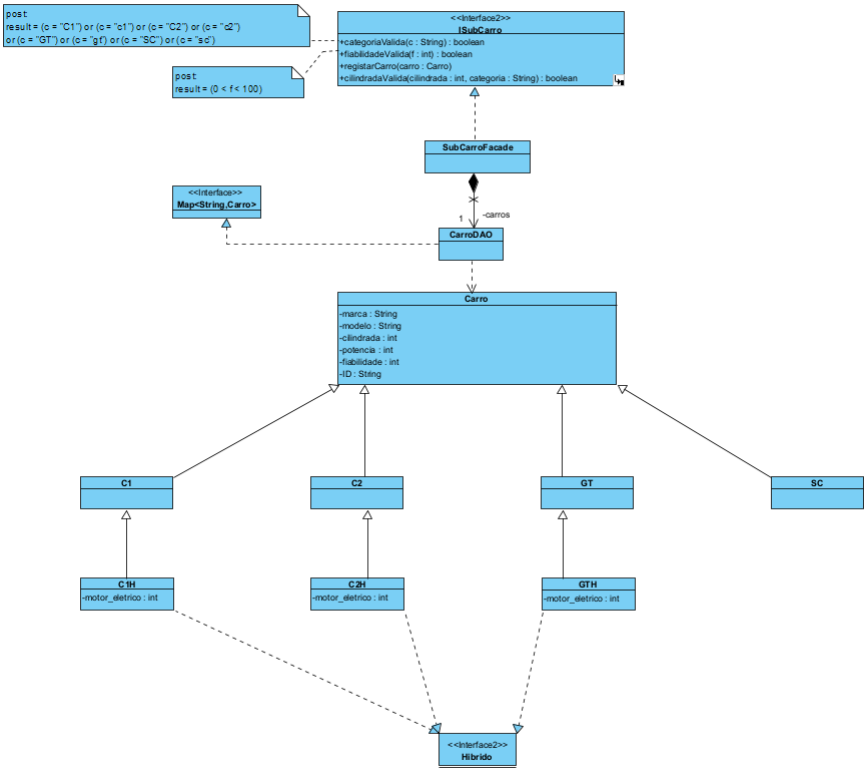


Figura 6.5: Diagrama de Classes SubCarros com DAOs

Seguidamente, este diagrama de classes é também visível o diagrama de sequências do método que apenas sofre, de igual forma, a outros casos anteriormente demonstrados, uma alteração no *Map* por **CarroDAO**.

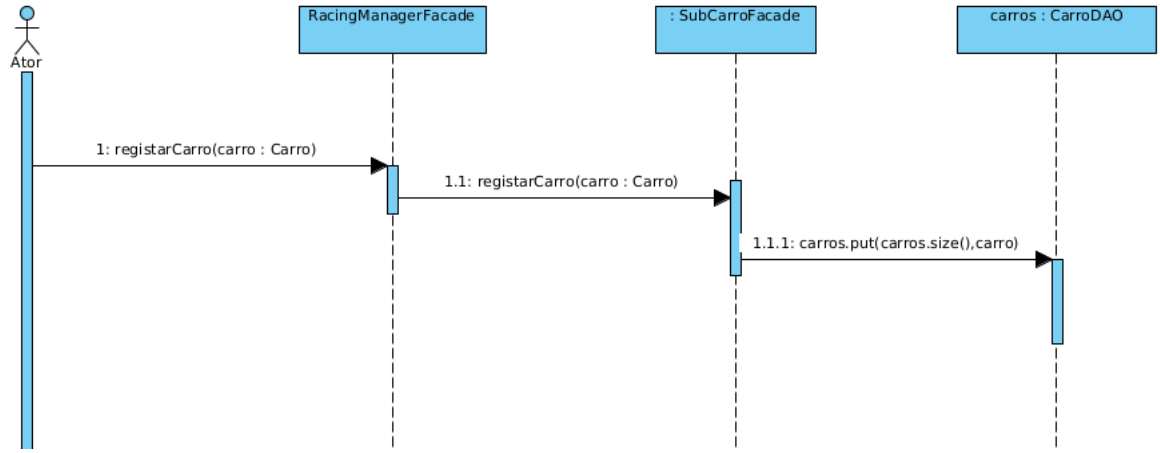


Figura 6.6: Diagrama de Sequências registarCarro com DAO

6.1.4 SubPilotos

Neste diagrama de classes submetido na última entrega intermédia, é visível agora a entidade **PilotoDAO** que implementa o antigo **Map<String,Piloto>**.

Em **PilotoDAO** tal como em grande parte, identifica-se como chave primária o **nome** do piloto, sendo esta chave um *varchar*, acabando por esta tabela guardar dois valores do tipo *float* que se titulam de **cts** e **sva**

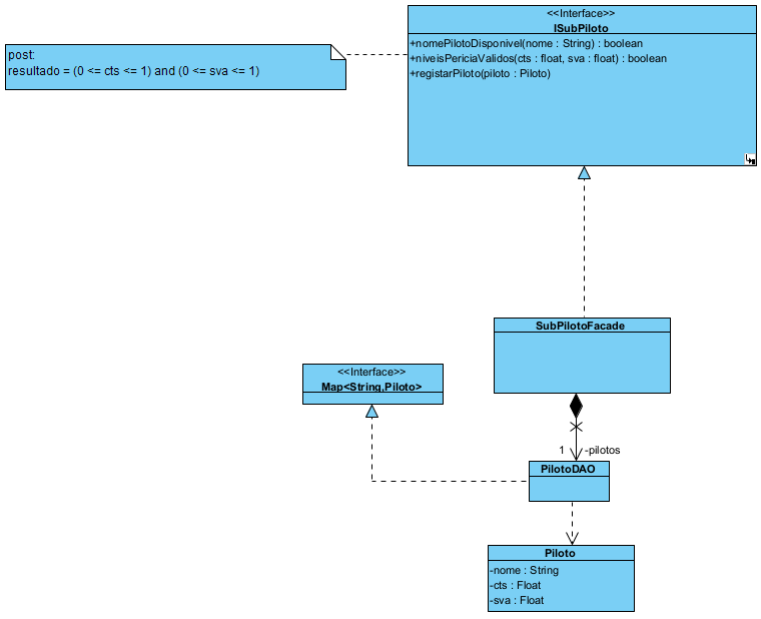


Figura 6.7: Diagrama de Classes SubPilotos com DAOs

Posteriormente a este diagrama de classes é também visível o diagrama de sequências do método *nomePilotoDisponivel* que apenas sofre, de igual forma, a outros casos anteriormente demonstrados, uma alteração no *Map* por **PilotoDAO**.

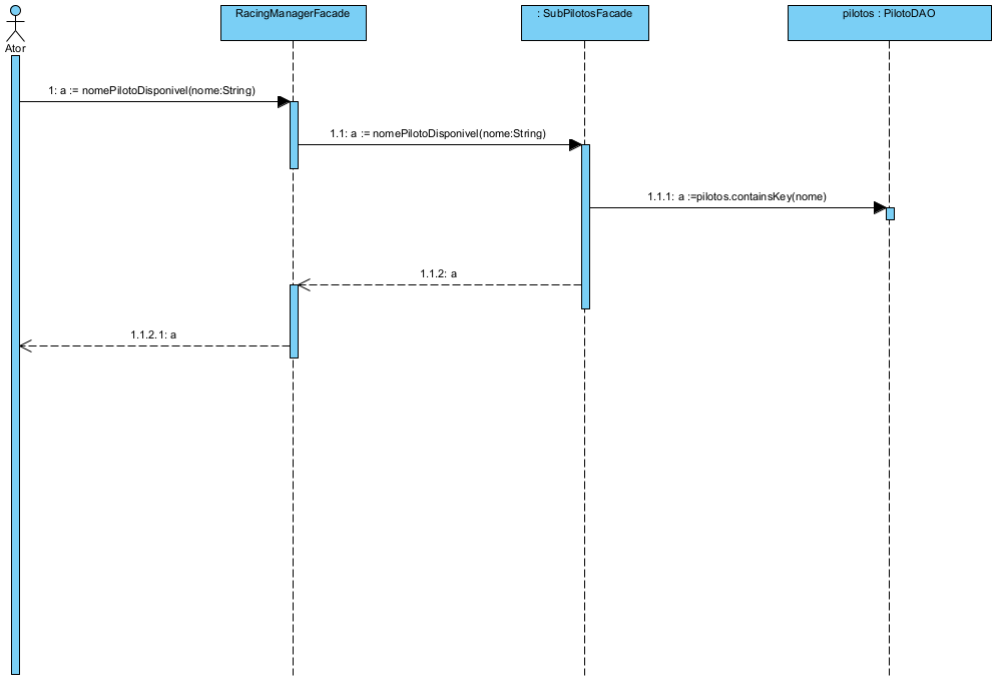


Figura 6.8: Diagrama de Sequências nomePilotoDisponivel com DAO

6.1.5 SubSimulação

Comparando ao diagrama de classes submetido na última entrega intermédia, é visível agora a entidade **CampeonatoAtivoDAO** que implementa o antigo **Map<Integer,CampeonatoAtivo>** e o **JogadorAtivoDAO** que por sua vez implementa **Map<Integer,JogadorAtivo>**.

Por fim no subsistema de simulação possumimos, tal como referenciado anteriormente, as tabelas de **CampeonatoAtivoDAO** e **JogadorAtivoDAO**. Referente ao **CampeonatoAtivoDAO** este é identificado pelo seu **id** do tipo *integer* e o atual número da corrida **nCorridaAtual** também ele um *integer*. Guarda também a chave estrangeira do **JogadorAtivoDAO**, este que possui como chave o **jogadorID** proveniente de **DadosJogador**, do tipo *barchar* e outros dois valores, um *boolean* e um *integer* que respetivamente são: **pronto** e **nAfinacoes**.

Já neste diagrama de sequências é visível que o **CampeonatoAtivoDAO** e **JogadorAtivoDAO** substitui os *Map*, neste caso em específico, não efetuando qualquer tipo de alterações nos restantes aspetos do diagrama.

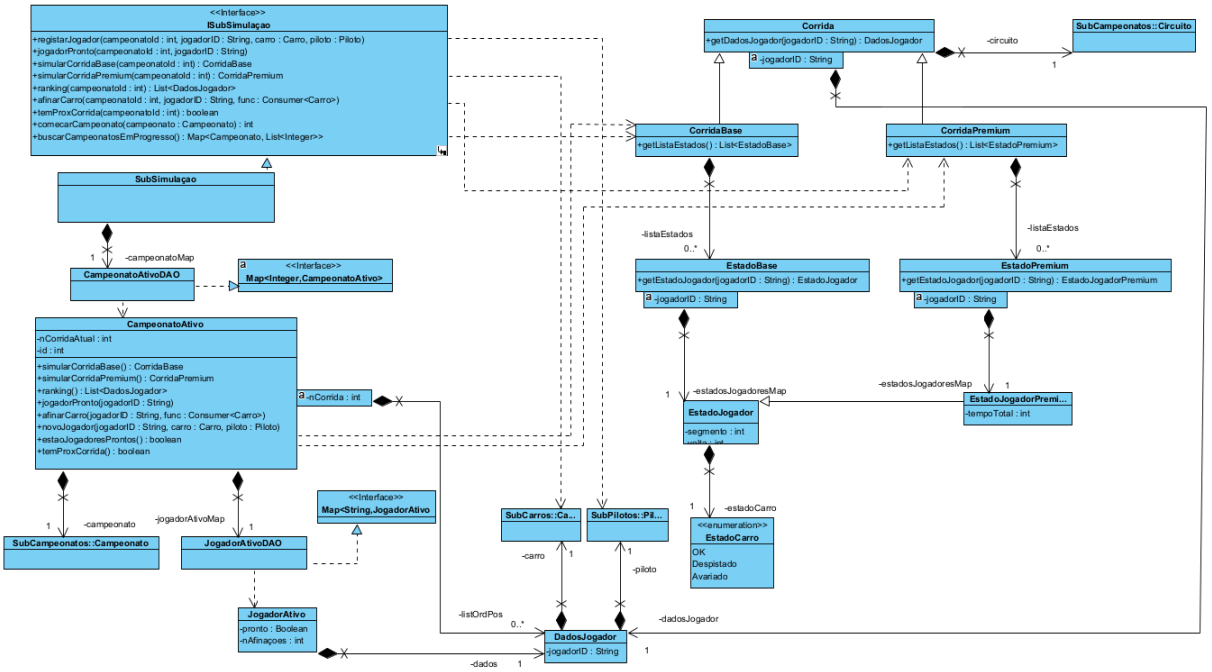


Figura 6.9: Diagrama de Classes SubSimulação com DAOs

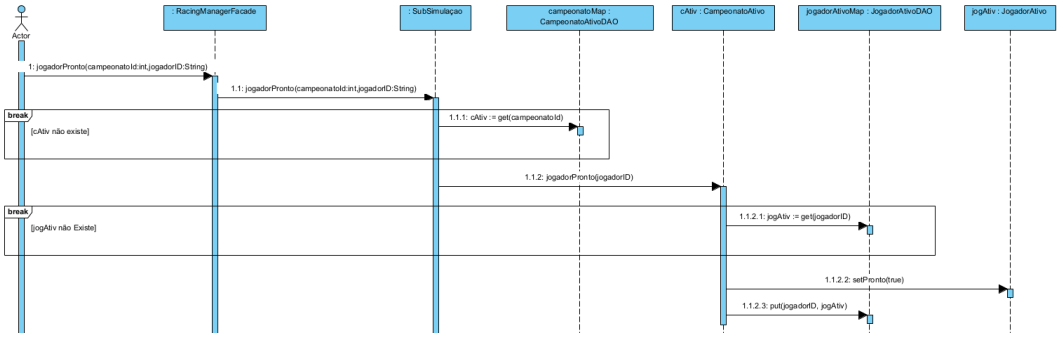


Figura 6.10: Diagrama de Sequências jogadorPronto com DAO

7 Interface

7.1 Mapa de Navegação

De modo a ser mais fácil a decisão de operações a demonstrar na UI, tal como recomendado, criamos um mapa de navegação utilizando um diagrama máquina de estado onde é visível as operações possíveis em cada menu, sendo que a aplicação possui 2 menus. Um menu referente apenas às operações possíveis a ser realizadas pelos administradores e eoutro para as ações de utilizadores em geral.

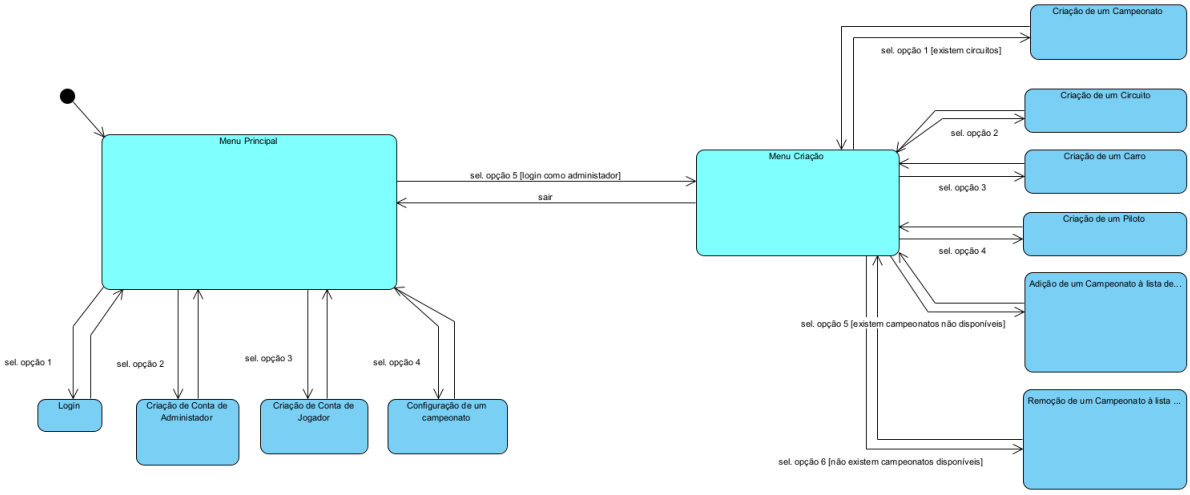


Figura 7.1: Mapa de Navegação UI

7.2 UI

Tal como visualizado no mapa de navegação, o nosso menu principal possui as opções registo, login, entrar no menu administrador e jogar. Em registo, apesar de aparecer os casos de registar conta de administrador e registar conta de utilizador, mas apenas será possível criar conta de não administradores uma vez que nem todos os utilizadores do *Racing Manager* poderão o ser. Após o seu registo ficaram automaticamente logados, sendo assim isentos do mesmo. A operação de avançar para o menu de administrador encontra-se portanto, com uma restrição de acesso a apenas utilizadores logados e identificados como administradores. Por fim, terá a opção de iniciar o jogo, podendo assim configurar o campeonato, configurar piloto, carro e posteriormente simular o campeonato.

Já no menu de administrador, é possível adicionar carros, pilotos, circuitos. É também possível criar campeonatos com uma coletânea de circuitos, podendo assim disponibiliza-los e indisponibiliza-los.

8 Implementação Final

Tal como demonstrado previamente neste documento, no diagrama de componentes é visível a *interface* **IRacingManager**, a *facade* **RacingManagerFacade** que resulta na criação do ficheiro **RacingManager** nque executa o típico papel de *main* de qualquer projeto. Este módulo chama um método intitulado de *TextUI()* que demonstra as diversas operações possíveis presentes na UI. Essas operações utilizam os métodos implementados nesta **RacingManagerFacade** que possui todos os métodos públicos de todos os outros subsistemas presentes.

Figura 8.1: Implementações Finais

9 Conclusão

Dando fim às três fases do trabalho prático, onde definimos o jogo *RacingManager*, através do desenvolvimento da arquitetura conceptual do sistema, arquitetura relacional, arquitetura funcional e não funcional. Sendo agora esta última fase, comparativamente a outras fases, esta demonstrou ser muito menos trabalhosa. Isto porque nas outras fases, apesar de umas diagramas terem um conceito individual entre si, tornavam-se processos repetitivos. Após esses diagramas todos definidos que acabam por restringir e definir as ações necessárias para atingir os objetivos a implementação torna-se bastante mais simples. Utilizando também as ferramentas do *Visual Paradigm* conseguimos converter grande parte da estrutura do projeto para *Java* poupando assim bastante tempo. Obviamente que tivemos que verificar tudo e ajustar algumas partes do código pois obviamente esta conversão não era perfeita.

Durante a reta final verificamos também que há métodos ou estruturas que possivelmente teriamos criado de outra forma, de modo a ser mais simples e estando mais completo. Quer isto seja devido à falta de um planeamento sobre determinado assunto em questão, falta de conhecimento e/ou estar habituado a este novo método de trabalho, i.e. projetar graficamente utilizando diagramas todas as funcionalidas e restrições, ligações entre outras coisas de um projeto.

Por fim, concluimos que este trabalho prático foi bastante evolutivo na gestão/criação de projetos, dando nos um bom conjunto de conhecimentos/métodos/processos necessários a executar, e só *a posteriori*, passar para a implementação a nível mais baixo, a codificação.

Anexos

Tal como requerido nesta entrega final, encontram-se, em anexo, , o mapa de navegação, o diagrama de componentes, com os seus respetivos diagramas de classes e de sequência, e também as identificações das responsabilidades do sistema nos use cases.