

F1 Manager: Especificação UML

Desenvolvimento de Sistemas de Software

Guilherme Sampaio

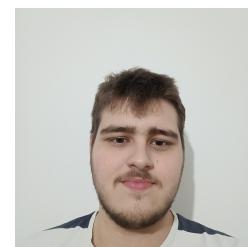
Luís Araújo

Luís Pereira

A96766

A86772

A96681



Rui Oliveira

A95254



Tiago Pereira

A95104



Grupo 20

<https://github.com/ruioliveira02/dss>



Departamento de Informática

Universidade do Minho

novembro 2022

Conteúdo

1	Introdução	3
1.1	Trabalho Anterior	3
2	Subsistemas	4
2.1	UserSystem	5
2.2	RaceSystem	6
2.2.1	prepareForRace	6
2.2.2	getRaceState	6
2.2.3	getRaceResults	6
2.3	ChampionshipSystem	6
2.3.1	signUp	7
2.3.2	getStandings	7
2.3.3	getAvailableDrivers	7
2.3.4	canChangeSetup	7
2.3.5	changeSetup	8
2.3.6	setStrategy	8
2.4	ContentSystem	8
3	Simulação de Corridas	9
3.1	Desgaste de Pneus	9
3.2	Fiabilidade	11
3.3	Meteorologia	12
3.4	Desempenho	13
3.5	Erros	14
3.6	Ultrapassagens	15
4	Conclusão	16
5	Anexos	16
5.1	Diagrama de Componentes	17
5.2	Diagrama de Classes	18
5.3	Diagramas de Sequência	18
5.4	Diagrama de Atividades	23

Lista de Figuras

1	Modelo de Domínio	3
2	Diagrama de Casos de Uso	4
3	Gráfico de polinómios de graus sucessivos (2 a 5)	10
4	Secção do diagrama de atividades relativa ao desgaste de pneus	11
5	Secção do diagrama de atividades relativa à fiabilidade dos carros	12
6	Secção do diagrama de atividades relativa à meteorologia	13
7	Secção do diagrama de atividades relativa ao desempenho dos carros	14
8	Secção do diagrama de atividades relativa aos erros dos pilotos	15
9	Secção do diagrama de atividades relativa às ultrapassagens	16
10	Diagrama de Componentes	17

11	Diagrama de Classes	18
12	Diagrama de Sequência do Método <i>loginAdmin</i>	18
13	Diagrama de Sequência do Método <i>loginPlayer</i>	19
14	Diagrama de Sequência do Método <i>canChangeSetup</i>	19
15	Diagrama de Sequência do Método <i>changeSetup</i>	19
16	Diagrama de Sequência do Método <i>getAvailableDrivers</i>	20
17	Diagrama de Sequência do Método <i>getStandings</i>	20
18	Diagrama de Sequência do Método <i>getRaceState</i>	20
19	Diagrama de Sequência do Método <i>prepareForRace</i>	21
20	Diagrama de Sequência do Método <i>registerAdmin</i>	21
21	Diagrama de Sequência do Método <i>registerPlayer</i>	22
22	Diagrama de Sequência do Método <i>setStrategy</i>	22
23	Diagrama de Sequência do Método <i>signUp</i>	23
24	Diagrama de Sequência do Método <i>getRaceResults</i>	23
25	Diagrama de Atividades da Simulação de Corridas (parte 1)	24
26	Diagrama de Atividades da Simulação de Corridas (parte 2)	25

Lista de Tabelas

1	Caso de uso de registo de jogadores (revisão)	5
---	---	---

1 Introdução

O presente relatório é referente à segunda fase do projeto prático da Unidade Curricular de Desenvolvimento de Sistemas de Software da Licenciatura em Engenharia Informática do Departamento da Informática da Escola de Engenharia da Universidade do Minho, para o ano de 2022/2023.

O projeto apresentado foi realizado pelo grupo número 20, constituído por Guilherme Geraldes Sampaio (A96766), Luís Guilherme Guimarães de Araújo (A86772), Luís Manuel Fernandes Pereira (A96681), Rui Pedro Esteves Vasques Correia de Oliveira (A95254), e Tiago Miguel Moreira Bacelar Pereira (A95104).

O objetivo desta segunda fase consiste em desenvolver uma especificação recorrendo a diagramas UML que sustente a implementação do sistema pretendido.

Todos os diagramas desenvolvidos para esta fase encontram-se em anexo. Juntamente com este relatório, foi enviado o ficheiro do *Visual Paradigm* que suportou a sua construção.

1.1 Trabalho Anterior

Antes de avançar para a especificação UML desenvolvida, é relevante recordar o trabalho desenvolvido na primeira fase do projeto. Assim sendo, as figuras 1 e 2 representam o modelo de domínio desenvolvido anteriormente e o diagrama de casos de uso, respetivamente.

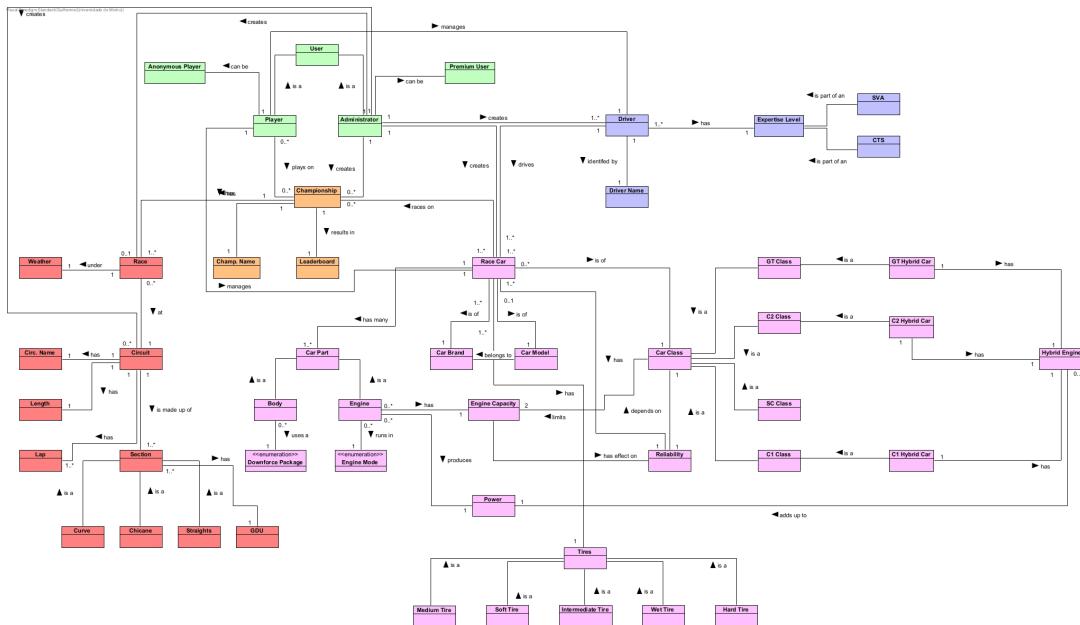


Figura 1: Modelo de Domínio

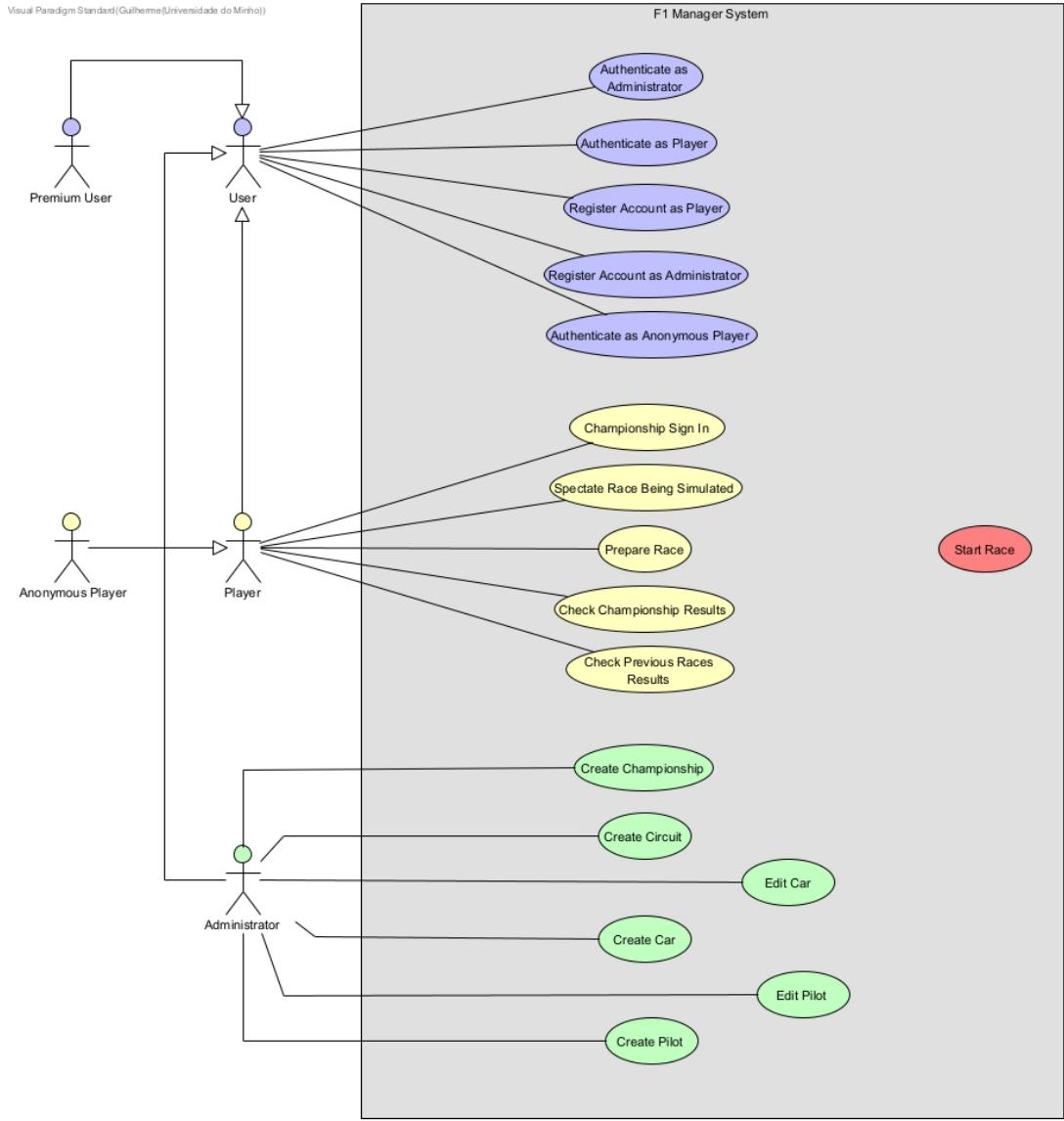


Figura 2: Diagrama de Casos de Uso

2 Subsistemas

Decidiu-se, como é demonstrado pelo diagrama de componentes da figura 10, que a aplicação seguiria uma arquitetura em três camadas. A camada **F1ManagerUI** será responsável pela interface com o utilizador; a camada **F1ManagerBL** por toda a lógica de negócio da aplicação; e camada **F1ManagerDL** pela lógica de dados do sistema.

Por questões de simplicidade, e como foi pedido, apenas se modelou em detalhe a camada de lógica de negócios.

Através da análise dos casos de uso desenvolvidos na fase anterior, decidiu-se criar os seguintes subsistemas da lógica de negócio:

- ContentSystem: subsistema relacionado com toda a gestão do conteúdo (carros, circuitos e pilotos) existentes na aplicação.
- RaceSystem: subsistema responsável pela gestão das corridas: desde a sua preparação até à sua simulação.

- ChampionshipSystem: subsistema responsável pela gestão dos campeonatos, incluindo a sua criação, inscrição e consulta de dados relevantes aos mesmos.
- UserSystem: subsistema responsável pela autenticação dos utilizadores (jogadores e administradores) na aplicação.

Dada a reduzida janela temporal disponível para a realização desta fase do projeto, apenas se modelou o comportamento do subsistema de corridas, dado ser este o mais importante para a aplicação.

Procurou-se maximizar a independência dos diferentes subsistemas. No entanto, dadas as fortes interligações entre as diferentes entidades do modelo de dados, não foi possível atingir independência total.

2.1 UserSystem

Como mencionado anteriormente, este sistema é responsável por tratar todas as questões de autenticação no sistema. Para isso, naturalmente, são precisos métodos para administradores e jogadores se inscreverem no sistema. Estes advém diretamente dos casos de uso assinalados a azul no diagrama; no entanto, e conforme visível no caso de uso de registar jogador, que é recordado na tabela 1, há ainda a necessidade de verificar a existência de um determinado nome de utilizador. Daí surgir o método `doesUsernameExist`.

Use Case	Registrar Jogador	
Autor	Utilizador	
Pré-Condição		
Pós-Condição	Jogador existe	
Fluxo normal	Autor	Sistema
	1. Utilizador dá o seu username e a sua password	2. O sistema regista o jogador
Fluxo de exceção (o username já existe)	1. Sistema notifica o utilizador de que já existe alguém com aquele nome de utilizador	

Tabela 1: Caso de uso de registo de jogadores (revisão)

De forma análoga, o caso de uso de registo de jogador deu origem ao diagrama de sequência da figura 21.

Os restantes métodos de registo e *login* são bastante semelhantes ao descrito anteriormente, pelo que não serão aqui apresentados em detalhe.

2.2 RaceSystem

O sistema de corridas é, como o próprio nome indica, responsável por tratar de questões relativas à participação nas corridas. Relativamente aos casos de uso, refere-se a alguns de entre aqueles pintados de amarelo.

Mais concretamente, este sistema cobre os casos de uso de preparar corrida, observar a corrida enquanto esta é simulada, e consultar resultados de eventos anteriores. Estes casos de uso levaram, de imediato, ao surgimento dos métodos `prepareForRace`, `getRaceState` e `getRaceResults`, respetivamente.

2.2.1 `prepareForRace`

Este método indica que o jogador já decidiu a estratégia e a afinação do carro que pretende utilizar na corrida. Mal todos os jogadores tiverem indicado prontidão para correr, a corrida começa de forma automática e imediata.

Esta necessidade de verificar se a corrida tem ou não de começar é o aspetto mais relevante do diagrama de sequência da figura 19. Além disso, é feita uma validação relativa à existência ou não da corrida fornecida no sistema.

2.2.2 `getRaceState`

Este método é invocado sempre que um jogador pretende saber o estado atual de uma corrida que está a ser simulada. A assinatura do método e a sua especificação são relativamente simples: o método apenas deve receber uma corrida, verificar que a corrida efetivamente existe e, em caso positivo, devolver uma nova cópia do estado atual da mesma.

Este comportamento é descrito pelo diagrama de sequência da figura 18.

2.2.3 `getRaceResults`

Este método devolve o resultado de uma corrida já concluída. Para isso, recebe, obviamente, a corrida sobre a qual se pretende obter resultados. Os resultados devolvidos serão devolvidos numa lista, por ordem de classificação. Essa necessidade de haver ordem é precisamente o motivo da escolha dessa estrutura de dados em detrimento de outras.

O comportamento deste método é modelado pelo diagrama de sequência da figura 24. Resumidamente, é feita a validação habitual da existência da corrida na base de dados; assim como uma verificação adicional de que a corrida efetivamente terminou. Esta última é efetuada através de uma chamada ao método `hasFinished` da classe `Race`.

2.3 ChampionshipSystem

O sistema de gestão de campeonatos, como o próprio nome indica, trata de todas as questões diretamente ligadas com campeonatos. Na sua essência, é responsável por todos os casos de uso a amarelo, à exceção de assistir a corridas, que faz parte do subsistema de corridas.

Além dos métodos que derivam trivialmente dos diferentes casos de estudo, foi necessária a criação de um método: `getAvailableDrivers`. Este método surge da restrição imposta pelo caso de uso de inscrição num campeonato de um piloto só poder ser utilizado por um só jogador.

Além disso, o caso de uso de preparação de corrida levou à criação de três métodos: canChangeSetup, changeSetup e setStrategy. Estes métodos resolvem as questões de alteração da configuração do carro e da estratégia da corrida definidos por esse caso de uso.

2.3.1 signUp

Este método inscreve um jogador para participar num campeonato. Para isso, recebe, obviamente, o campeonato no qual o jogador quer participar, que jogador é, o carro e piloto com que quer participar.

O comportamento deste método é modelado pelo diagrama de sequência da figura 23. Resumidamente, é feita a validação habitual da existência do campeonato e jogador na base de dados; assim como uma verificação adicional de que o carro e o piloto não estão já a ser utilizados por outro jogador. Estas últimas são efetuadas através de uma chamada aos métodos isRaceCarInUse e isDriverInUse da classe championship.

2.3.2 getStandings

Este método devolve as classificações atuais dos participantes em um campeonato. Para isso, recebe, obviamente, o campeonato sobre a qual se pretende obter as classificações. Os resultados serão devolvidos num mapa, de participante para numero de pontos no campeonato. A necessidade de saber o numero de pontos por participante é precisamente o motivo da escolha dessa estrutura de dados em detrimento de outras.

O comportamento deste método é modelado pelo diagrama de sequência da figura 17. Resumidamente, é feita a validação habitual da existência do campeonato na base de dados; assim como uma iteração pelas corridas existentes no campeonato obtendo em cada um a lista de classificações. Esta última é posteriormente transformada em pontos através de uma chamada ao método estático getPointsOfPosition da classe Race.

2.3.3 getAvailableDrivers

Este método devolve todos os pilotos ainda não escolhidos num campeonato. Para isso, recebe, obviamente, o campeonato sobre o qual se pretende obter os pilotos. Os resultados devolvidos estarão numa lista. A necessidade de apenas saber que pilotos existem é precisamente o motivo da escolha dessa estrutura de dados em detrimento de outras.

O comportamento deste método é modelado pelo diagrama de sequência da figura 16. Resumidamente, é feita a validação habitual da existência do campeonato na base de dados. Também são obtido todos os pilotos na base de dados, realizando uma iteração por todos os participantes existentes no campeonato com o método getDriver da classe Participant, e remove-se da lista de todos os pilotos aqueles que já foram escolhidos.

2.3.4 canChangeSetup

Este método devolve se um jogador pode mudar a configuração do seu carro. Para isso, recebe, obviamente, o campeonato sobre a qual se pretende verificar se pode mudar a configuração e qual é o jogador a quem verificar. O resultado devolvido será um booleano indicando se pode ou não mudar.

O comportamento deste método é modelado pelo diagrama de sequência da figura 14. Resumidamente, é feita a validação habitual da existência do campeonato na base de dados; a verificação que o jogador participa no campeonato; depois obtém-se o número de vezes que o jogador já trocou a configuração no campeonato (utilizando-se, para esse efeito, o método `getNumberOfSetupChanges` da classe `Participant`); e o número de corridas no campeonato. Por fim, compara-se estes dois e retorna verdadeiro se o número de trocas já efetuadas for menor do que $\frac{2}{3}$ do número de corridas no campeonato.

2.3.5 changeSetup

Este método muda a configuração do carro de um jogador se puder mudar num campeonato. Para isso, recebe, obviamente, o campeonato sobre o qual se pretende mudar a configuração, o jogador e o componente alvos da modificação. O resultado será a modificação, se possível, da configuração do carro do jogador no campeonato.

O comportamento deste método é modelado pelo diagrama de sequência da figura 15. Resumidamente, é feita a validação habitual da existência do campeonato na base de dados; a verificação que o jogador participa no campeonato; depois verifica-se se o jogador não ultrapassou o numero de máximo de vezes que pode alterar a configuração com o método `canChangeSetup` da classe `Participant`. Se ainda não tiver ultrapassado o limite obtém-se qual o carro do jogador com o método `getRaceCar` da classe `Participant` e modifica-se a sua configuração. Por fim aumenta-se o numero de vezes que o utilizador alterou a configuração por um com o auxilio do método `increaseNumberOfSetupChanges` da classe `Participant`.

2.3.6 setStrategy

Este método muda a estratégia do carro de um jogador num campeonato. Para isso, recebe, obviamente, o campeonato sobre no qual se pretende mudar a configuração, qual é o jogador a quem modificar, para que tipo de pneu se mudar e para que estado do motor mudar. O resultado será a modificação da estratégia do carro do jogador no campeonato.

O comportamento deste método é modelado pelo diagrama de sequência da figura 22. Resumidamente, é feita a validação habitual da existência do campeonato na base de dados; a verificação que o jogador participa no campeonato; depois obtém-se qual o carro do jogador com o método `getRaceCar` da classe `Participant` e modifica-se a sua estratégia.

2.4 ContentSystem

Este subsistema é responsável pela gestão do conteúdo disponível no jogo, ou seja, pela criação de novos pilotos, circuitos, carros e campeonatos. Em termos de diagrama de casos de uso, corresponde aos verdes.

Os métodos deste subsistema são métodos que sustentam a criação, consulta e edição dessas mesmas entidades.

Dado não estarem relacionados com o cenário 5, e tal como foi indicado pela equipa docente, os métodos deste subsistema não foram modelados em detalhe.

3 Simulação de Corridas

Dado o papel central da simulação de corridas, o grupo considerou relevante a sua modelação através de um diagrama de atividades. As figuras 8, 9, 7, 5, 4, 6 e 25 e 26 em anexo correspondem a esse mesmo diagrama.

De um modo abstrato, e após análise do enunciado fornecido, decidiu-se que a simulação se poderia dividir nas seguintes partes:

1. Simulação de desgaste de pneus
2. Simulação da fiabilidade dos carros
3. Simulação da meteorologia
4. Simulação de desempenho
5. Simulação de erros por parte dos pilotos
6. Simulação de ultrapassagens

Feita esta divisão, é necessário decidir a ordem em que as simulações são feitas. A ordem decidida foi a ordem de apresentação das mesmas, ou seja, o desgaste de pneus é a primeira componente da simulação a executar e as ultrapassagens a última.

Essencialmente, podemos dividir os componentes da simulação em dois grupos: os diretamente relacionados com a classificação em pista (os últimos três a executar) e os não relacionados (os primeiros três a executar). A ordem de execução relativa destes grupos não é relevante: poder-se-ia ter trocado e o resultado, apesar de diferente, não seria melhor nem pior do que a solução proposta.

Entre os três primeiros a executar, a ordem não é relevante. No entanto, a ordem relativa dos últimos três componentes a executar é mais delicada. A questão prende-se, essencialmente, com o momento onde se simulam as ultrapassagens. O grupo decidiu que isso devia ser o último componente da simulação a executar.

Esta decisão prende-se com o facto de, se as ultrapassagens ocorressem antes de simular o desempenho dos carros e os erros por partes dos pilotos, poderia haver situações onde as ultrapassagens seriam *desfeitas* pelos elementos seguintes da simulação, o que dificultaria o cumprimento do pedido pelo enunciado.

3.1 Desgaste de Pneus

A simulação de desgaste de pneus é relativamente linear. Os diferentes compostos de pneus têm diferentes graus de desgaste: pneus macios desgastam-se mais depressa que médios e duros, por exemplo.

Consideramos o desgaste do pneu como sendo um valor de 0 a 1, em que 0 corresponde a um pneu novo e 1 a um pneu no fim da sua vida. A cada momento, o desgaste aumenta um valor predeterminado em função do composto usado (estes valores são afinados em fase de implementação).

Adicionalmente, se os pneus usados forem de chuva (intermédios ou de chuva), a esse fator multiplica-se ($2 - \text{quantidade_de_agua_na_pista}$), de forma a fazer o desgaste ser superior se os pneus forem usados inapropriadamente em piso seco, o que acontece na vida real.

Depois do desgaste de pneus simulado, foi necessário simular a ocorrência de furos, para desincentivar a utilização em massa dos pneus mais macios. Assim sendo, gera-se um número aleatório entre 0 e 1. Se o $desgaste^{constante_de_furo}$ for maior ou igual que esse número, acontece um furo. Essa expressão deve-se ao facto de se querer que também aconteçam furos com baixos níveis de desgaste, embora com muita baixa probabilidade; mas com elevados valores de desgaste o furo deve ser quase certo. Esta é uma propriedade das funções polinomiais no intervalo de 0 a 1, sendo a constante de furo o grau do polinómio, que poderá ser definida durante a implementação para afinar o quanto acentuada é a curva.

A figura 3 demonstra essas funções, graus maiores tornam menos provável que haja furos com valores muito baixos de desgaste, como pretendido.

O diagrama da figura 4 demonstra esta arquitetura num diagrama de atividades.

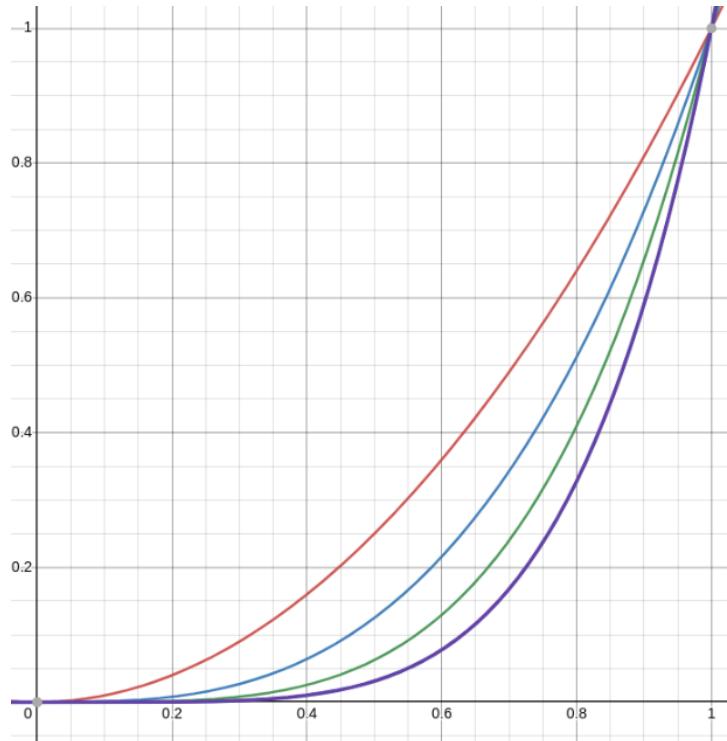


Figura 3: Gráfico de polinómios de graus sucessivos (2 a 5)

Fonte: desmos.com

A curva fica cada vez mais acentuada à medida que o grau aumenta (a vermelho é um polinómio do 2º grau, a roxo é um do 5º grau)

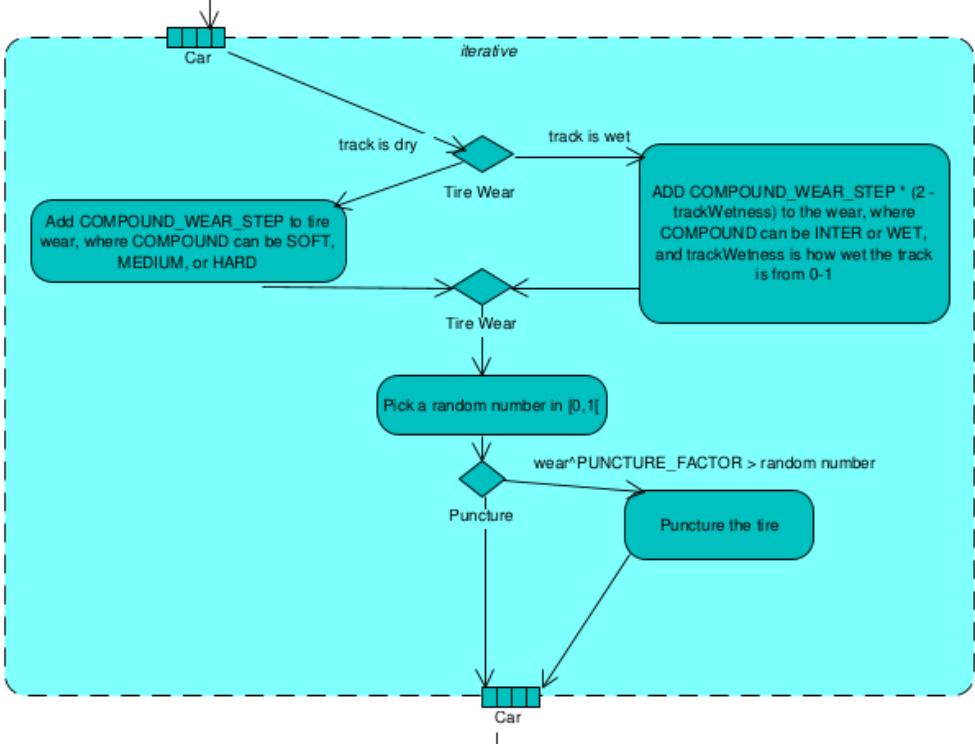


Figura 4: Secção do diagrama de atividades relativa ao desgaste de pneus

3.2 Fiabilidade

A fiabilidade dos carros depende significativamente da classe a que estes pertencem. Por exemplo, os carros da classe C1 tem 95%¹.

Após esse valor de fiabilidade ser calculado, e assumindo que a simulação irá atualizar a um dado número f por volta², compara-se o valor $\sqrt{1 - \text{fiabilidade}}$ ³, com um número gerado aleatoriamente entre 0 e 1. Se o número aleatório for maior, o carro tem uma avaria.

Este fluxo está modelado no diagrama de atividades da figura 5

¹Fiabilidade corresponde à probabilidade do carro terminar a volta em que se encontra

²este valor será calculado individualmente por classe de carro

³Este valor, ao fim de f iterações independentes, garante uma probabilidade de o carro não falhar igual a *fiabilidade*. A sua demonstração advém trivialmente das propriedades de eventos independentes, e não será aqui apresentada.

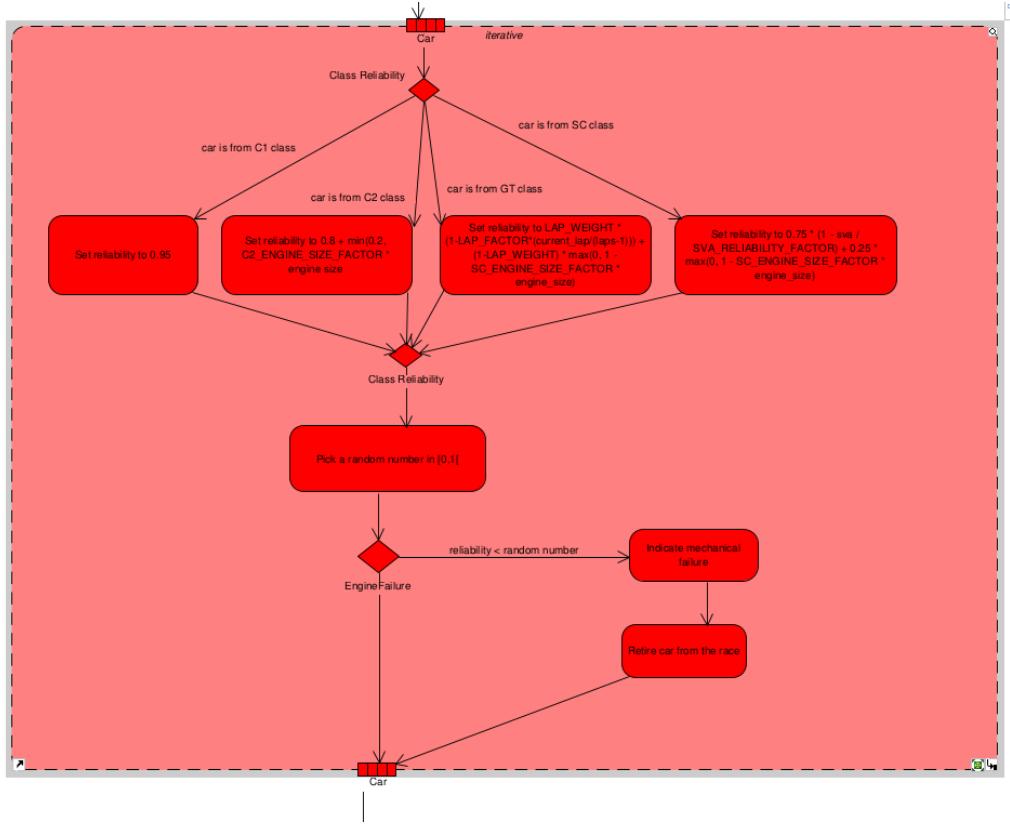


Figura 5: Secção do diagrama de atividades relativa à fiabilidade dos carros

3.3 Meteorologia

Embora esta simulação não seja um requisito essencial do projeto, o grupo considerou que seria de valor acrescentar uma componente variável à meteorologia durante as corridas.

Resumidamente, as condições climatéricas a um qualquer momento da corrida são definidas por 2 valores: a intensidade da chuva e a quantidade de água na pista. Esta quantidade de água na pista é igual para todo o circuito. Em iterações futuras do projeto, poder-se-ia distinguir as diferentes secções da pista, tornando a simulação mais realista. Além disso, antes de uma corrida começar, é selecionado um valor de variabilidade, entre zero e um, que corresponde ao quanto instável são as condições meteorológicas: 0 significa que não há alterações à quantidade de chuva; 1 significa que esse valor pode flutuar significativamente a qualquer momento.

Em qualquer momento da simulação, é calculado um fator aleatório entre -1 e 1. A inclusão de números negativos e positivos deve-se ao facto de se querer permitir flutuações do nível de chuva em ambos os sentidos. Esse fator é depois usado num produto que é usado para alterar o valor da quantidade de chuva. Esse produto tem um fator, WEATHER_CHANGE_FACTOR, que será definido em fase de implementação, que serve para se poder manipular o comportamento do sistema nessa mesma fase, de forma a melhorar a jogabilidade do *software* desenvolvido.

Antes de continuar, o valor da quantidade de chuva é limitado a 1 no máximo e 0 no mínimo, de forma a mantê-lo em valores aceitáveis.

O valor atualizado da quantidade de chuva é utilizado, de seguida, para calcular a variação da quantidade de água na pista. Há duas forças em jogo: a quantidade de água

adicionada pela chuva que caiu, e a água escoada pelos pneus dos carros. Assim sendo, a variação da quantidade de água em pista é a diferença desses valores.

Finalmente, e para balizar os valores como feito anteriormente, a quantidade de água na pista é limitada ao intervalo de zero até um.

Esta modelação encontra-se sob a forma de um diagrama de atividades, apresentado na figura 6.

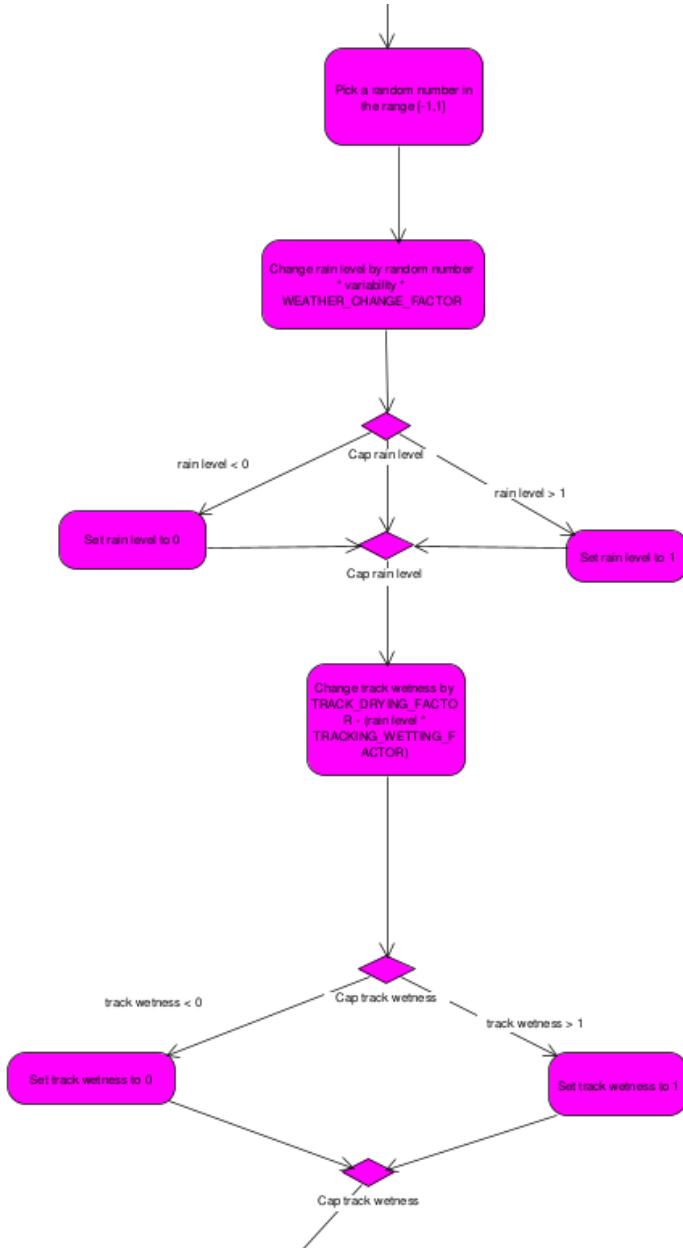


Figura 6: Secção do diagrama de atividades relativa à meteorologia

3.4 Desempenho

A simulação do desempenho dos carros é uma das áreas mais complexas do jogo e, por isso, o grupo não se quis comprometer muito com a sua especificação. Assim sendo, apenas definiu que a simulação de desempenho apenas acontece em campeonatos de administradores *premium*, sendo que todos os outros apenas observam as posições

relativas entre os carros.

De qualquer das formas, sempre que esse cálculo for a ser feito, deve ser realizado por um método que dependa da classe do carro, do estado do carro (pneus, dano, pacote aerodinâmico, etc) e das condições de pista. O desempenho não tem em conta a presença de outros competidores em pista.

Este fluxo encontra-se representado no diagrama da figura 7.

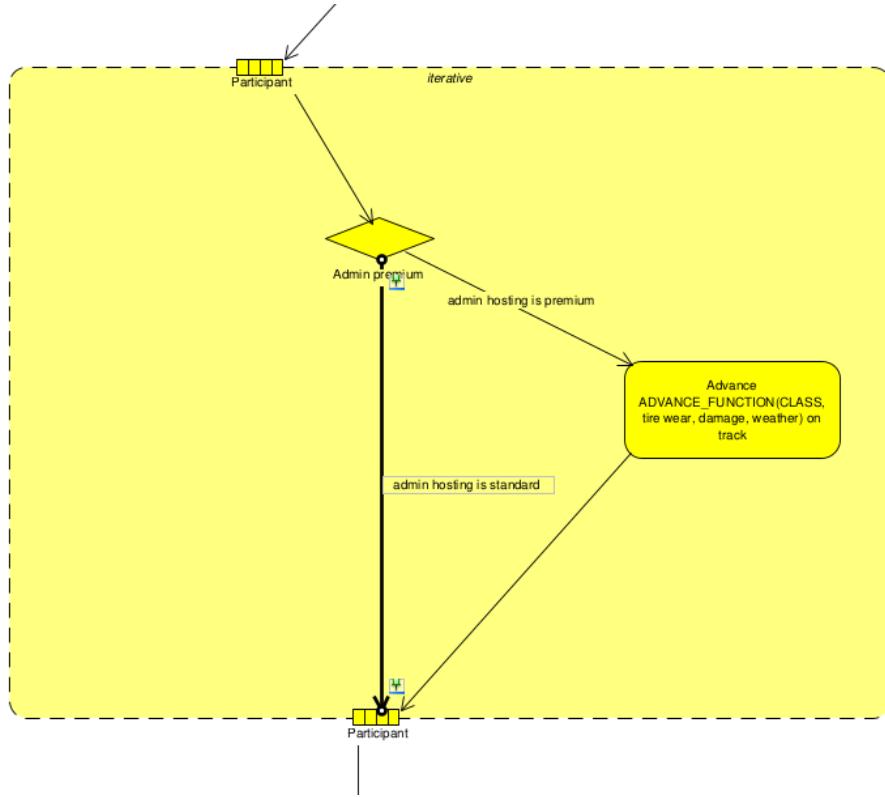


Figura 7: Secção do diagrama de atividades relativa ao desempenho dos carros

3.5 Erros

Os erros de piloto são a componente que traz aleatoriedade ao desempenho dos carros e dos pilotos. Definimos o erro como sendo a variação de desempenho dos pilotos relativamente à média - o valor esperado calculado segundo o modelado anteriormente. Isto significa que o erro, aqui, pode ter um valor negativo, isto é, um piloto está a exceder o ritmo natural do carro.

Para simular erros mais graves, que podem resultar em danos para o carro ou até o abandono da corrida, definiu-se um limite de tempo perdido a partir do qual se considera um acidente. Acima desse limite está o limite de abandono, ou seja, se o piloto perder mais do que esse valor num acidente, fica automaticamente fora da corrida. Para valores de tempo perdido entre estes 2 limites, adiciona-se um valor de dano que cresce linearmente com o aumento de tempo perdido. Este dano é cumulativo entre acidentes. Se um carro tiver um valor de dano superior a 1, abandona a corrida de forma imediata.

O diagrama de atividades da figura 8 reflete esta descrição.

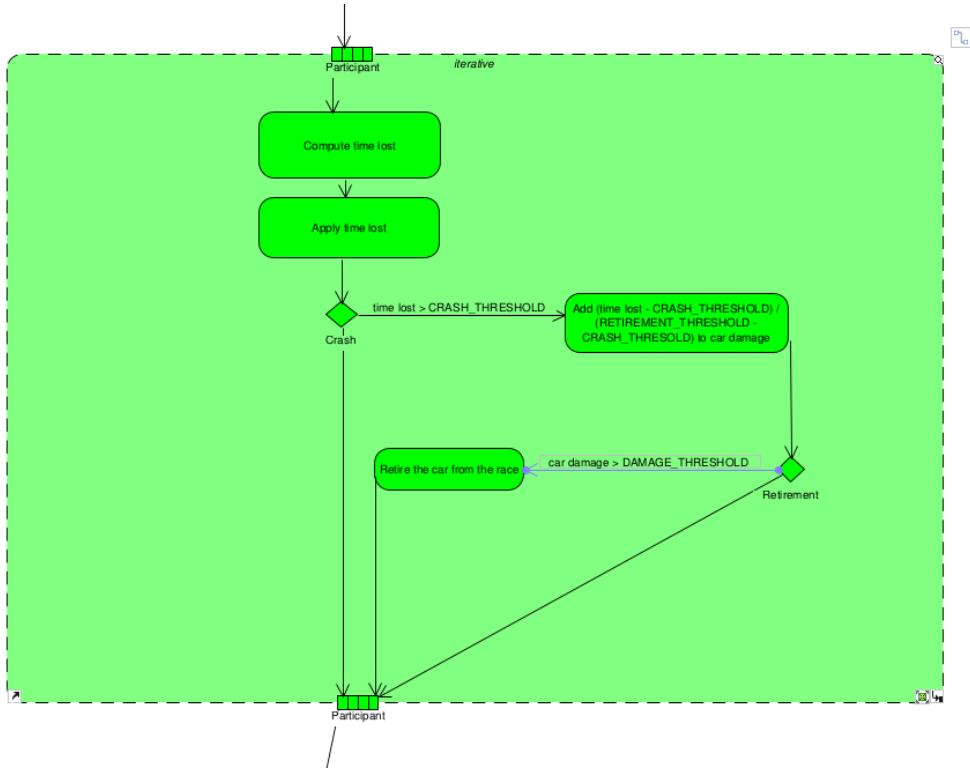


Figura 8: Secção do diagrama de atividades relativa aos erros dos pilotos

3.6 Ultrapassagens

A simulação de ultrapassagens é bastante complexa, pelo que a condição em que os pilotos se ultrapassam não foi totalmente especificada nesta fase. Define-se ultrapassagem como a alteração das suas posições físicas em pistas. Como a simulação varia conforme o administrador que opera o campeonato é um utilizador *premium* ou não, é necessário adicionar uma condição que permita distinguir estes dois casos.

A figura 9 mostra a modelação feita para as ultrapassagens.

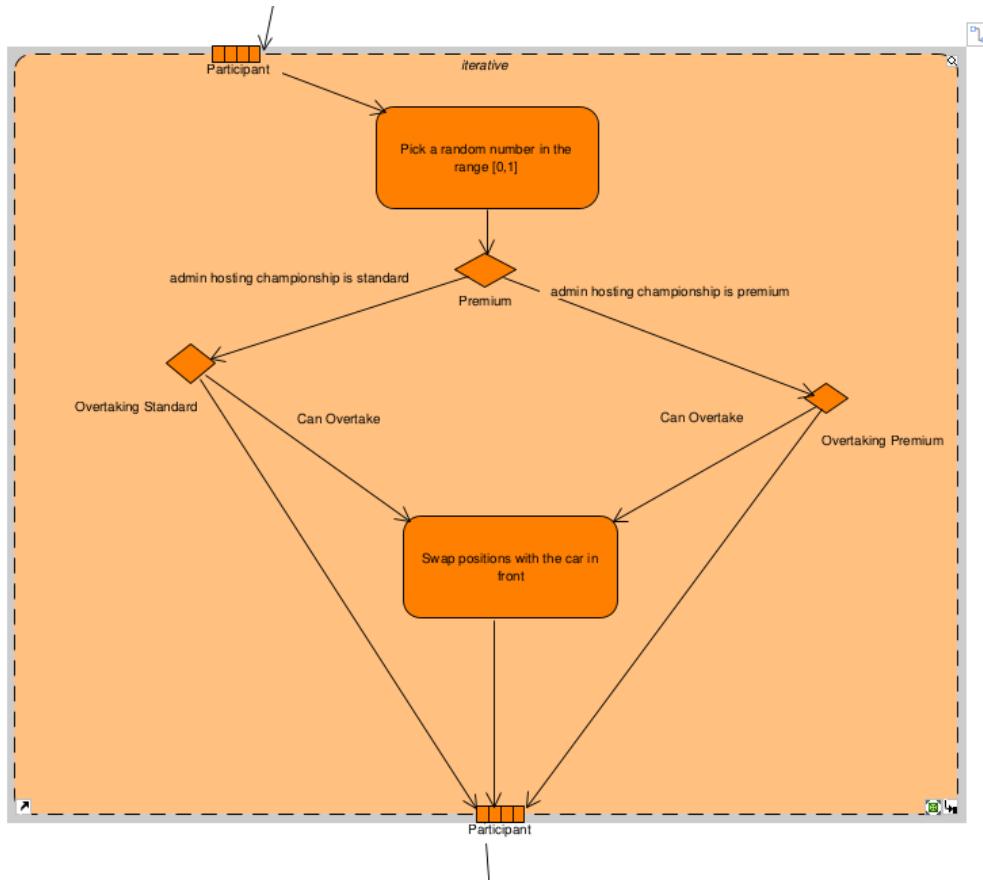


Figura 9: Secção do diagrama de atividades relativa às ultrapassagens

4 Conclusão

Com a segunda fase do projeto concluída, o grupo considera que o trabalho desenvolvido constituirá uma base sólida para a próxima e derradeira fase do projeto.

A não modelação de alguns sistemas poderá tornar a sua implementação mais difícil, embora tenha auxiliado no cumprimento do prazo para a fase que agora termina.

A modelação da simulação sob a forma de um diagrama de atividades tornará a sua implementação mais linear, o que será vantajoso para o grupo.

5 Anexos

Como mencionado anteriormente, o ficheiro do *Visual Paradigm* contendo estes diagramas foi enviado juntamente com este relatório.

5.1 Diagrama de Componentes

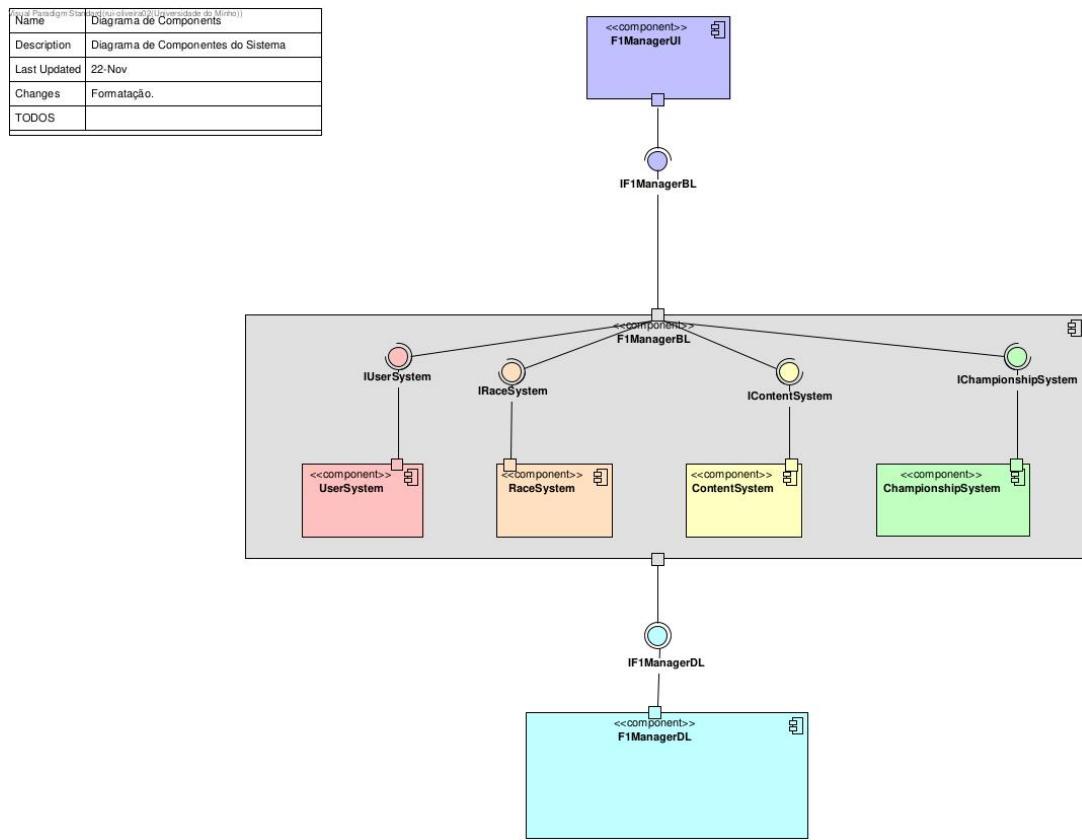


Figura 10: Diagrama de Componentes

5.2 Diagrama de Classes

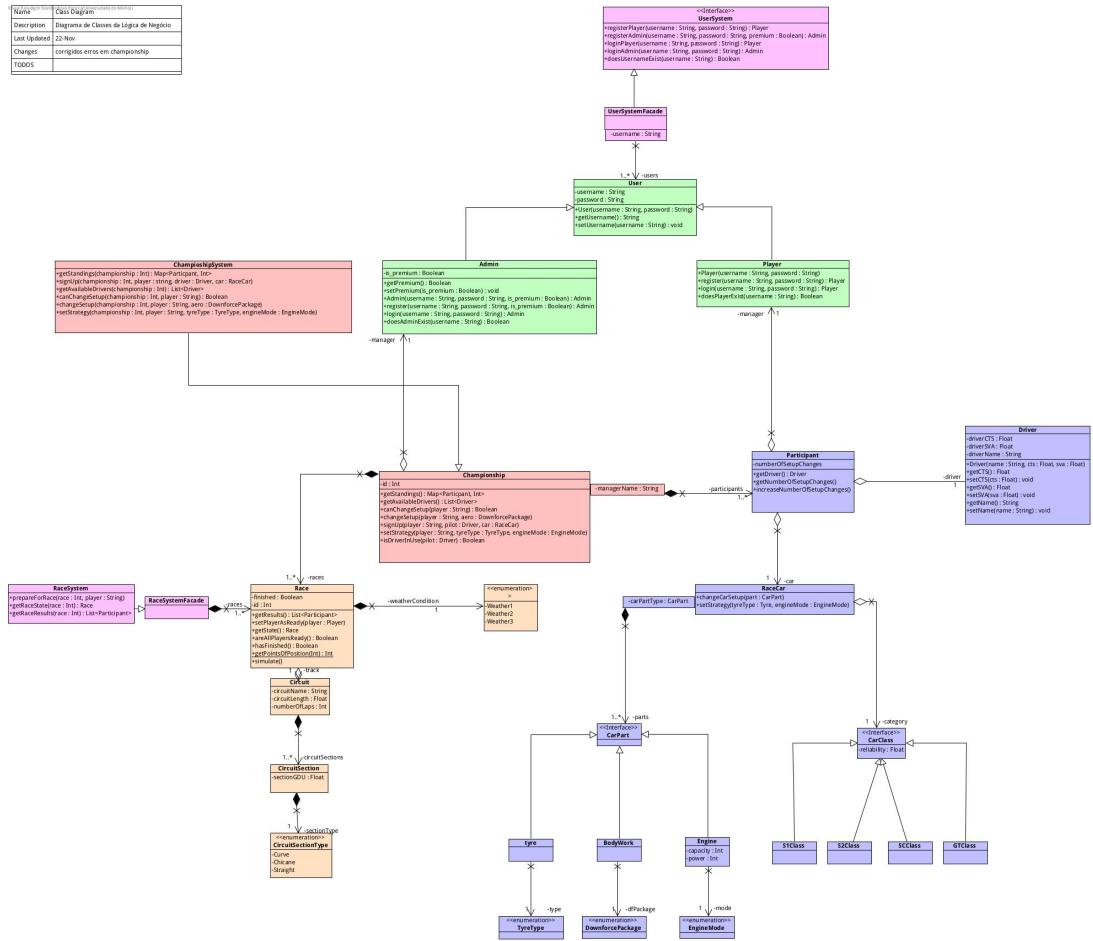


Figura 11: Diagrama de Classes

5.3 Diagramas de Sequência

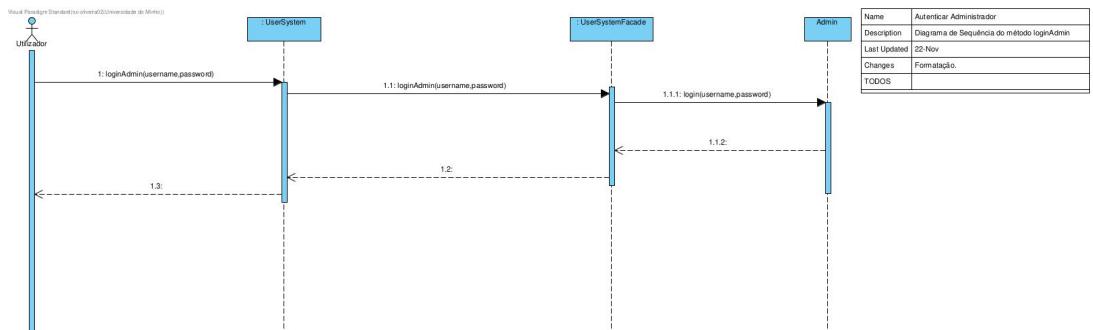


Figura 12: Diagrama de Sequência do Método `loginAdmin`

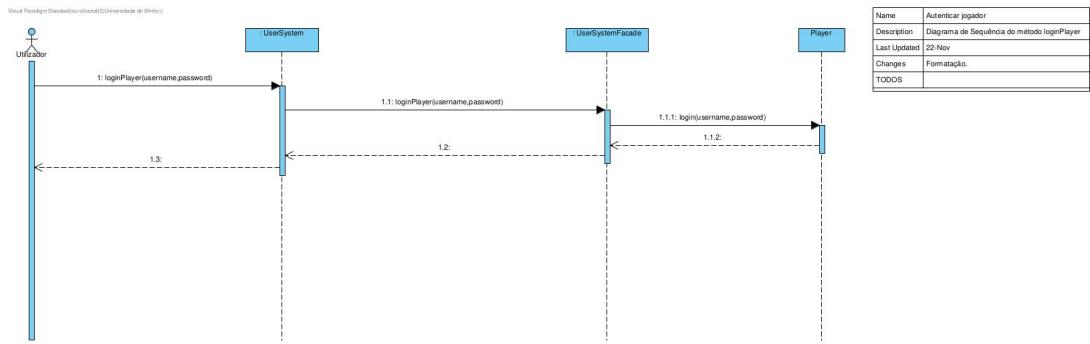


Figura 13: Diagrama de Sequência do Método *loginPlayer*

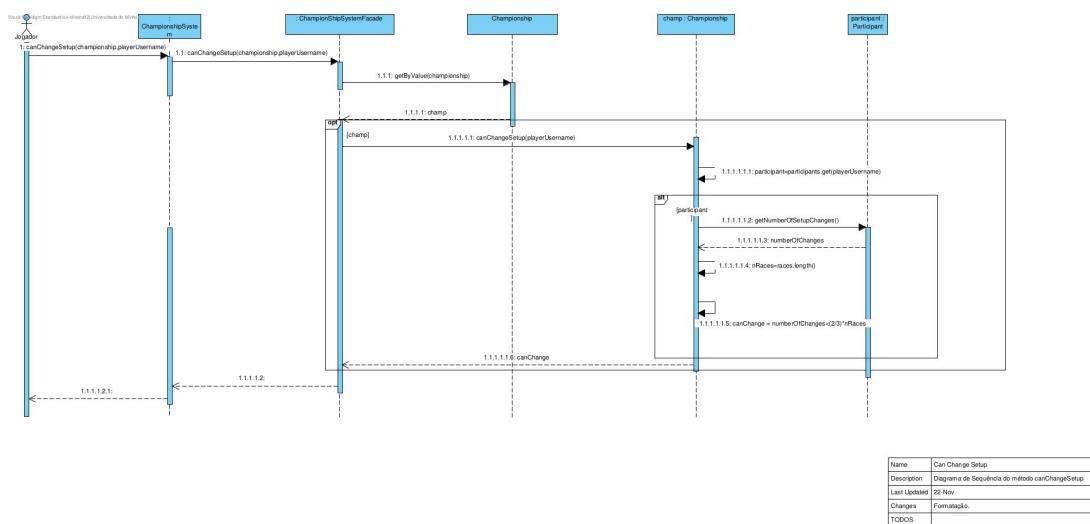


Figura 14: Diagrama de Sequência do Método *canChangeSetup*

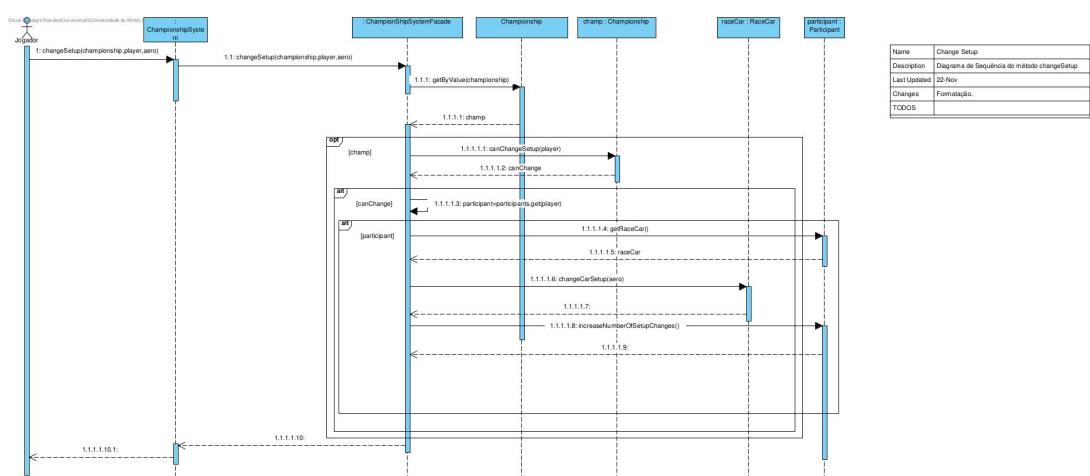


Figura 15: Diagrama de Sequência do Método *changeSetup*

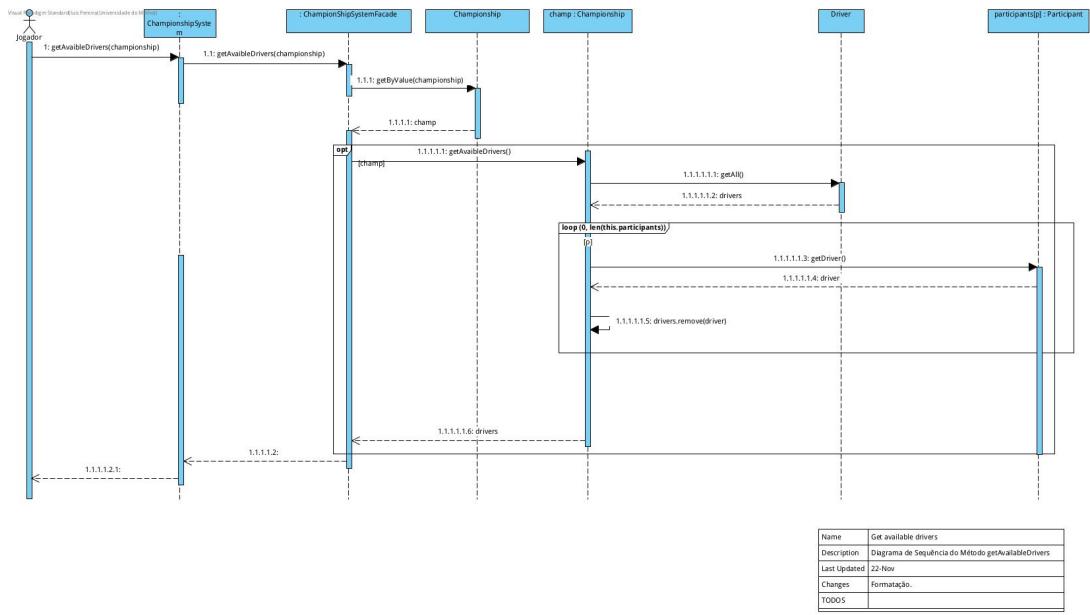


Figura 16: Diagrama de Sequência do Método *getAvailableDrivers*

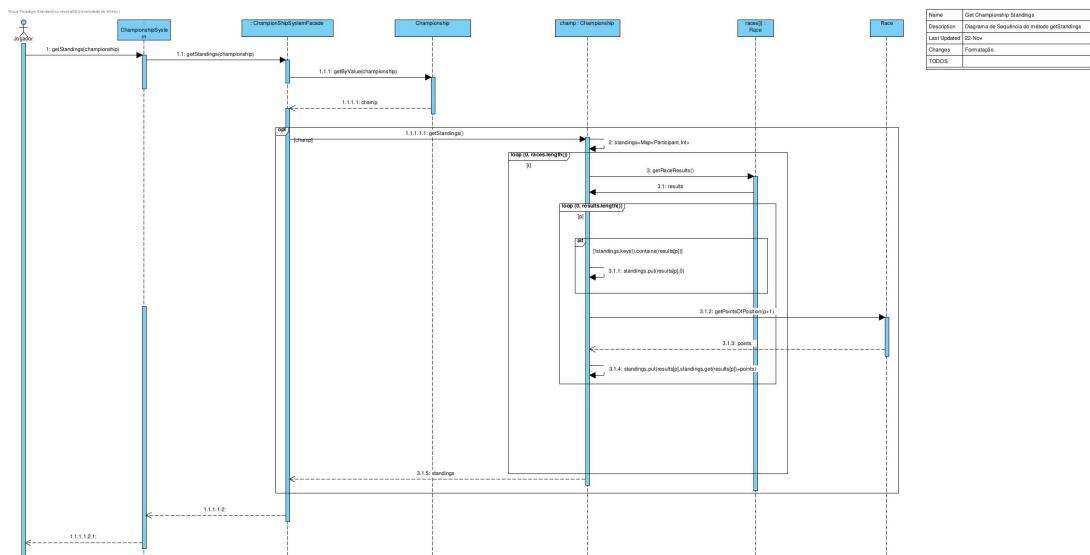


Figura 17: Diagrama de Sequência do Método *getStandings*

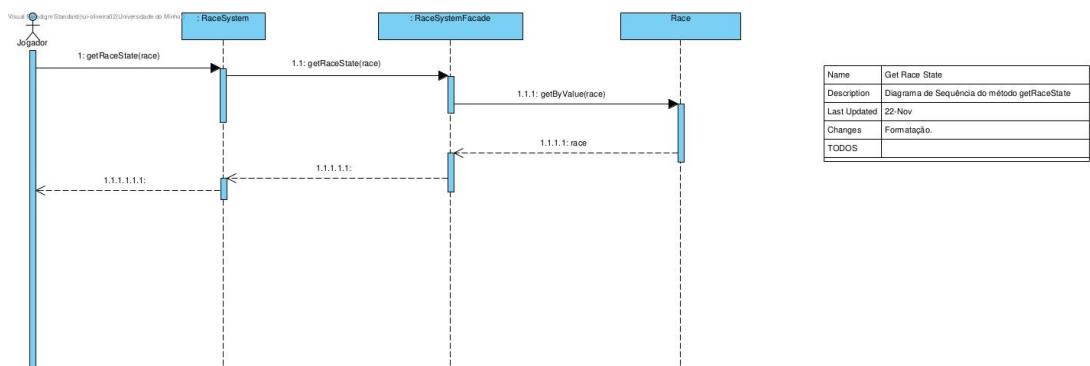


Figura 18: Diagrama de Sequência do Método *getRaceState*

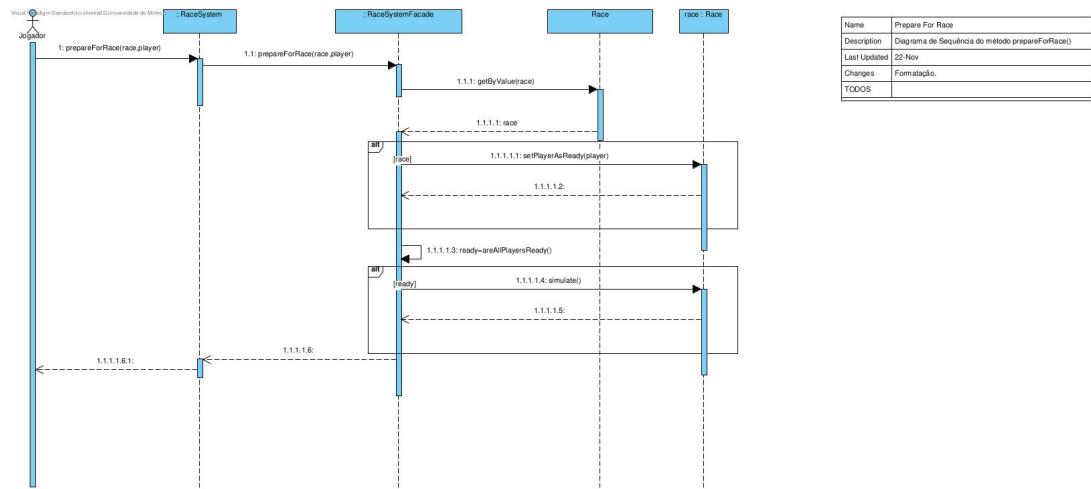


Figura 19: Diagrama de Sequência do Método *prepareForRace*

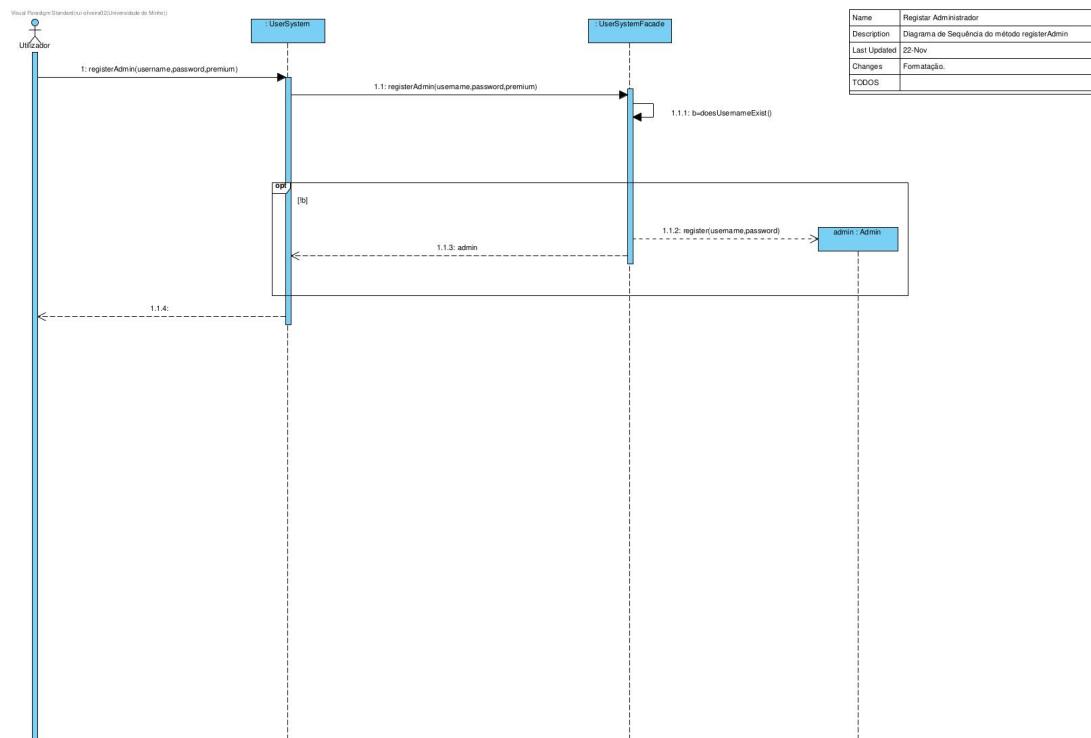


Figura 20: Diagrama de Sequência do Método *registerAdmin*

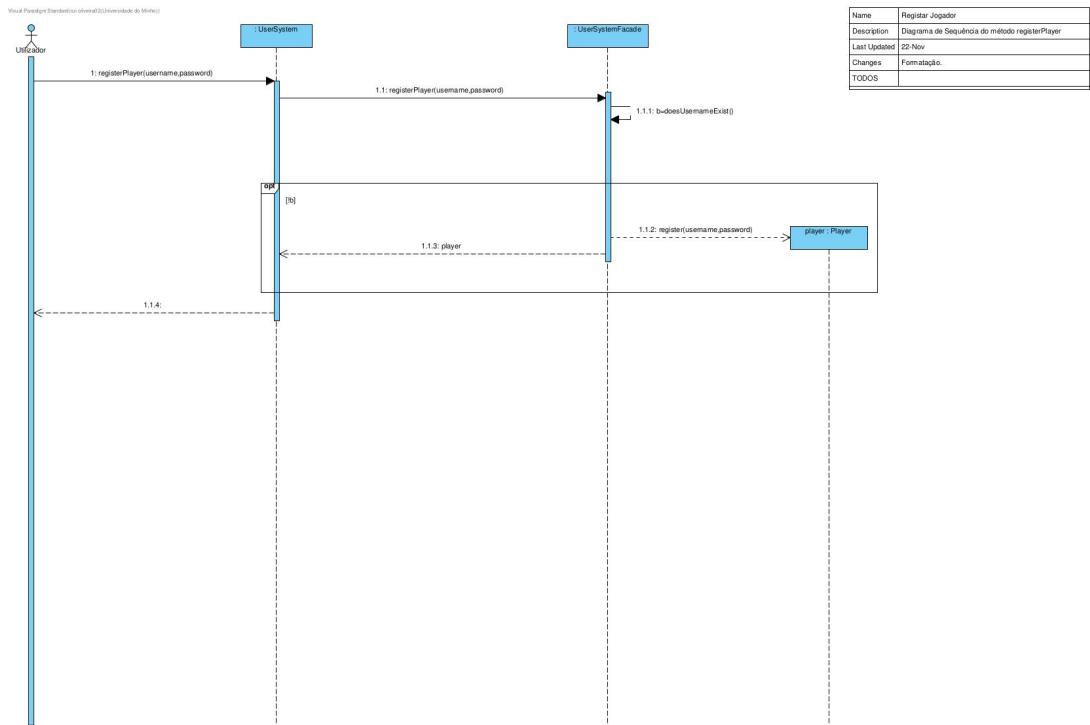


Figura 21: Diagrama de Sequência do Método *registerPlayer*

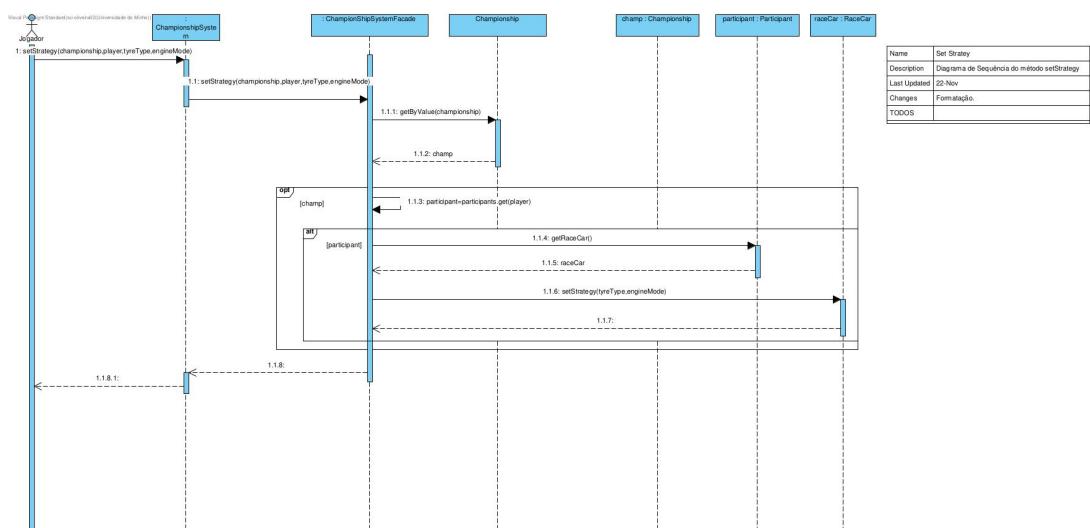


Figura 22: Diagrama de Sequência do Método *setStrategy*

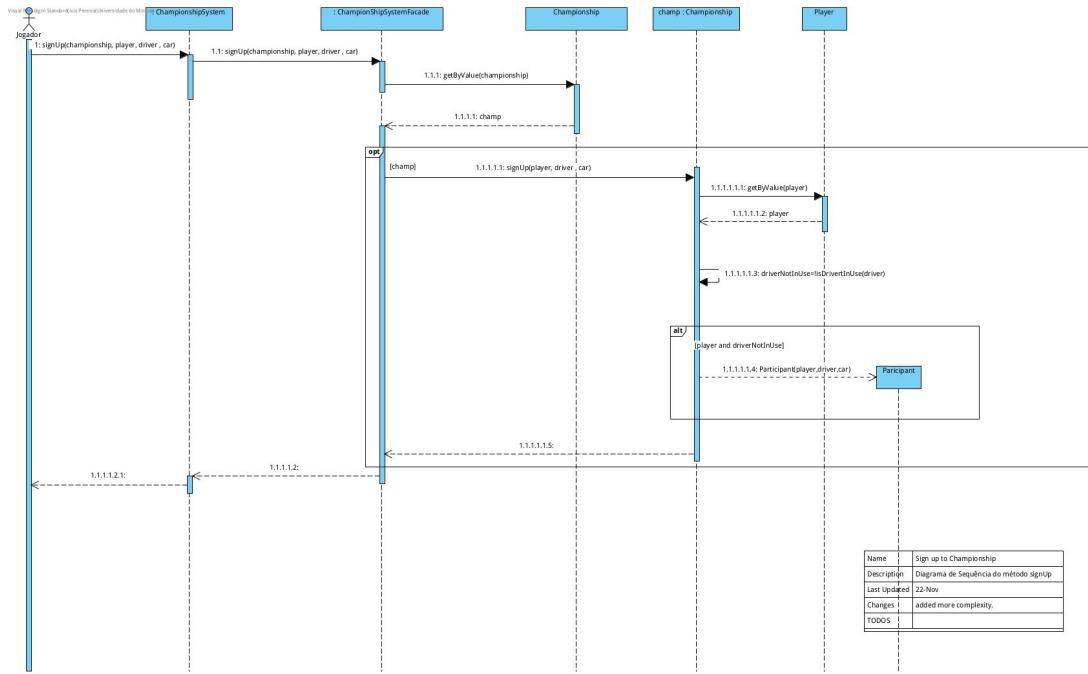


Figura 23: Diagrama de Sequência do Método *signUp*

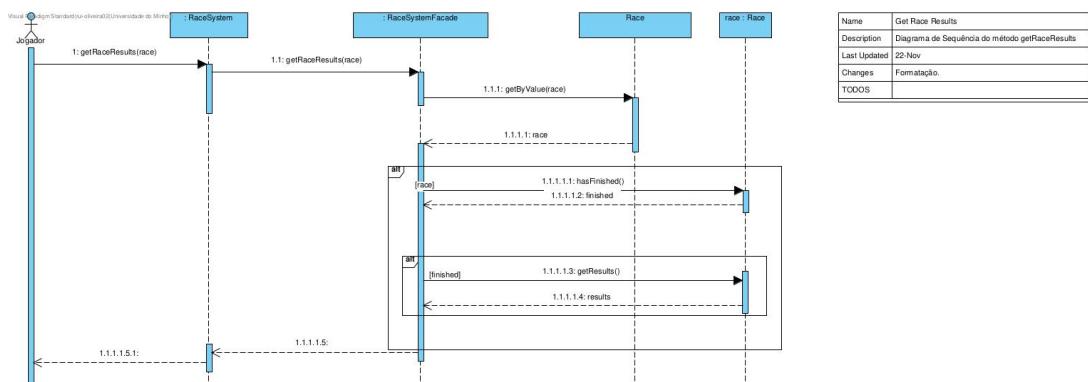


Figura 24: Diagrama de Sequência do Método *getRaceResults*

5.4 Diagrama de Atividades

Este diagrama foi partido em 2 imagens devido à sua altura, que impossibilitava que este coubesse numa só página.

Visual Procedure Description (VPD) (Universidade de Minas)	
Name	Diagrama de Atividade da Simulação
Description	Diagrama de atividade relativo à simulação de corridas
Last Updated	22-Nov
Changes	Fórmula na fiabilidade.
TODOs	

Everything in LIPPER_CASE means that it is a constant whose value currently unknown and will be adjusted in the implementation step to improve gameplay.

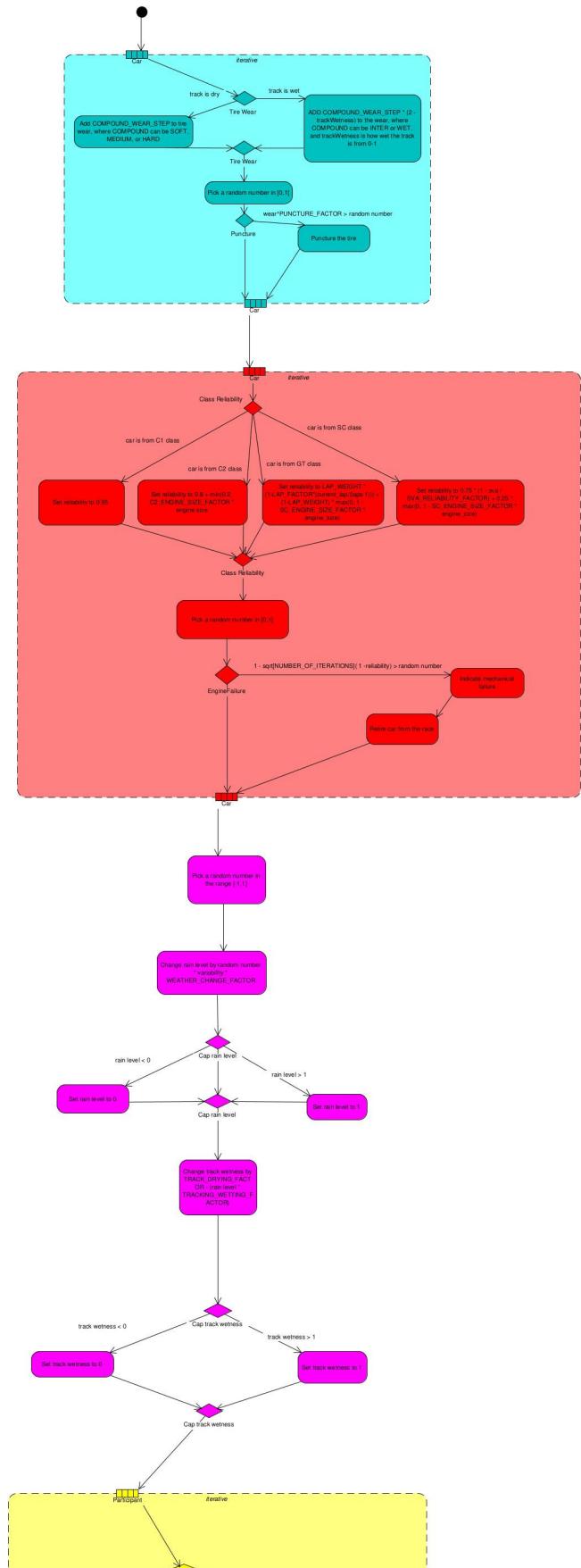


Figura 25: Diagrama de Atividades da Simulação de Corridas (parte 1)

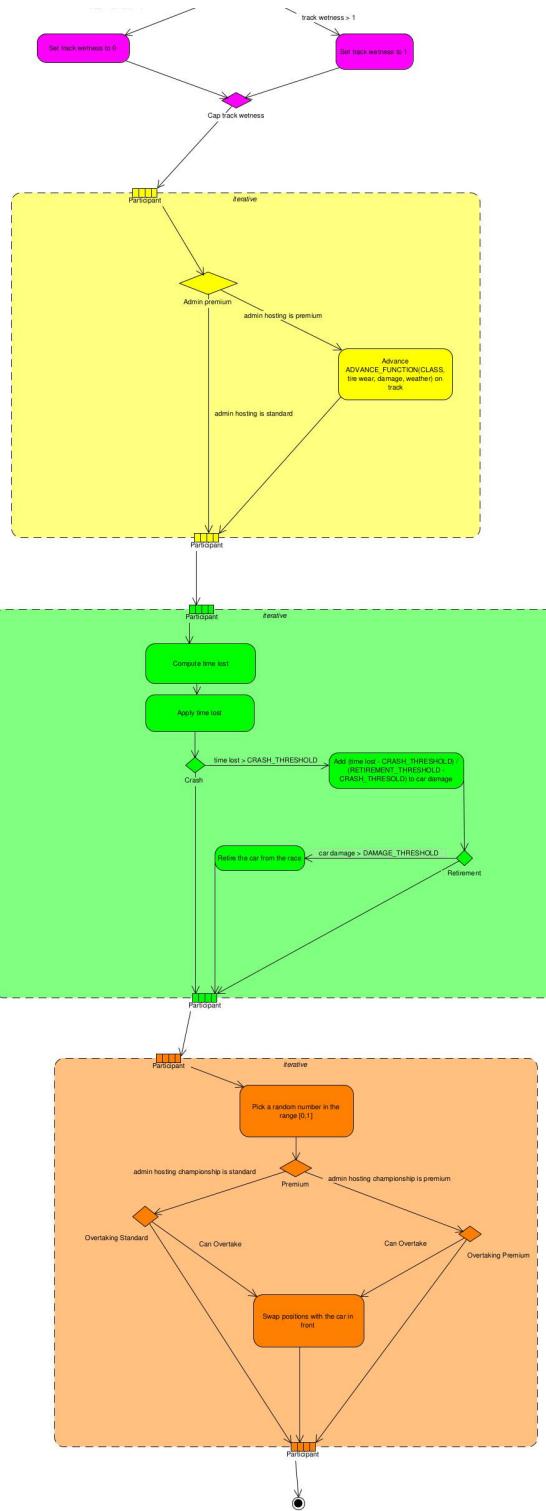


Figura 26: Diagrama de Atividades da Simulação de Corridas (parte 2)