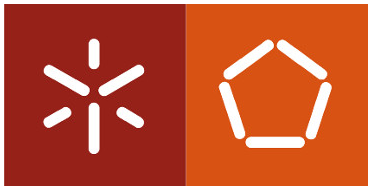


UNIVERSIDADE DO MINHO

DEPARTAMENTO DE INFORMÁTICA

Trabalho Prático - Fase 2

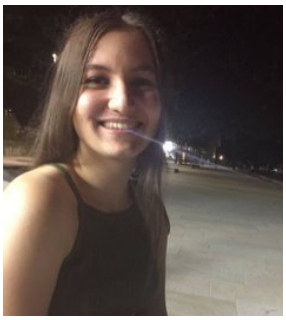
Desenvolvimento de Sistemas de Software

Grupo 19

Ana Filipa Pereira (A89589) Carolina Santejo (A89500)

Raquel Costa (A89464)

28 de Novembro de 2020



(a) Ana Filipa Pereira

(b) Carolina Santejo

(c) Raquel Costa

Conteúdo

1 Introdução

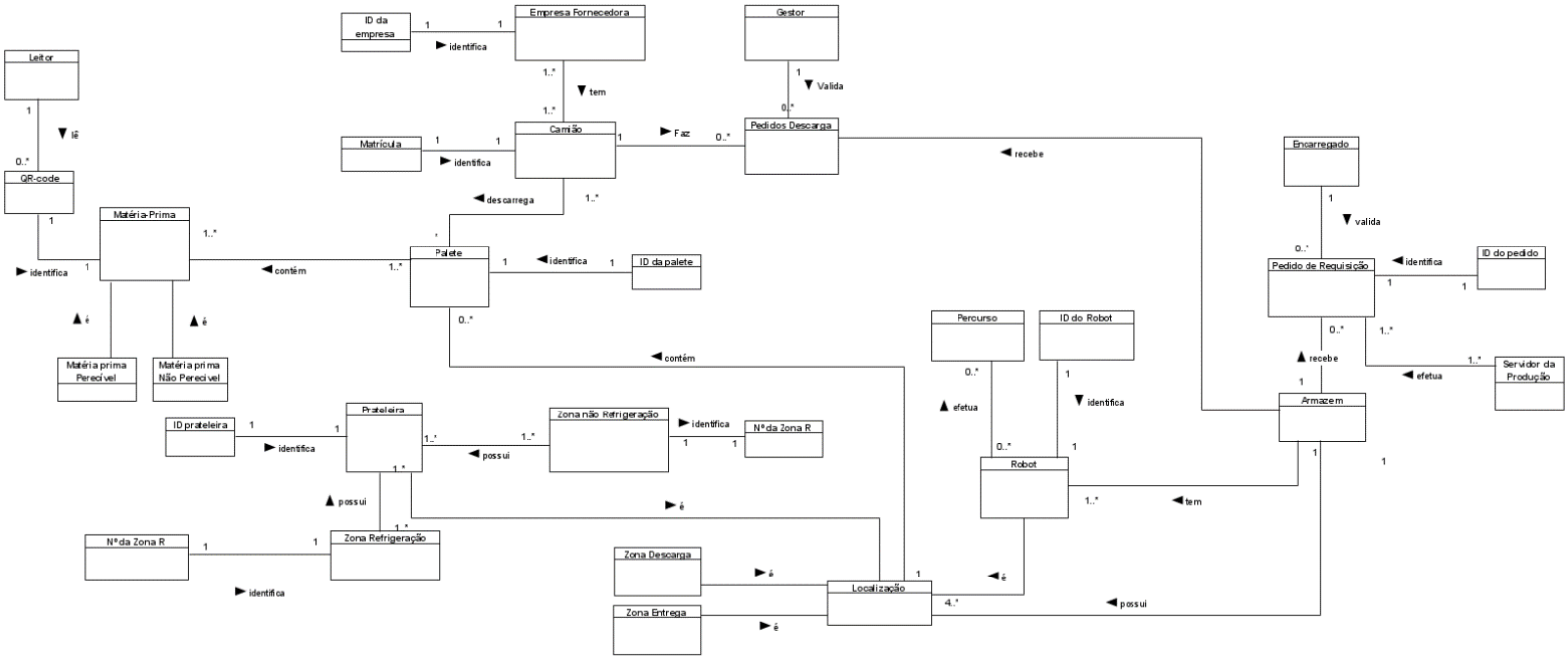
Esta segunda fase do projeto da cadeira de Desenvolvimento de Sistemas de Software, consistiu no desenvolvimento de um modelo conceptual que seja capaz de suportar os Use Cases definidos na fase anterior. Visto que todo o processo é incremental e iterativo, esta fase consistiu numa continuação da primeira, 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 Principais Objetivos

Ao longo desta segunda fase foram-nos apresentados vários desafios. Numa fase 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. Após isto, criámos o diagrama de componentes no qual foram representados os vários subsistemas que o grupo considerou apropriados. Por último, desenvolvemos o diagrama de Packages. 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



Após uma analise detalhada do modelo de domínio inicialmente proposto, o grupo decidiu que seria necessário redefinir algumas entidades e associações que, graças aos conhecimentos adquiridos foi possível detetar. Primeiramente, foi retirada a entidade sistema que tínhamos enunciado ser a entidade mais importante do modelo, mas neste caso não o vai ser porque o objetivo é representar as entidades presentes dentro do sistema. No caso das associações, foi alterada a direção da 'é' referente à localização e, por outro lado, passamos a associar o QR-Code com matéria prima em vez de palete pois foi considerado que este código identifica o produto incluído na palete. Além disso, foi retirada também, a ligação entre as zonas de refrigeração e não refrigeração com as matérias primas, pois estas já estavam relacionadas através da entidade prateleira. Foi acrescentada também uma

associação Localização ‘contém’ palete e retirada a que relacionava Robot com Palete porque seria redundante deixar ambas. A frota, que inicialmente incluía um conjunto de Robots , foi eliminada pois não acrescentaria informação ao modelo. Por último, os pedidos de descarga e requisição anteriormente ligados ao sistema, associam-se agora ao armazém pois é lá que vão ser avaliados.

3.2 Modelo de Use Cases

Figura 2: Modelo de Use Cases Atualizado

Nesta nova fase do projeto foi pedido para considerar os Use Cases pintados a verde. Para tal, comparando com o nosso modelo da fase anterior, vemos que houve uma substituição do Use Case "Efetuar Autenticação"por dois novos Use Cases : o "Inicia Sessão"e o "Termina Sessão". Foi ponderado continuar com o "Efetuar Autenticação", sendo que o logout estaria implícito sempre que o utilizador fechava a aplicação, mas depois chegamos à conclusão que caso um utilizador

tivesse de fechar a aplicação muitas vezes, seria aborrecido em termos de praticalidade para o próprio User, tornando-se assim um processo pouco eficiente.

Além disso, ainda adicionamos o Use Case "Consulta listagem de localizações". Na fase passada, o grupo considerava que o gestor acedia à listagem no Use Case "Valida Pedidos de Descarga", após os conhecimentos adquiridos e, consequentemente, uma nova perspetiva, vemos que realmente foi uma lacuna nossa e que realmente trata-se de uma funcionalidade que o sistema deverá ter, daí a necessidade deste processo ser considerado como um Use Case.

As especificações dos Use Cases referidos são as seguintes:

3.3 Consultar listagem de localização

- **Ator:** Gestor.
- **Pré-Condição:** Gestor tem de estar autenticado.
- **Pós-Condição:** Sistema disponibiliza lista.
- **Fluxo normal:**
    1. Gestor pede listagem de localização ao Sistema.
    2. Sistema disponibiliza a listagem.

3.4 Terminar Sessão

- **Ator:** Gestor e Encarregado.
- **Pré-Condição:** Ator tem de estar autenticado.
- **Pós-Condição:** Sistema volta para a interface de Efetuar Login.
- **Fluxo normal:**
    1. Ator solicita logout do sistema.
    2. Sistema termina sessão do ator.

3.5 Iniciar Sessão

No caso deste Use Case consideramos a mesma especificação feita na 1ªFase do projeto para o Use Case "Efetuar Autenticação".

4 Construção da API da Lógica de Negócio

4.1 Identificação de Responsabilidades

Através da análise dos Use Cases determinados na primeira fase e daqueles que tiveram de ser adicionados nesta fase 2, foram levantadas as seguintes responsabilidades:

- Validar acesso de um utilizador
- Efetuar o logout de um utilizador
- Disponibilizar Listagem
- Registar paletes
- Atualizar localização de uma palete
- Validar pedido de requisição
- Registar pedido de requisição
- Registar paletes em falta
- Aceitar pedido de requisição
- Obter um robot disponível
- Notificar transporte de palete
- Atualizar estado do robot
- Indicar destino do percurso ao robot
- Calcular uma rota
- Validar rota
- Adicionar rota
- Cancelar uma rota

4.2 API da Lógica de Negócio

Após termos levantado as responsabilidades do sistema, passamos para a definição da API da lógica de negócio. Os métodos anteriores permitiram elaborar os seguintes métodos, respeitando a ordem acima usada:

- validaAcesso(s:String,password:String):boolean
- logout(codUser:String)
- disponibilizaListagem(): Listagem
- registarPaletes(codQR : String, codPalete : String, peso : float ,loc:Localizacao)
- atualizaLocalizacaoPalete(codPalete : String, locPalete : Localizacao)
- validaPedidoRequisicao(requisicaoP : Requisição) :List<Palete>
- registaPedidoRequisicao(requisicaoP : Requisição)
- registaPaletesFalta(pFalta : List<String>)
- aceitaPedidoRequisicao(estado : int)
- getRobotDisponivel() : String
- notificaTransporte(codRobot : String, codPalete : String)
- atualizaEstadoRobot(codRobot : String, estado : int)
- indicaDestino(codRobot : String, destino : Localizacao)
- calculaRota(codRobot : String) : List<Aresta>
- validarRota(rota : List< Aresta>) : boolean
- adicionaRota(codRobot : String, rota : List< Aresta>)
- cancelaRota(codRobot : String)

4.3 Identificação de Subsistemas

Quando se levantou as responsabilidades do sistema, o grupo decidiu utilizar três subsistemas: GestPaletes que trata da gestão de paletes, GestRobots que trata da gestão de robots e o Autenticação que serve para o acesso á API do sistema. Desta forma foi feio o seguinte agrupamento de métodos:

- validaAcesso(s:String,password:String):boolean -> Autenticacao
- logout(codUser:String) -> Autenticacao
- disponibilizaListagem(): Listagem -> GestPaletes
- registarPaletes(codQR : String, codPalete : String, peso : float ,loc:Localizacao) -> GestPaletes
- atualizaLocalizacaoPalete(codPalete : String, locPalete : Localizacao) -> GestPaletes
- validaPedidoRequisicao(requisicaoP : Requisição) :List<Palete> -> GestPaletes
- registaPedidoRequisicao(requisicaoP : Requisição) -> GestPaletes
- registaPaletesFalta(pFalta : List<String>) -> GestPaletes
- aceitaPedidoRequisicao(estado : int) -> GestPaletes
- getRobotDisponivel() : String -> GestRobots
- notificaTransporte(codRobot : String, codPalete : String) -> GestRobots
- atualizaEstadoRobot(codRobot : String, estado : int) -> GestRobots
- indicaDestino(codRobot : String, destino : Localizacao) -> GestRobots
- calculaRota(codRobot : String) : List<Aresta> -> GestRobots
- validarRota(rota : List< Aresta>) : boolean -> GestRobots
- adicionaRota(codRobot : String, rota : List< Aresta>) -> GestRobots
- cancelaRota(codRobot : String) -> GestRobots

5 Diagrama de Componentes

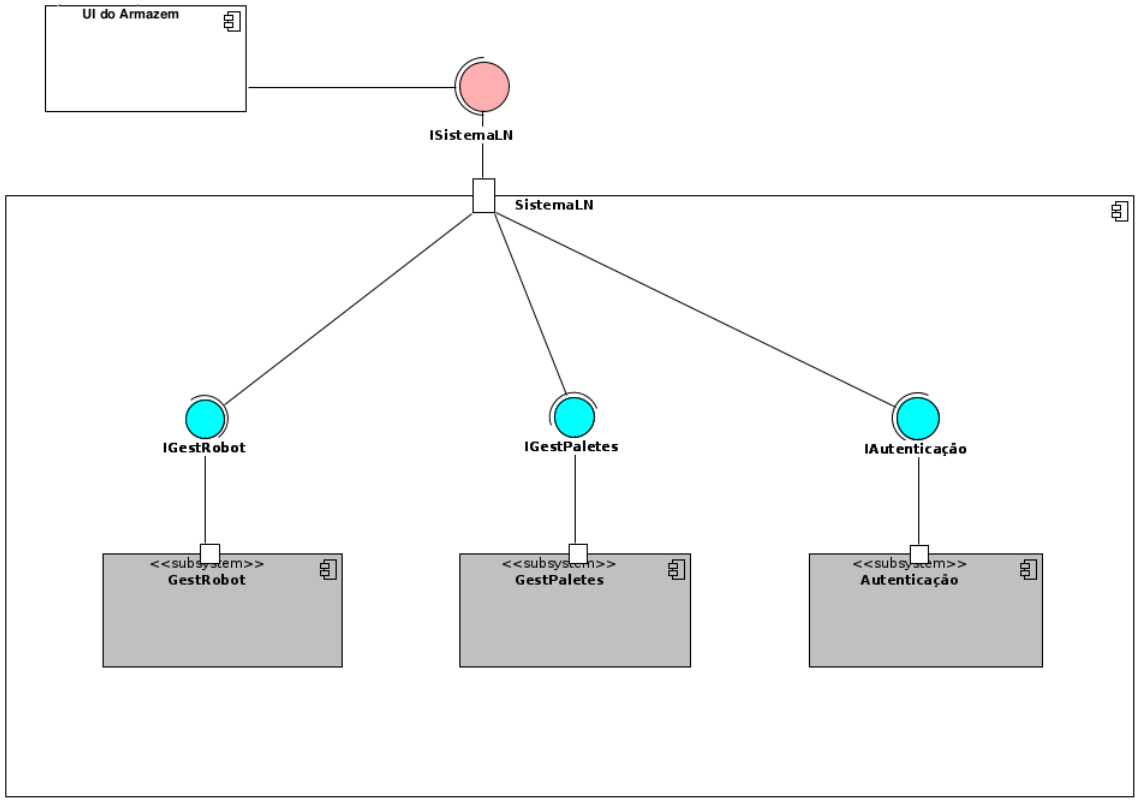


Figura 3: Diagrama de Componentes

Após o levantamento das responsabilidades do sistema e a definição dos respetivos métodos (API da Lógica de Negócio), foi necessária a determinação dos subsistemas que seriam usados no trabalho e que estariam representados no nosso Diagrama de Componentes. Isto foi necessário pois a criação de subsistemas permite uma maior organização, uma vez que haverá o agrupamento dos métodos pelos subsistemas onde melhor se adequam.

Além disto, esta prática é fundamental na questão do encapsulamento uma vez que cada subsistema implementa uma interface com os métodos que lhes foram atribuídos, impedindo isto o acesso ao seu "core" (o subsistema poderá conter métodos que não estão na interface e que não convém que "entidades" exteriores lhe tenham acesso). Inicialmente o grupo pensou em 4 subsistemas: um relativo

á gestão de paletes, um relativo á autenticação dos funcionários, outro relativo á gestão dos robots e por último um relativo às requisições. No entanto, reparámos que os subsistemas da gestão paletes e das requisições tinham coisas em comum, pelo que se decidiu que tudo o que fosse relativo a requisições seria tratado na gestão de paletes.

Deste modo, ficaram definidos os seguintes subsistemas: GestRobot que implementa a interface IGestRobot, GestPaletes que implementa a interface IGestPaletes e a Autenticação que implementa a interface IAutenticação. Por sua vez, o Armazém User Interface (UI do Armazém) acede às funcionalidades do sistema através de uma outra interface, ISistemaLN, que implementa as interfaces dos já referidos subsistemas.

6 Diagramas de Classes

6.1 GestPaletes

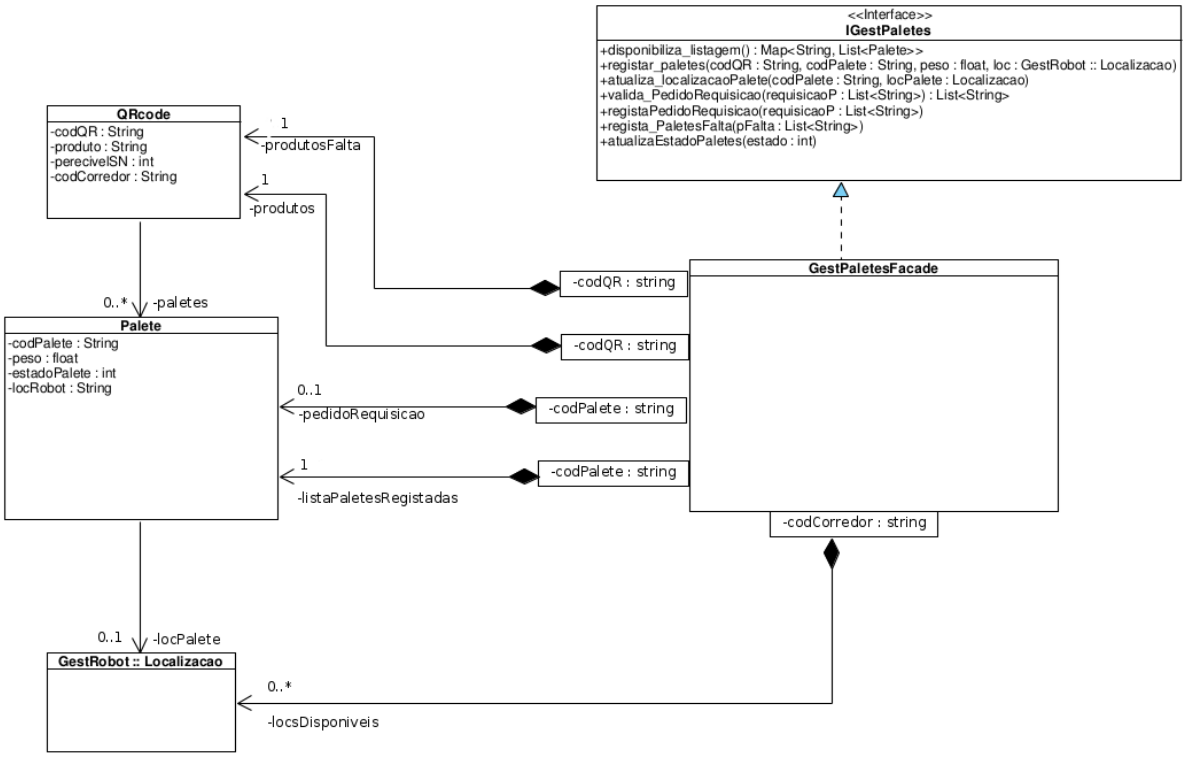


Figura 4: Diagrama de Classes - GestPaletes

Neste subsistema é onde nós vamos incluir os métodos relacionados com o gerenciamento das paletes presentes no armazém, bem como quais as matérias-primas que o armazém tem, por exemplo. Existem 3 classes além do Facade do Subsistema : QRcode, Palete , GestRobot:: Localizacação.

A classe QRcode está relacionada com o facade através de dois Maps. O *produtosFalta* trata-se do registo que irá ser criado caso numa requisição feita pelo Servidor haja paletes em falta e este queira proceder ao registo delas para mais tarde recebe-las assim que o armazém for reabastecido. O map *produtos* é o map de todos os produtos que o armazém deverá guardar, qualquer outro produto com um QRCode além daqueles que estão armazenados neste Map, não

poderá entrar no Armazém. É importante referir que a Key de ambos os maps é o códigoQR e cada key tem um value QRCode.

O QRCode, além do seu código contém também a desiganção do produto que representa, se é perecível - 1 ou não - 0 , já que o tipo deste atributo é int, e ainda tem o código do corredor, isto é a aresta de um grafo que corresponde ao mapa do armazenamento, tal como iremos ver mais à frente. Cada corredor está associado a um produto, isto é, apenas irá armazenar nas suas prateleiras paletes que tenham o mesmo QRCode (sejam a mesma matéria-prima). Além disso, contém também uma lista das paletes que têm o QR Code em causa.

A classe Palete tem como atributos, o código **único** de cada palete, o peso, o estado (se está requisitada ou não) ,ainda tem também a locRobot (caso a palete esteja localizada num Robot) e a locPalete que consiste numa Localização (classe importada de outro package) que contém o corredor/aresta e prateleira caso a palete se encontre-se neste local específico, estes dois atributos podem ter valor null (mas não simultaneamente). No facade existem dois maps relacionados com a classe Palete. O *pedidoRequisicao* trata-se se um map cuja key é o código da palete e o value é uma Palete, este map pode ou não estar vazio, uma vez que caso não haja requisições feitas pelo servidor, o map estará vazio. Tendo em conta que o servidor faz pedidos através do QRCode, nós temos de procurar paletes disponíveis desse QRcode e fornecer o código da palete para responder ao pedido do servidor. Mas isto é algo que iremos abordar mais à frente. Também temos o Map *listaPaletesRegistadas* , onde armazenamos todas as paletes presentes no armazém e que foram registadas à entrada.

Em relacação à classe importada ,Localizacao do package *GestRobot*, está ligada ao facade através de um Map onde a key é o código do corredor de forma a que o gestor saiba quais as prateleiras disponíveis para armazenar uma palete. É importante referir que no caso das zonas de descarga e entrega, consideramos sempre como disponiveis e que aqui não existem prateleiras.

6.2 GestRobot

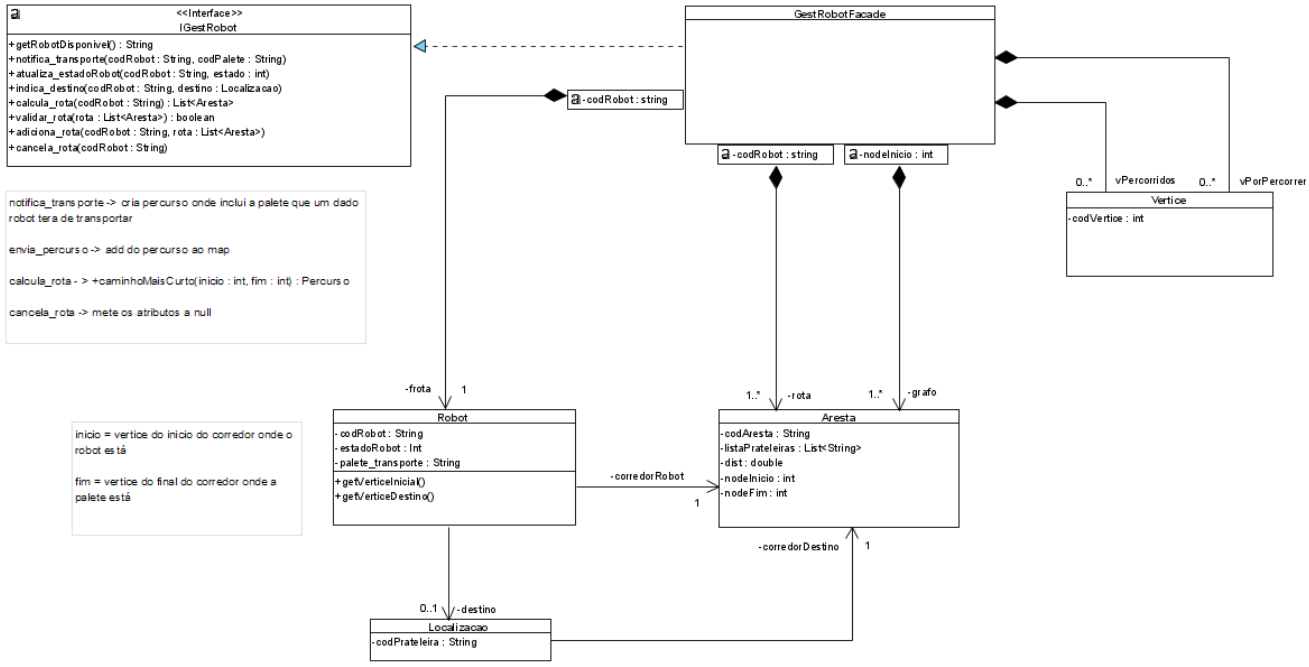


Figura 5: Diagrama de Classes - GestRobot

É neste subsistema onde armazenamos todos os métodos e classes relacionadas com o gerenciamento de Robots e consequentemente Percursos a serem tomados pelos mesmos.

Para começar temos 3 classes além do Facade. A classe Robot está conectada ao facade através de um Map com o código do respetivo Robot, a este damos o nome de frota. Esta classe contém como atributos o código do Robot, o estado do Robot, i.e, se este está disponível ou não, a palete que este deverá transportar (caso não tenha de transportar, este atributo fica a null), a Aresta/Corredor onde o Robot se encontra e, ainda, a Localização do seu destino (caso não tenha sido atribuído um destino, este atributo encontra-se a null também). Além disso, ainda definimos dois métodos que irão retornar um vértice cada um, tal como diz na Nota ao lado do diagrama na Figura 5.

Em relação à classe Localização tal como vimos esta inclui o código de uma prateleira e o corredor onde essa se encontra. Foi considerado que cada corredor

corresponde a uma aresta do grafo que representa o mapa do Armazém.

A classe Aresta contém o código dessa aresta/corredor, a lista de prateleiras nesse mesmo corredor, a distancia entre os seus dois vértices, representados por dois inteiros (nodeInicio, nodeFim). Esta classe está ligada ao facade através de dois Maps : rota e grafo. O map rota contém todas as arestas que um dado Robot irá percorrer, resultante do cálculo de um percurso. O map grafo tem todas as arestas e, consequentemente, vértices que o grafo representante do mapa irá ter.

Numa fase tardia da elaboração desta parte do projeto, implementamos também duas listas, que irão auxiliar no cálculo do percurso mais curto para um dado Robot. Para tal, o nosso algortimo consiste em começar com a lista dos percorridos vazia e a dos por percorrer com os vértices todos. Escolhemos um vértice inicial, vemos as arestas ligadas a este e guardamos as respetivas distâncias num atributo do facade , ou então futuramente numa classe ainda por implementar, de seguida, adicionamos este vértice inicial à lista de vertices percorridos e retiramos da lista dos PorPercorrer. Depois escolhemos o vértice ligado ao anterior que tem a menor distancia , e voltamos a repetir o processo descrito anteriormente, somando sempre a distancia a cada vertice. No final iremos obter como resultado todas as distancias mais curtas desde o vértice que consideramos inicialmente até cada um dos outros existentes.

6.3 Autenticação

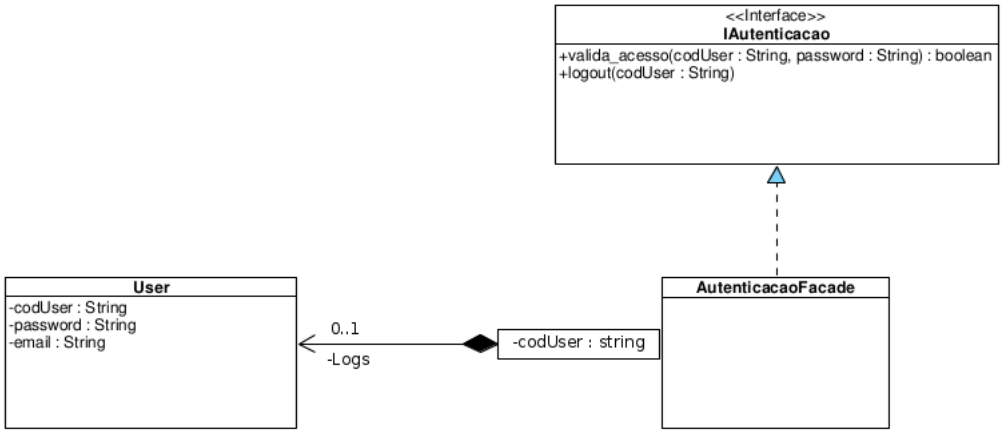


Figura 6: Diagrama de Classes - Autenticação

Este diagrama de classes corresponde ao subsistema Autenticação, que foi criado de forma a conter todos os métodos necessários ao processo de autenticação do gestor e do encarregado. A classe *AutenticacaoFacade* importa a interface IAutenticacao cujos métodos definidos são o *logout()*, que permite ao utilizador sair do sistema, e o *validoAcesso()*, que permite o acesso ao sistema por parte de um utilizador caso as suas credenciais (código e password) sejam válidas.

No facade, existe uma estrutura map, denominada logs, através do qual é possível aceder às informações de um user, através do seu código. É importante referir que consideramos que o código do user irá conter um caracter diferenciador que irá permitir saber se este user trata-se de um gestor ou servidor.

7 Diagramas de Sequência

7.1 DisponibilizaListagem

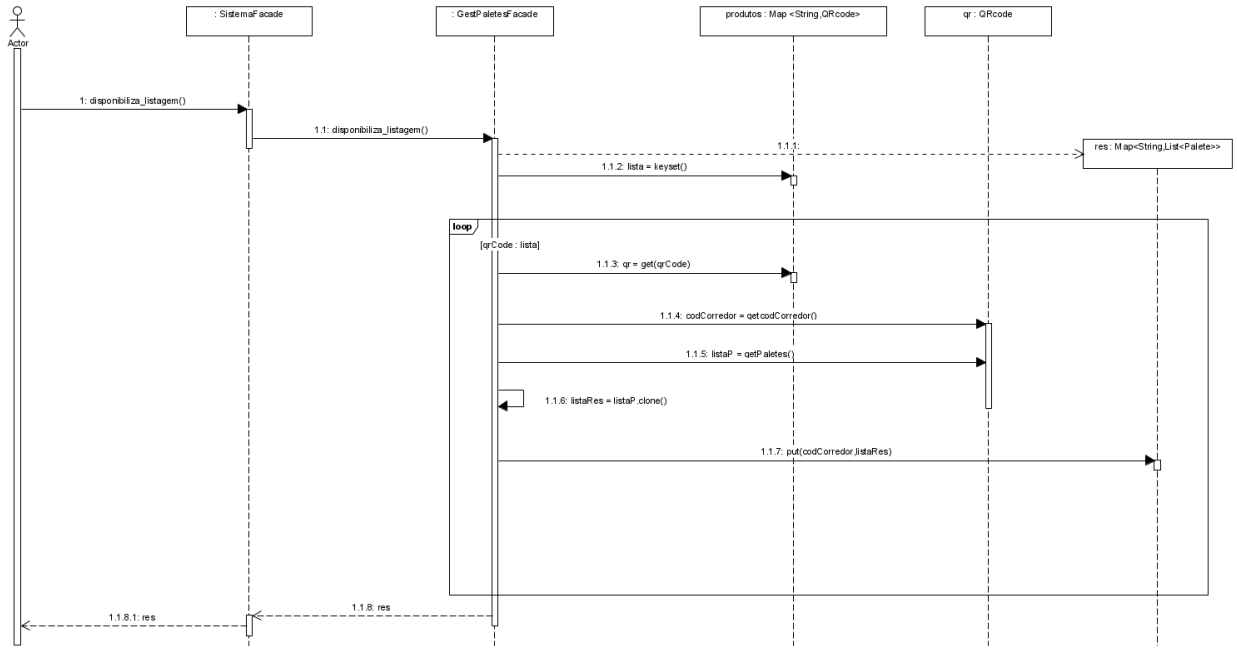


Figura 7: Diagrama de Sequência - DisponibilizaListagem

Este diagrama de sequência corresponde ao método disponibilizaListagem() implementado pelo GestPaletesFacade. Quando o gestor pede uma listagem de todas as localizações das paletes, para avaliar a ocupação do armazém, é utilizado este método que a partir do Map produtos vai organizar todas as paletes registadas pelo corredor onde se encontram. Assim vai ser retornado um Map<String,List<Palete» que associa cada código de corredor a uma lista de paletes.

7.2 RegistaPaletes

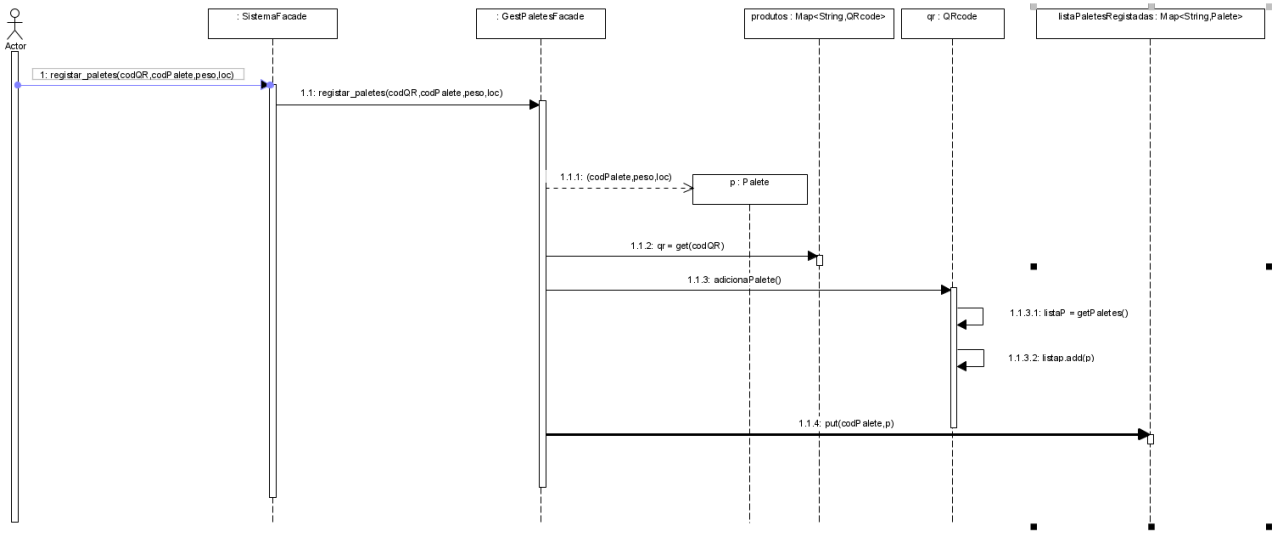


Figura 8: Diagrama de Sequência - RegistaPaletes

Este diagrama de sequência corresponde ao método registarPaletes(codQR,codPalete,peso,loc) implementado pelo GestPaletesFacade. Quando uma palete entra num armazém, após ser lido o seu código QR, terá de ser guardada no sistema. Assim, este método será o responsável por, inicialmente, através do QRcode fornecido adicionar uma palete à classe QRcode correspondente presente no Map de produtos, e, por fim insere-se também no Map listaPaletesRegistadas.

7.3 AtualizaLocalização

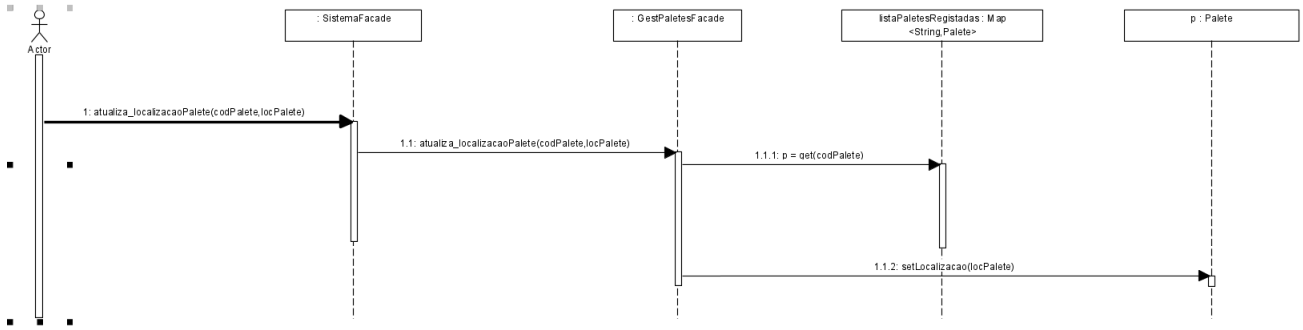


Figura 9: Diagrama de Sequência - AtualizaLocalização

Este diagrama de sequência corresponde ao método atualizaLocalizacaoPalete(codPalete,locPalete) implementado pelo GestPaletesFacade. Quando uma palete altera a sua localização (neste caso será quando for transportada por um robot para um local diferente) é invocado este método modifica o atributo locPalete para a localização dada como argumento.

7.4 ValidaPedidoRequisição

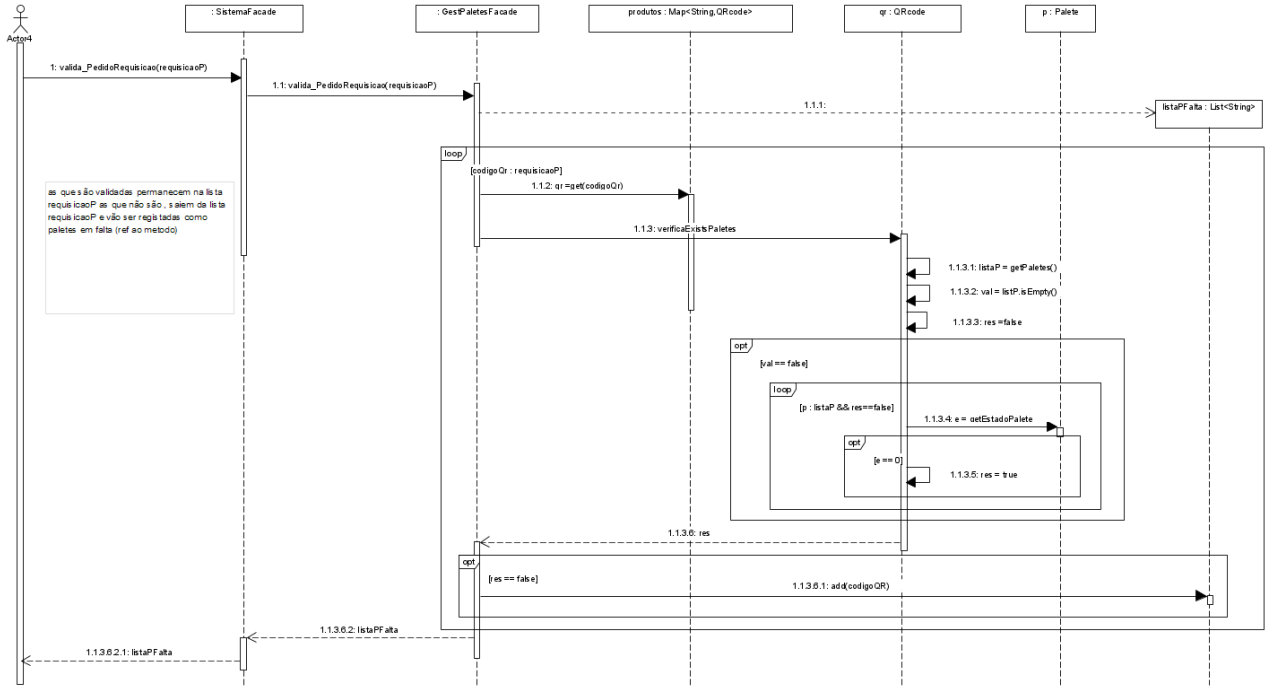


Figura 10: Diagrama de Sequência - ValidaPedidoRequisição

Este diagrama de sequência corresponde ao método validaPedidoRequisicao(requisicaoP) implementado pelo GestPaletesFacade. Para verificar se é possível requisitar os produtos de uma encomenda (lista de códigos QR), é necessário aceder ao Map produtos e verificar se existe pelo menos uma palete de cada produto (dado pela lista do argumento do método). No caso de não estar presente, vai ser criada uma lista de todos os produtos que não poderão ser requisitados.

7.5 RegistaPedidoRequisição

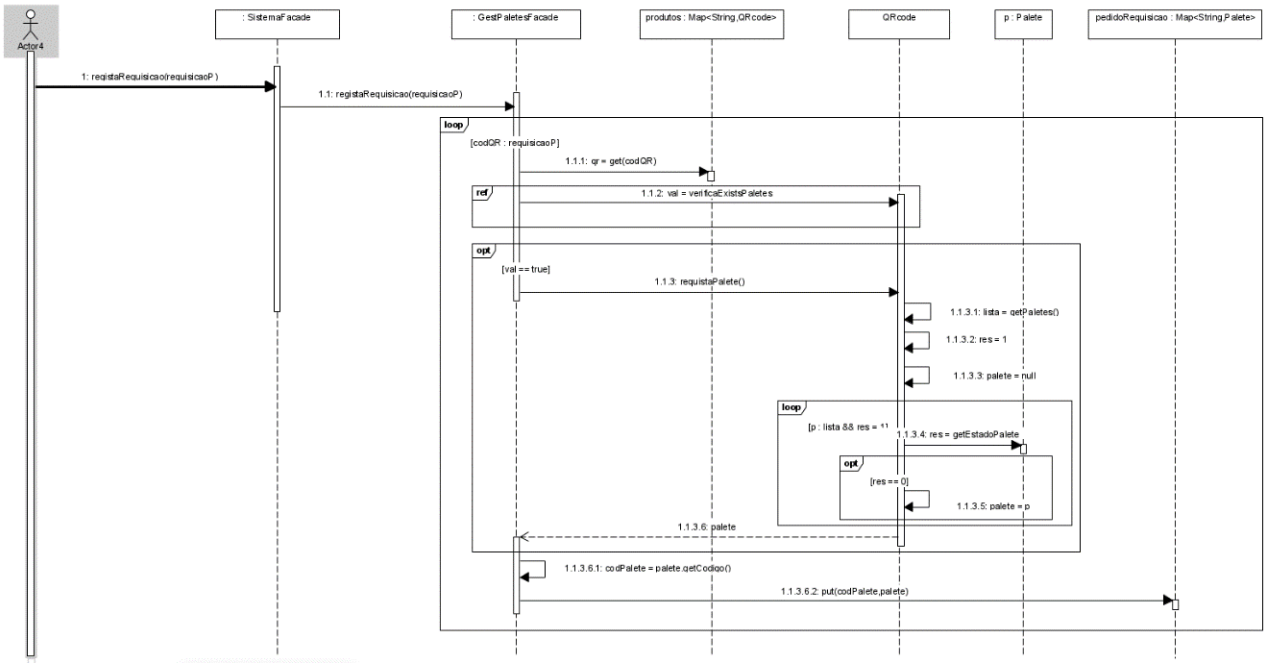


Figura 11: Diagrama de Sequência - RegistaPedidoRequisição

Este diagrama de sequência corresponde ao método registaPedidoRequisicao(requisicaoP) implementado pelo GestPaletesFacade. Este método será o responsável por atribuir uma palete a cada produto que poderá ser requisitado.

7.6 RegistaPaletesFalta

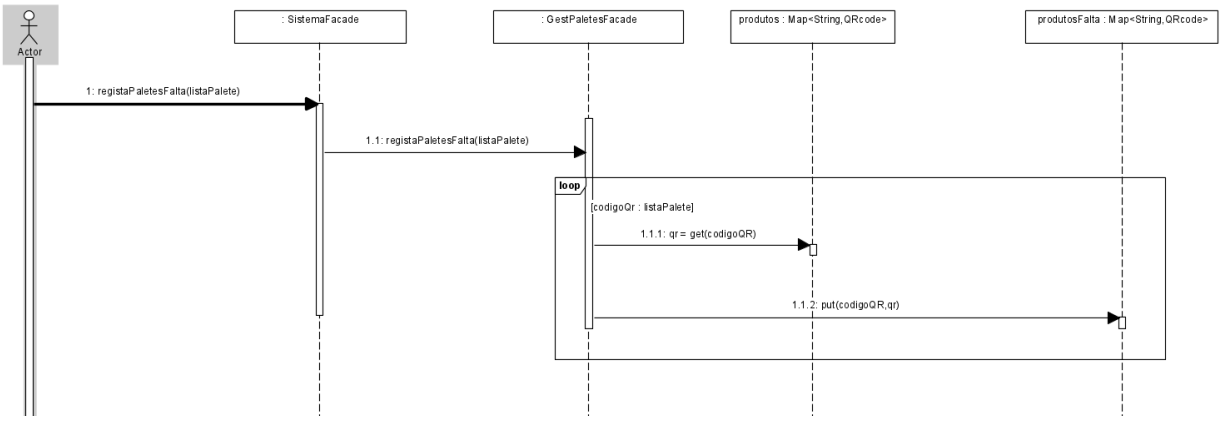


Figura 12: Diagrama de Sequência - RegistaPaletesFalta

Este diagrama de sequência corresponde ao método registaPaletesFalta(listaPalete) implementado pelo GestPaletesFacade. Quando o servidor de produção requisita produtos que não se encontram disponíveis no momento e decide manter o pedido daqueles que estão em falta, invocamos este método cujo argumento é uma lista de QRcodes (identificam produtos). Desta forma, adicionamos todas as matérias primas numa estrutura Map que denominamos paletesFalta.

7.7 AceitaPedidoRequisição

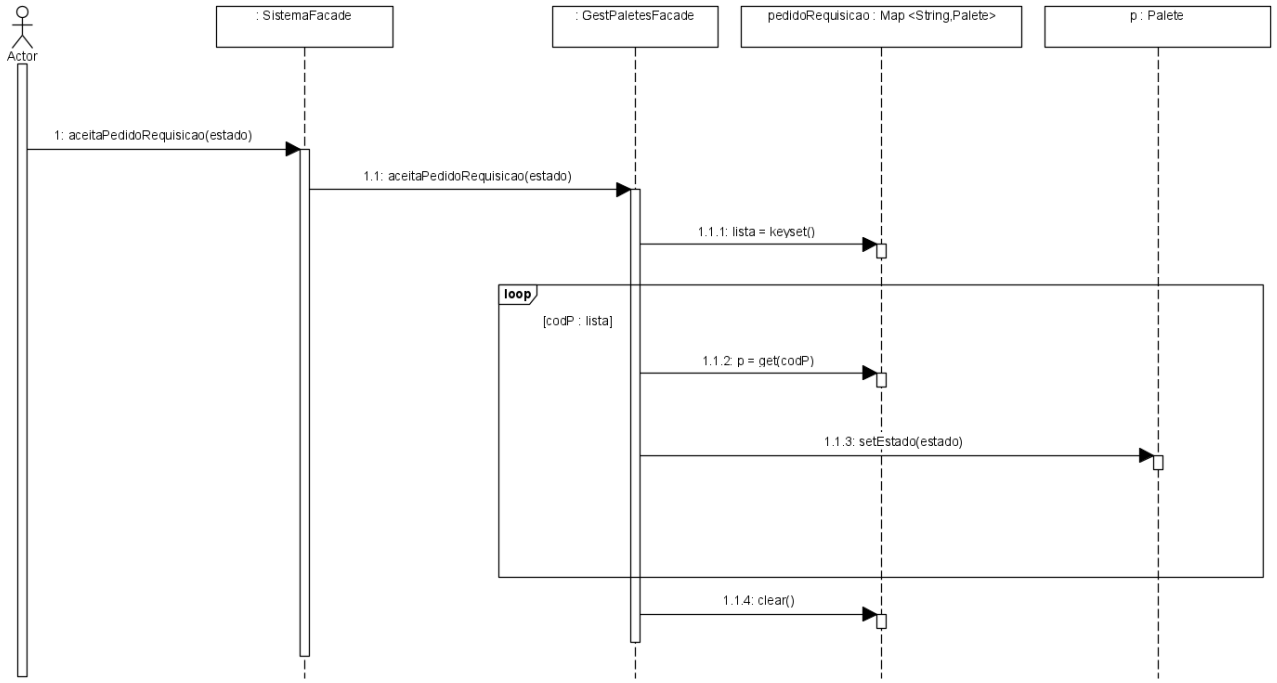


Figura 13: Diagrama de Sequência - AceitaPedidoRequisição

Este diagrama de sequência corresponde ao método aceitaPedidoRequisicao(estado) implementado pelo GestPaletesFacade. Para que seja aceite uma requisição, é invocado este método, que para cada palete presente no Map pedidoRequisicao altera o seu estado para requisitada. No final é feito um clear do Map para retirar todos pedidos de requisição do sistema.

7.8 CancelaPedidoRequisição

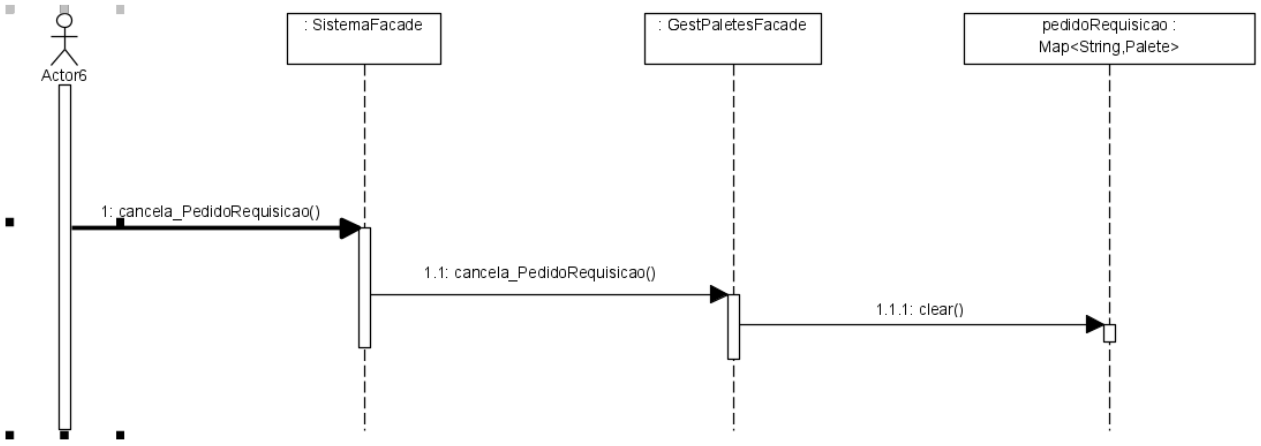


Figura 14: Diagrama de Sequência - CancelaPedidoRequisição

Este diagrama de sequência corresponde ao método cancelaPedidoRequisicao() implementado pelo GestPaletesFacade. No caso do servidor querer cancelar uma requisição, vai ser utilizado este método que vai eliminar todas as entradas do Map pedido Requisição.

7.9 GetRobotDisponivel

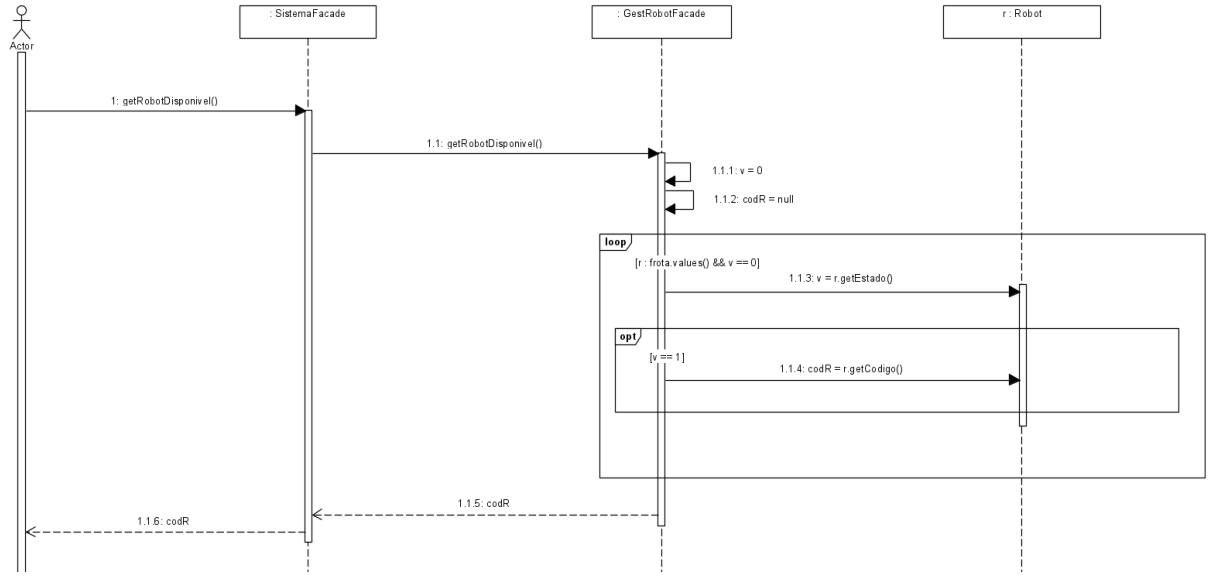


Figura 15: Diagrama de Sequência - getRobotDisponivel

Este diagrama de sequência corresponde ao método getRobotDisponivel() implementado pelo GestRobotFacade. Quando é necessário o transporte de uma palete, o método é invocado e será feita uma pesquisa na estrutura map frota (presente em GestRobotFacade e que possui um código associado a um robot). Isto acontece de forma a tentar encontrar um robot cujo estado seja disponível. Caso isto se verifique devolve-se o código desse robot. Caso contrário é devolvido null.

7.10 NotificaTransporte

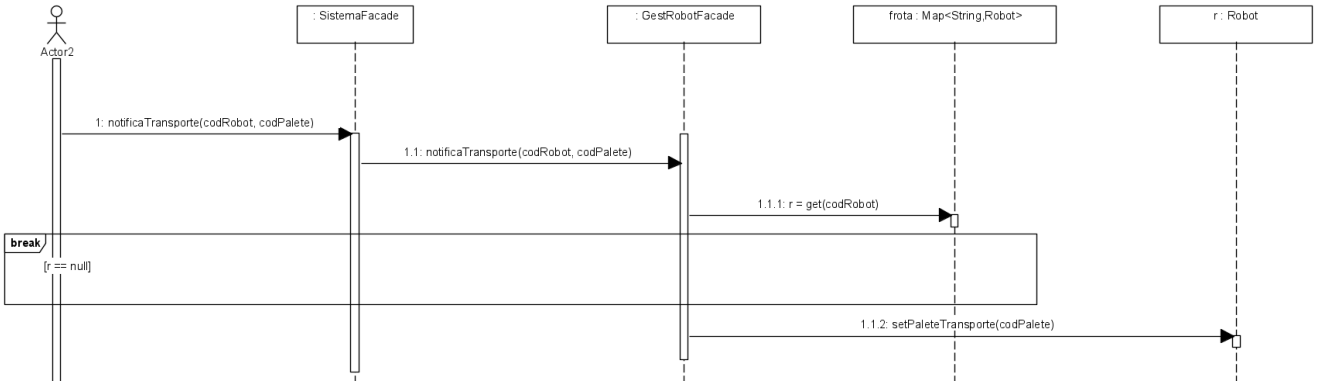


Figura 16: Diagrama de Sequência - NotificaTransporte

Este diagrama de sequência corresponde ao método *notificaTransporte(codRobot, codPalete)* implementado pelo *GestRobotFacade*. A classe robot possui um atributo denominado 'palete' que corresponde ao código de uma palete que o robot terá de transportar. Caso o robot esteja disponível este atributo está null. Desta forma, quando se invoca este método, é necessário procurar pelo robot cujo código é o passado como argumento. Caso ele existe, modifica-se o atributo palete, caso ele não exista é lançada uma exceção.

7.11 AtualizaEstadoRobot

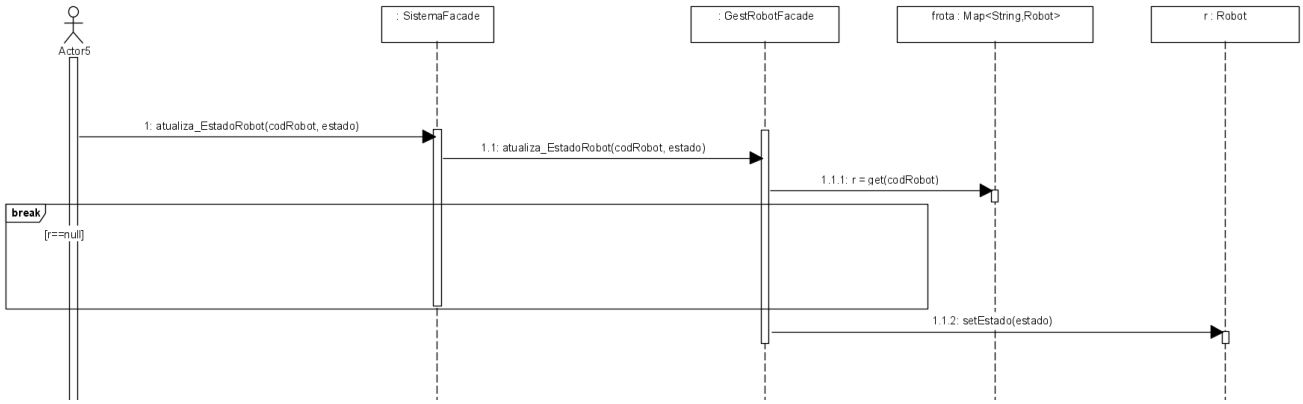


Figura 17: Diagrama de Sequência - AtualizaEstadoRobot

Este diagrama de sequência corresponde ao método *atualizaEstadoRobot(codRobot, estado)* implementado pelo *GestRobotFacade*. Quando o robot inicia ou termina uma tarefa é necessário alterar o seu estado (para disponível ou ocupado). De forma a fazê-lo, é invocado o método referido, passando como argumentos o código do robot em questão e o estado que lhe queremos passar. Com isto, é feita uma pesquisa no Map *frota*, e caso o código do robot já lá esteja obtemos o objeto robot correspondente e é-lhe mudado o estado. Caso o robot não exista é lançada uma exceção.

7.12 IndicaDestino

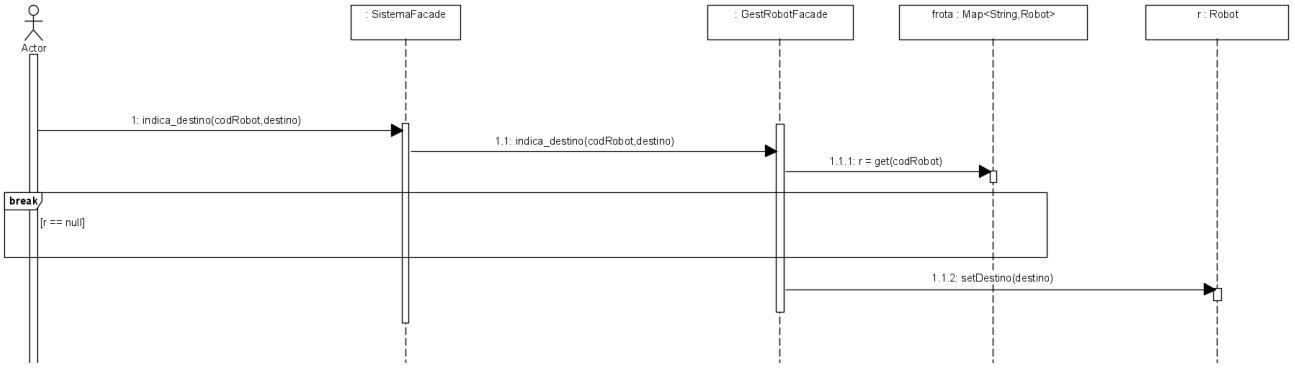


Figura 18: Diagrama de Sequência - IndicaDestino

Este diagrama de sequência corresponde ao método *IndicaDestino(codRobot, destino)* implementado pelo *GestRobotFacade*. A classe robot possui um atributo denominado 'localização' que corresponde ao destino do seu percurso. Caso o robot tenha sido solicitado para uma tarefa que exigiu que ele se movimentasse o atributo adquire a localização final do percurso. Caso o robot esteja parado o atributo não tem nenhuma atribuição. Desta forma, é necessário procurar, na estrutura map denominada frota, o código do robot passado como argumento. Caso ele exista, altera-se o atributo 'localização'. Caso não exista, é lançada uma exceção.

7.13 CalculaRota

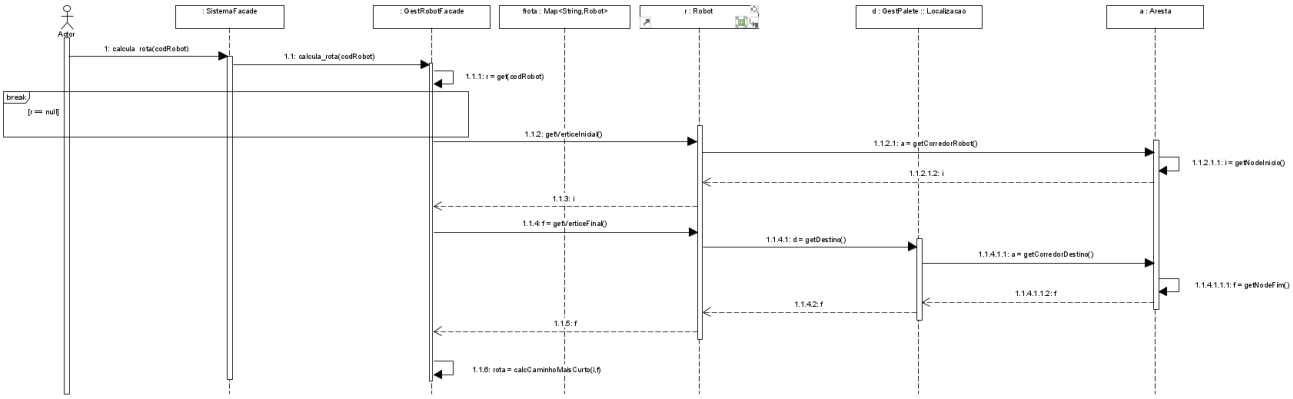


Figura 19: Diagrama de Sequência - CalculaRota

Este diagrama de sequência corresponde ao método calculaRota(codRobot) implementado pelo GestPaletesFacade. Após todas as atribuições necessárias feitas a um robot, é invocado este método que terá acesso às posições inicial e final (vértices do grafo) e utilizando o algoritmo de Dijkstra calculará o caminho mais curto entre elas.

7.14 ValidaRota

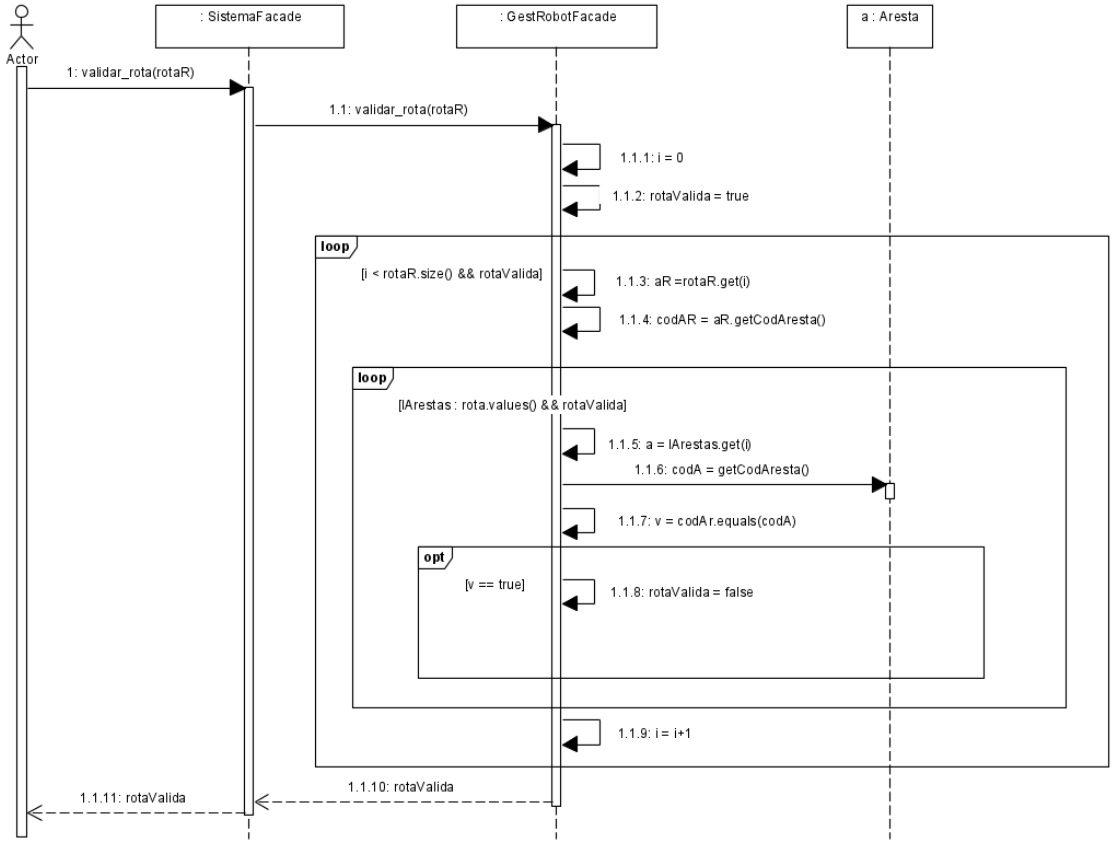


Figura 20: Diagrama de Sequência - ValidaRota

Este diagrama de sequência corresponde ao método validaRota(rotaR) implementado pelo GestRobotFacade. Antes de um robot efetuar um percurso, é necessário avaliar se essa rota é válida, ou seja, se não haverá outros robots no caminho com os quais possa colidir. Desta forma, invoca-se este método cujo argumento é uma lista de arestas (rota a percorrer) e verifica se os robots passam numa mesma aresta ao mesmo tempo. Caso isto aconteça não aconteça, a rota é válida.

7.15 AdicionaRota

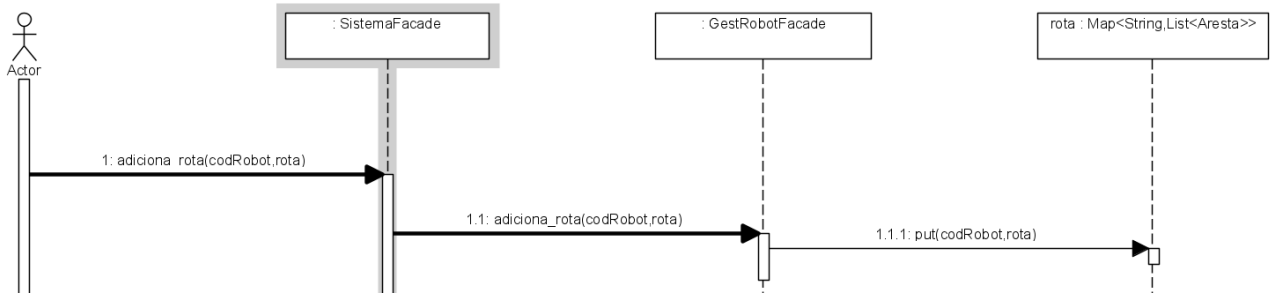


Figura 21: Diagrama de Sequência - AdicionaRota

Este diagrama de sequência corresponde ao método *adicionaRota(codRobot, rota)* implementado pelo *GestRobotFacade*. No facade em questão, existe uma estrutura map, denominadas rotas, na qual temos códigos de robot associados a uma lista de arestas que representam o percurso que eles estão a efetuar. Quando um robot inicia uma tarefa e precisa de se movimentar, é necessário invocar este método, cujos argumentos terão de ser o código do robot em questão e o percurso que lhe foi atribuído. Com isto, é apenas preciso adicionar ao map rotas uma nova entrada.

7.16 CancelaRota

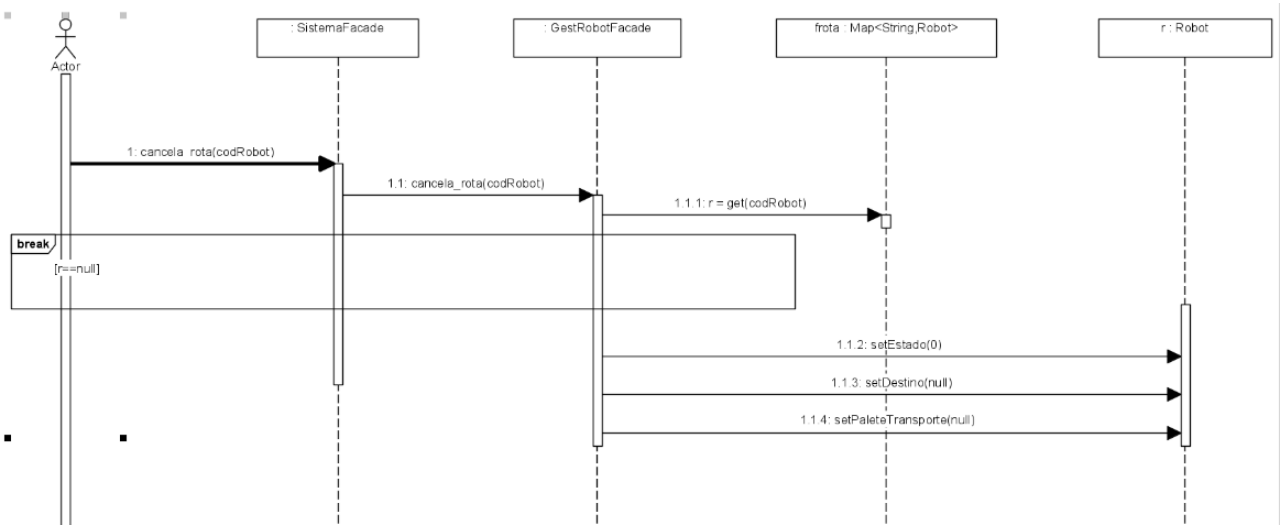


Figura 22: Diagrama de Sequência - CancelaRota

Este diagrama de sequência corresponde ao método *cancelaRota(codRobot)* implementado pelo *GestRobotFacade*. Quando um robot é solicitado para transportar uma palete, antes de se movimentar, é necessário verificar se existe uma rota válida. Caso não haja, todos os atributos que lhe foram atribuídos durante o processo de validação são lhe retirados novamente. Desta forma, quando se invoca este método é necessário , utilizando o código de robot passado como argumento, alterar o seu atributo estado pra disponível, o seu atributo destino para null (pois ele não se irá deslocar) e o seu atributo paleteTransporte também para null (não irá mais transportar uma palete visto que o percurso não foi validado). Caso o robot não exista é lançada uma exceção.

7.17 ValidaAcesso

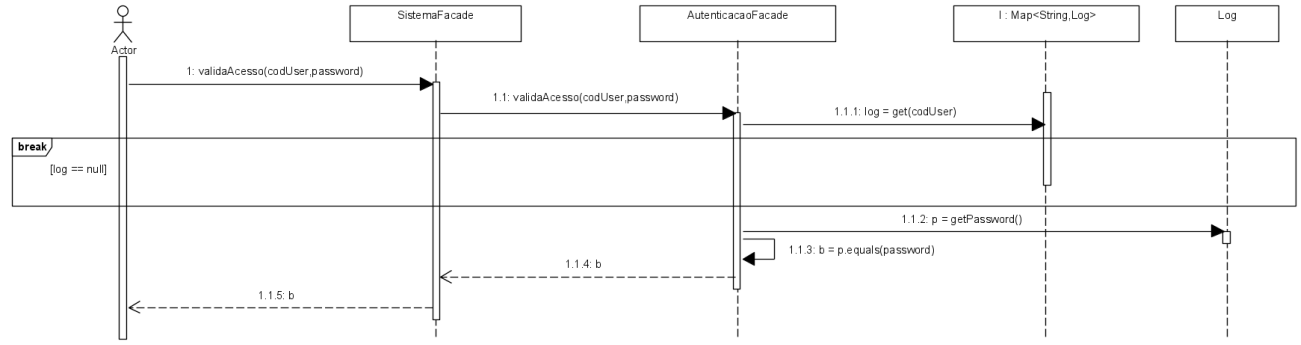


Figura 23: Diagrama de Sequência - ValidaAcesso

Este diagrama de sequência corresponde ao método *validaAcesso()* implementado pelo *AutenticacaoFacade*. Assim que é invocado este método, são verificadas se as credenciais passadas como argumentos correspondem a algum utilizador na estrutura *logs*. Caso isto seja verdade, o utilizador terá acesso á API do sistema, caso contrário a tentativa de acesso é negada.

7.18 Logout

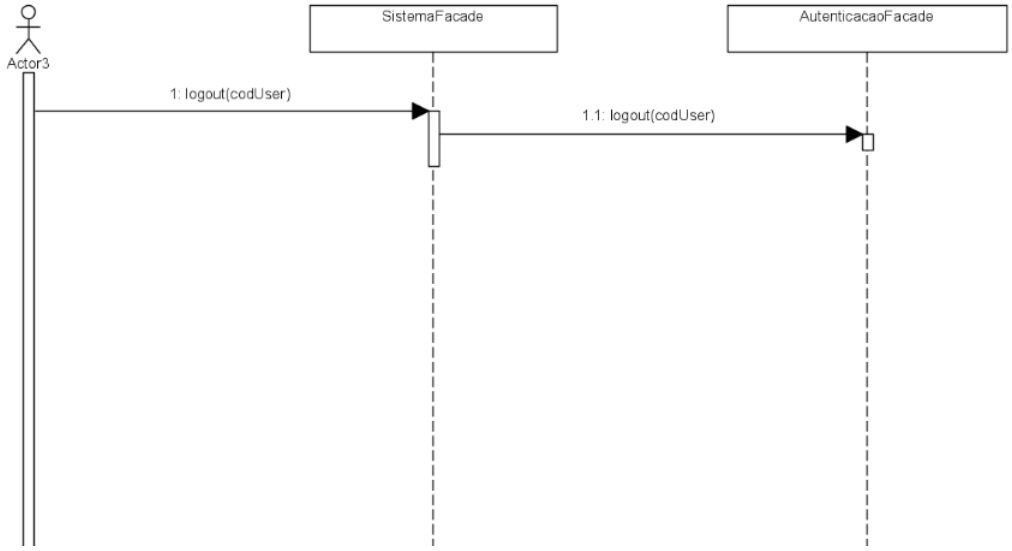


Figura 24: Diagrama de Sequência - Logout

Este diagrama de sequência corresponde ao método *logout(codUser)* implementado pelo *AutenticacaoFacade*. Assim que é invocado este método o utilizador cujo código é passado como argumento deixa de ter acesso á API do sistema.

8 Diagrama de Packages

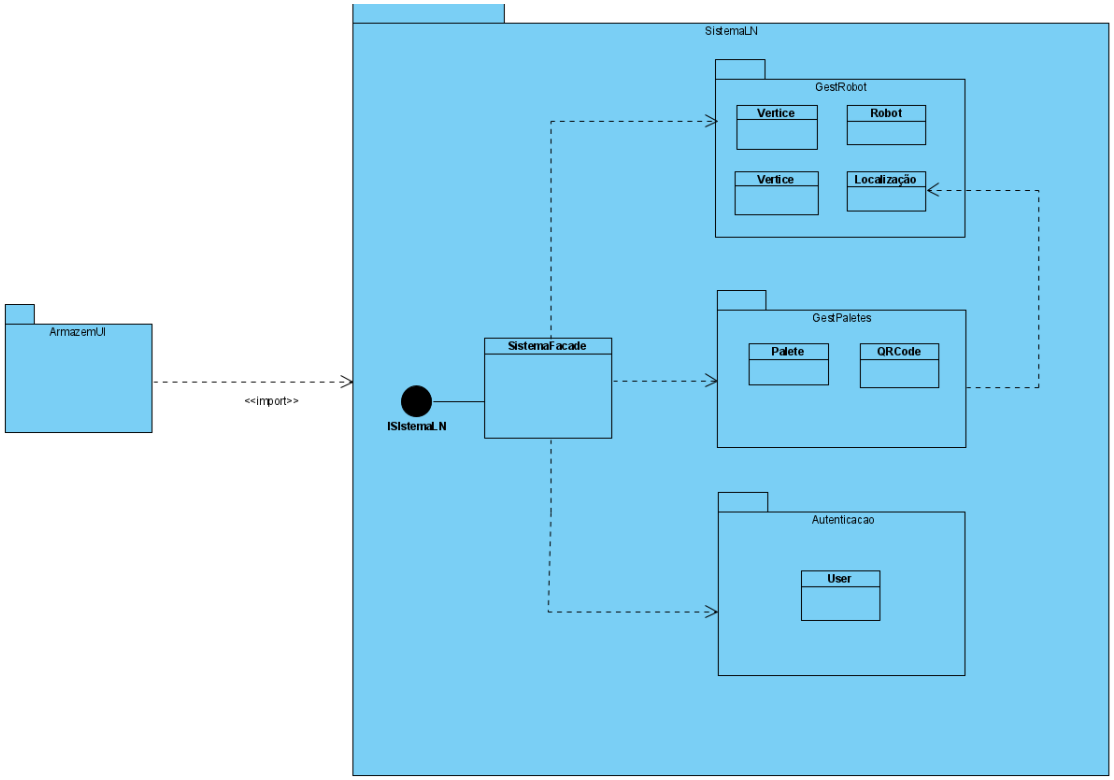


Figura 25: Diagrama de Packages

Este tipo de diagrama tem como objetivo representar as dependências entre os vários packages. Desta forma, o package ArmazemUI irá importar o package SistemaLN que, por sua vez, contém o seguinte: a classe SistemaFacade, a interface que este implementa ISistemaLN e os packages correspondentes a cada subsistema. Cada package de subsistema contém as classes desenvolvidas (e que lhes foram atribuídas) durante todo o processo. Visto que o GestPaletes necessita da classe Localização presente em GestRobot, foi preciso representar o import dessa classe por parte do package referido.

9 Reflexão Final

9.1 Análise Crítica

Durante a elaboração desta segunda fase do projeto deparamo-nos com algumas adversidades. Inicialmente, notámos algumas falhas no nosso Modelo de Domínio, que tiveram de ser corrigidas além de termos um Use Case do gestor em falta. À parte disto, tudo o que foi mudado na fase 1 serviu apenas aumentar o grau de detalhe do trabalho e adicionar funcionalidades que achamos que nos seriam úteis.

Durante a elaboração do Diagrama de Componentes, o grupo discordou quanto ao número de Subclasses, tendo escolhido, numa primeira tentativa, 4 diagramas de classe, mas no final pensámos que seria mais correto, ter apenas 3. As grandes dificuldades com que o grupo se deparou, residiram na elaboração do Diagrama de Classes e dos Diagramas de Sequência. Quanto aos Use cases, visto que tínhamos uma classe (Localização) que era necessária em dois subsistemas, a equipa discordou á sua localização base.

Posteriormente, tivemos de fazer uma reavaliação da nossa abordagem do projeto, pois, foi pedido pela equipa docente, para considerar o armazém como um "grafo lógico", tendo em conta que as arestas seriam corredores com prateleiras e os vértices, zonas de decisão do robot (local onde ele escolhe que novo corredor vai percorrer). Isto veio alterar a nossa metodologia inicial que consistia em trabalhar com coordenadas. Esta nova interpretação revelou-se mais adequada ao projeto em questão. O grupo também pensou em criar uma classe grafo, no subsistema Gestrobot, no entanto como ideia surgiu um no dia da entrega e isso implicaria mudanças de última hora, preferimos não aplicar isso nesta fase. Além de tudo isto, surgiram algumas dificuldades no diagrama de classe, nomeadamente, em relação á escolha das estruturas de dados (lists e maps) que achámos relevantes para estarem no facade e que métodos deveriam estar em classes concretas e não no facade do Subsistema.

Por último, o grupo reparou, ao desenvolver os Diagramas de Sequência, que alguns métodos sobrecarregavam um pouco o facade do subsistema correspondente. Isto foi um motivo de alguma discussão entre o grupo, pois foi referido, várias vezes, pela equipa docente, que seria uma má prática de programação. No entanto, chegámos á conclusão de que, com a metodologia por nós implementada, isto seria incontornável em alguns métodos. Desta forma, o que acabou de ser referido foram os grandes desafios que o grupo enfrentou ao longo desta fase, tendo havido outros de menor importância e que foram rapidamente resolvidos.

9.2 Conclusão

A segunda fase deste projeto da cadeira de DSS, consistiu essencialmente na aplicação das metodologias lecionadas nas aulas teóricas. Isto permitiu, não só consolidar todo o conhecimento adquirido, bem como permitiu ao grupo ter uma nova perspetiva de programação, não começando logo a criar código puro, mas sim a planear o trabalho faseadamente, permitindo-nos ter uma visão mais geral e concreta. A primeira fase consistiu na análise de requisitos e o levantamento das funcionalidades dos vários atores, enquanto que a segunda foi uma continuação da anterior, e na qual foram criados os vários diagramas (classe, packages, sequência, componente) que representam o funcionamento geral do nosso projeto. Foi necessária, uma atenção redobrada aos pormenores e aos detalhes da nossa metodologia, uma vez que é a partir deste ponto que vamos passar para a implementação em código Java do sistema desenvolvido, sendo isto facilitado por uma funcionalidade do Visual Paradigm.

É importante referir também, que ao longo desta fase os elementos do grupo iam tendo diferentes pontos de vista (todos igualmente válidos) em relação às adversidades que iam surgindo. Isto permitiu-nos analisar o problema de vários "ângulos" de maneira a escolher a melhor resolução entre todas as apresentadas.