Universidade de Aveiro

Departamento de Eletrónica,
Telecomunicações e Informática

Sistemas Distribuídos
Relatório Trabalho Prático nº 2

Rapsódia no Aeroporto
Turma 1 - Grupo 2

Autores:

60456 - Tiago Soares

60089 - David Simões

Supervisor:

Prof. Óscar Pereira

29 de Abril de 2014

Conteúdo

Introdução

O objectivo deste trabalho prático consiste em construir uma simulação de atividades ligadas ao desembarque de passageiros no aeroporto, algures nos arredores da cidade de Aveiro, baseada no modelo cliente-servidor, com replicação de servidor, em que o bagageiro, os passageiros e o motorista são os clientes e as regiões de interacção que tenha estabelecido representam os serviços que lhes são prestados pelos servidores.

A nossa solução final deve ser implementada em Java, ser passível de execução em Linux sobre sockets TCP, de uma forma distribuída (até dez máquinas diferentes), e terminar (deve contemplar a possibilidade de *shutdown* do serviço).

Foram tomadas em conta as seguintes condições:

1. Vão realizar-se K chegadas de aviões, envolvendo cada uma delas o desembarque de N passageiros;

2. Cada passageiro transporta no porão do avião de 0 a M peças de bagagem;

3. O autocarro de transferência entre terminais tem uma lotação de T lugares;

Assumimos por defeito que há cinco chegadas de aviões, cada uma envolvendo o desembarque de seis passageiros, transportando um máximo de duas peças de bagagem no porão do avião, e que a lotação do autocarro de transferência entre terminais é de três passageiros.

Capítulo 1

Arquitectura Geral da Solução

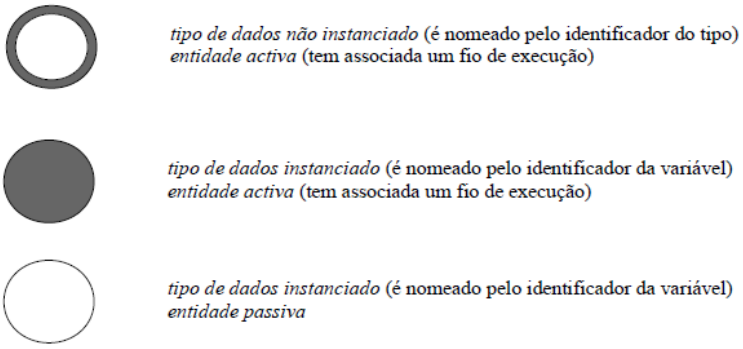

Figura 1.1: Elementos Base do Diagrama de Decomposição

A figura acima representada permite legendar e/ou definir quais são os tipos de dados (elementos base) que existem no nosso diagrama de arquitectura da decomposição da solução (ver imagem seguinte).

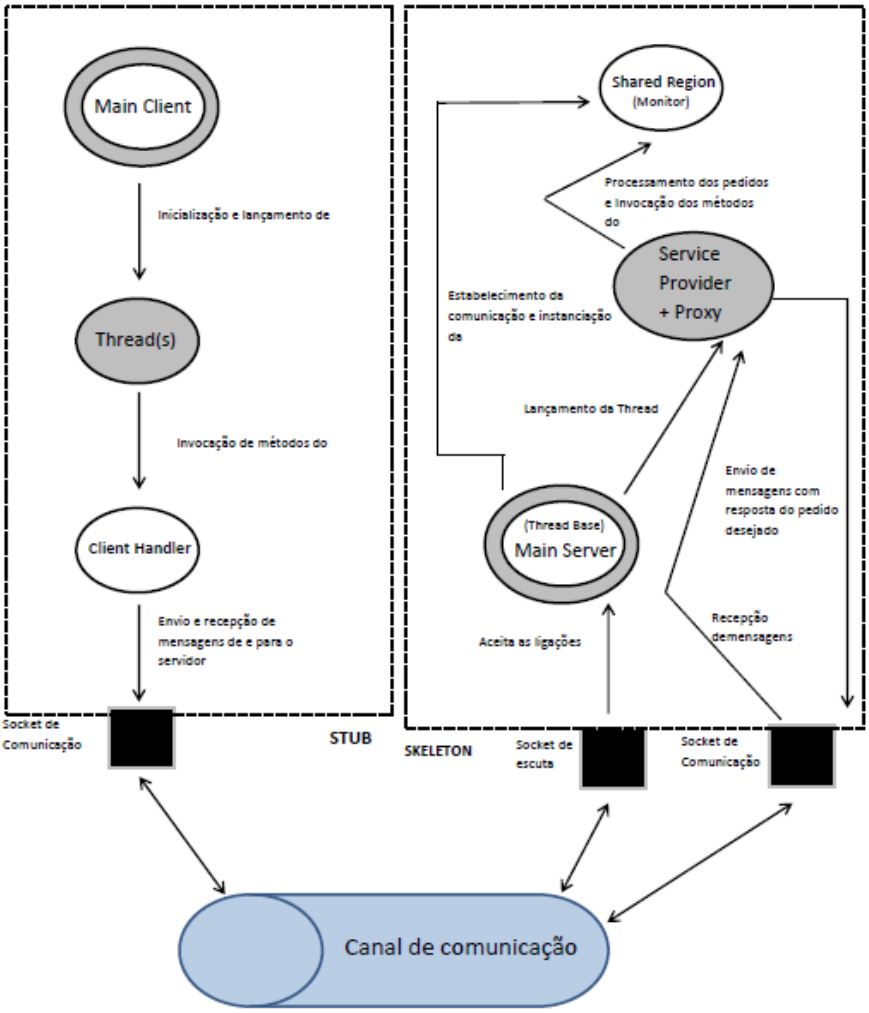

Figura 1.2: Diagrama da Arquitectura Geral da nossa Solução

Com base no diagrama de arquitectura geral da nossa aplicação, é possível verificar que existem duas componentes distintas mas que estão interligadas entre si: a componente do cliente e a do servidor.

Neste modelo de sistema cliente-servidor, a Main Client faz a inicialização e o lançamento da(s) nossa(s) thread(s) já implementada(s) no projecto anterior (Porter, Driver e Passenger), depois cada uma desta(s) invoca os métodos do Client Handler que por sua vez processa a mensagem, chama o método apropriado do monitor e envia a mensagem para o servidor através da socket de comunicação.

Do lado do servidor é necessário fazer a instanciação do objecto e a Main Server (Thread Base) estabelece a comunicação e aceita a ligação através da socket de escuta por ligação ao canal de comunicação e faz o lançamento da Thread para o elemento Service Provider+Proxy e este interpreta a mensagem e faz o processamento do pedido da mesma sendo ainda responsável por invocar o método no Monitor que está contido na mensagem.

Finalmente a Thread Base estabelece a instanciação da Shared Region (Monitor associado) e o Service Provider+Proxy envia a mensagem com a resposta do pedido desejado de novo para o cliente mas através de uma socket diferente (socket de comunicação), ou seja, é-lhe retornada uma mensagem de OK com o resultado do método anexado (caso exista). Essa mensagem é recebida pelo Client Handler (lado do cliente) através da socket de comunicação.

Capítulo 2

Entidades

2.1 Threads e Bagagens

De acordo com o projecto prático nº1, definimos que os passageiros, bagageiro e motorista seriam as nossas threads, ou seja, classes que fazem extensão à classe *Thread*, dado que estes são os intervenientes do nosso problema (são os que realizam as diversas actividades). Assim, podemos enunciar as nossas threads da seguinte forma:

1. O **plane**, que simula a ida e volta de múltiplos aviões no aeroporto cada um com alguns passageiros e algumas bagagens. É responsável por inicializar a thread do Passenger bem como a redefinição das zonas entre os aviões;

2. Os **passageiros**, que contêm malas e podem ter terminado a sua viagem ou continuar em trânsito;

3. O **bagageiro**, que descarrega as bagagens do avião (quando este aterra) e as transporta para as zonas de recolha de bagagens ou de armazenamento temporário;

4. O **motorista** do autocarro, que transporta os passageiros em trânsito entre as zonas de transferência de terminal;

A class que representa as bagagens (*Luggage class*) e que já foi criada no projecto anterior foi mantida neste e contêm duas variáveis: **ownerId** que corresponde à identificação do dono da bagagem e **ownerTravelling** que é um tipo de dados *boolean* e que indica se o dono está em viagem ou não.

2.2 Zonas e Monitores

Quanto aos monitores, decidimos que estes seriam as zonas do aeroport com as quais as threads interagem e onde é necessário haver informação partilhada. Conseguimos isolar 8 zonas: Arrival Zone, Luggage Claim, Storage Zone, Guichet, Arrival Transfer Zone, Departure Transfer Zone, Exit Zone e Entry Zone. No entanto, nem todas estas zonas têm de ser monitores. Conseguimos unir zonas com caracteristicas semelhantes, como as zonas de Entry e de Exit, ou as zonas relacionadas com bagagens (Luggage Claim, Storage e Guichet).

Deste modo, os monitores são os seguintes: *ZoneArrival* - zona de desembarque; *ZoneEntryExit* - entrada no terminal de embarque e saída do terminal de desembarque; *ZoneLuggage* - zona de recolha de bagagens, zona de armazenamento temporário de bagagens e guichet de reclamação de bagagens perdidas; *ZoneTransferArrival* - zona de transferência no terminal de desembarque e *ZoneTransferDeparture* - zona de transferência no terminal de embarque.

Criámos ainda um monitor designado por *General* que permite incorporar um ficheiro de *logging* que descreve de um modo conciso e claro a evolução do estado interno das diferentes entidades envolvidas. Desta forma, no final de todas as operações, é emitido um relatório completo das actividades desenvolvidas num ficheiro de saída (definido por argumento).

2.3 Interações Thread/Monitor

Figura 2.1: Diagrama de interação entre entidades passivas e activas

Legenda:

1. Ciclo de Vida do Passageiro
    - whatShouldIDo()
    - goCollectABag()
    - reportMissingBags(int bags)
    - goHome()
    - takeABus()
    - enterTheBus()
    - leaveTheBus()
    - prepareNextLeg()

2. Ciclo de Vida do Porter
    - takeARest()
    - noMoreBagsToCollect()
    - tryToCollectABag()
    - carryItToAppropriateStore(Luggage t)

3. Ciclo de Vida do Driver
    - goToDepartureTerminal()
    - parkTheBusAndLetPassOf()
    - goToArrivalTerminal()
    - parkTheBus()
    - hasDaysWorkEnded()
    - announcingBusBoarding()

4. Main Instancia o Proxy
    - GeneralProxy client = new GeneralProxy (serverSock.accept(), monitor)

- client.start()

5. Main Instancia o Proxy
    - ZoneArrivalProxy client = new ZoneArrivalProxy (serverSock.accept(), monitor)
    - client.start()

6. Main Instancia o Proxy
    - ZoneEntryExitProxy client = new ZoneEntryExitProxy (serverSock.accept(), monitor)
    - client.start()

7. Main Instancia o Proxy
    - ZoneLuggageProxy client = new ZoneLuggageProxy (serverSock.accept(), monitor)
    - client.start()

8. Main Instancia o Proxy
    - ZoneTAProxy client = new ZoneTAProxy (serverSock.accept(), monitor)
    - client.start()

9. Main Instancia o Proxy
    - ZoneTDProxy client = new ZoneTDProxy (serverSock.accept(), monitor)
    - client.start()

10. Mensagens Recebidas e Método que Invocam
    - RL = reportLog()
    - ReL = resetLog()

- SDS = setDriverState()
- SPoS = setPorterState()
- SPaS = setPassengerState()
- AWQ = addWaitQueue()
- ABQ = addBusQueue()
- LWQ = leaveWaitQueue()
- LBQ = leaveBusQueue()
- SLB = setLCB()
- SSR = setST()
- SMA = setMalas()
- SPT = setPassengerTravel()
- SPMB = setPassengerMaxBags()
- SPCB = setPassengerCurrBags()
- RBM = reportBagsMissing()

11. Mensagens Recebidas e Método que Invocam

    - TAR = takeArest()
    - IGH = iGotHere()
    - GLN = getLugNumber()
    - GL = getLug()
    - AL = addLuggage()

12. Mensagens Recebidas e Método que Invocam

    - PL = peopleLeft()
    - GH = goingHome()
    - RP = resetPlane()

13. Mensagens Recebidas e Método que Invocam

    - PABLBC = putABagLBC()
    - NMB = noMoreBags()

- PABSR = putABagSR()
- RLM = reportLuggageMissing()
- GL = getLuggage()
- RP = resetPlane()

14. Mensagens Recebidas e Método que Invocam

    - DEP = depart()
    - ABB = announceBusBoarding()
    - TAB = takeABus()

15. Mensagens Recebidas e Método que Invocam

    - DEP = depart()
    - LB = leaveBus()
    - FBT = finishBusTrip()
    - PRK = park()

16. Métodos Disponíveis para a Plane Thread

    - setMalas(int fn, int ln)
    - resetLog()
    - setPassengerTravel(int i, boolean over)
    - setPassengerMaxBags(int i, int bags)
    - addLuggage(Luggage t)
    - resetPlane()

17. Métodos Disponíveis para a Passenger Thread

    - setPassengerState(int id, int st)
    - iGotHere()
    - goingHome(int id)
    - reportLuggageMissing(int bags)
    - getLuggage(int id, int lugTotal)

- takeABus(int id)
- finishBusTrip()
- leaveBus(int id)

18. Métodos Disponíveis para a Porter Thread

    - setPorterState(int st)
    - takeArest()
    - getLugNumber()
    - getLug()
    - putABagLBC(int ownerId)
    - noMoreBags()
    - putABagSR()

19. Métodos Disponíveis para a Driver Thread

    - setDriverState(int st)
    - peopleLeft()
    - announceBusBoarding()
    - park(int potb)
    - depart()

De acordo com o esquema anterior é possível verificar quais é que são as entidades activas e passivas na nossa implementação e ainda quais as relações entre elas (threads e respectivos monitores). Além destas relações, todas as entidades (monitores e threads) com exceção da ZoneArrival interagem com o monitor General, para tratar do logging correto da simulação.

Cada Client Main (Driver, Plane e Porter) espera pela configuração da mensagem e inicializa as respectivas Threads através da operação *start()* e cada thread executa o seu ciclo de vida (ver mais detalhes na seção "Ciclos de Vida"). Cada uma das threads é responsável por invocar os métodos do Client Handler e este envia a mensagem apropriada para o monitor respectivo com um dado IP e aguarda pela resposta.

Cada Server Main (GeneralMain, ZoneTAMain, etc...) espera pela configuração da mensagem e inicializa o Monitor. Depois disto, irá receber ligações e cria threads para lidarem com os pedidos.

Cada uma das Proxy processa a mensagem e chama o método apropriado no monitor e retorna uma mensagem de OK com o resultado do método (caso exista) como resposta ao cliente.
Foram criadas interfaces para cada Monitor consoante quem interagia com este para garantir que as threads não pudessem alterar informação que não estivesse relacionada com as próprias. As diferentes Zonas também utilizam interfaces para interagir com o monitor General.

De referir que foram estabelecidos diferentes **packages** para distinguir threads, interfaces e monitores tornando o projecto deste modo melhor estruturado.

Capítulo 3

Modificações ao Projecto 1

Após termos falado com o professor para avaliarmos o nosso trabalho prático nº1, tivémos em conta algumas alterações que nos foram pedidas para este projecto: minimizar os *notifyAll()*. Entre outras soluções possíveis, optámos por escolher duas formas distintas de reduzir o uso de notifyAll(), ou seja, acordar threads em específico tanto quanto possível. São elas: usando *ReentrantLock* com *Conditions* e ainda o wait() em objetos separados.

No que diz respeito à primeira solução, o primeiro aspecto a ter em conta foi saber em que zonas faria sentido estarmos a reduzir os notifyAll() e rapidamente percebemos que a zona ZoneTA era a principal zona pela qual adoptámos acordar todas as threads várias vezes. Para tal foi necessário criar um *lock* e a partir deste criar as Conditions necessárias, no nosso caso foram necessárias 1+N: uma para o Driver e uma Queue de Conditions para os Passengers. Todos os notifyAll() são substituídos pela operação signal() da Condition e o wait() é substituído por await(), mas tudo isto deve-se encontrar dentro de um try-finally onde são realizadas sempre as operações de lock() e unlock().

Outra solução seria fazer com que as diferentes Threads fizessem wait() em diferentes objetos, o que nos permitiria acordar cada Thread especificamente. Assim, no monitor ZoneLuggage, criámos um array de objetos onde os passageiros fazem wait() e alterámos os métodos para acordarem os passageiros específicos. Depois, tivémos de separar partes da zona crítica, de maneira a não termos wait()s em blocos synchronized encadeados, que levaria ao problema de "Nested Monitor Lockout".

Capítulo 4

Notas Suplementares

4.1 Classe Config

Como a solução agora é distribuída, todas as Threads e Monitores, antes de estarem prontos para cumprir as suas funções, esperam por uma mensagem de configuração, envida pela classe Config. Essa classe toma como argumentos os valores de K (número de aviões), N (número de passageiros por avião), M (número máximo de malas que um passageiro pode ter), T (lugares disponiveis no autocarro), LogName (o nome do ficheiro de log) e ConfigName (o nome do ficheiro de configuração de endereços). Estas variáveis têm valores por defeito (5, 6, 2, 3, log.txt, config.txt, respetivamente) mas, caso os argumentos estejam definidos, vão ser alteradas e criar simulações porventura muito diferentes.

A Config lê o ficheiro de configuração para saber em que endereço e porta estão cada Thread e cada Monitor e configura-os, de maneira a que a simulação possa dar início. Para fazer repetidas simulações, os programas com as Threads têm de ser re-iniciados, ao contrário dos Monitores, que nunca param. Nessa situação, a mensagem de configuração dá inicio às Threads e reconfigura os monitores para os novos parâmetros. Para terminar os monitores, a Config também pode receber como argumento a opção "ter"(de Terminate), que vai enviar aos monitores uma mensagem "EXIT"e os faz terminar.

4.2 General Monitor

Outra mudança foi no monitor de Logging. Dantes, a Main() sabia quando todas as threads acabavam e dizia ao monitor que a simulação terminara. Agora, o monitor em si analisa o estado das threads e sabe quando todas terminaram, percebendo então que o ficheiro de log deve ser guardado.

4.3 Scripts de Deployment

Para o deploy da solução num sistema distribuído ser rápido e eficiente, criámos scripts para copiar o .JAR para todas as máquinas fornecidas. O script define na primeira linha a password das máquinas e executa algo do género para todas as máquinas:

```
/usr/bin/expect -c 'spawn scp SD_T1_P2_G2.jar
    sd0102@l040101-ws1.clients.ua.pt:.;
    expect "sd0102" { send "'`echo $PW`'\r" };
    interact'
```

Também copia o ficheiro de configuração para a máquina onde corre a classe Config.

Para copiar com facilidade os ficheiros de log, foi criado um script que copia o ficheiro de log da máquina onde corre o Monitor de Logging para a máquina local.

```
/usr/bin/expect -c 'spawn scp
    sd0102@l040101-ws5.clients.ua.pt:./log.txt log.txt;
    expect "sd0102" { send "'`echo $PW`'\r" }; interact'
```

4.4 Scripts de Execução

Para correr cada .JAR com a classe correta na máquina correta, foram criados scripts do seguinte género para todas as diferentes Threads e Monitores, assim como para a máquina onde corre a classe Config:

```
/usr/bin/expect -c 'spawn ssh
    sd0102@l040101-ws3.clients.ua.pt;
    expect "sd0102" { send "'`echo $PW`'\r" };
    expect "sd0102" { send "java -cp SD_T1_P2_G2.jar
        Proj2.Plane.PlaneMain 22215\r" };
    interact'
```

4.5 Ficheiro de Configuração

O ficheiro de configuração foi definido de forma simples. Cada linha ia equivaler ao endereço e porta de uma das máquinas. No nosso caso, colocámos as portas dos monitores como 22217 e as portas de Threads como 22215 e acabámos por ter um ficheiro de configuração com o sgeuinte texto:

```
l040101-ws5.clients.ua.pt 22217 General
l040101-ws9.clients.ua.pt 22217 Arrival
l040101-ws7.clients.ua.pt 22217 EntryExit
l040101-ws8.clients.ua.pt 22217 Luggage
l040101-ws10.clients.ua.pt 22217 TA
l040101-ws11.clients.ua.pt 22217 TD
l040101-ws2.clients.ua.pt 22215 Porter
l040101-ws7.clients.ua.pt 22215 Driver
l040101-ws3.clients.ua.pt 22215 Plane
```

Este ficheiro é copiado para a máquina onde corre a classe Config e define como aceder a todas as Threads e Monitores da nossa solução num ambiente distribuído.

Capítulo 5

Ciclos de Vida

A partir dos diagramas abaixo, podemos retirar que as transições entre estados correspondem a funções que as threads vão executar concorrentemente. Quando as threads são started(), começam no seu estado inicial e percorrem o diagrama até acabarem a sua execução.

Por questões de teste, entre cada mudança de estado, todas as threads fazem um sleep de 0 a 100 milisegundos. Isto foi feito para forçar o sistema e garantir que não havia cenários de deadlock.

5.1 Porter

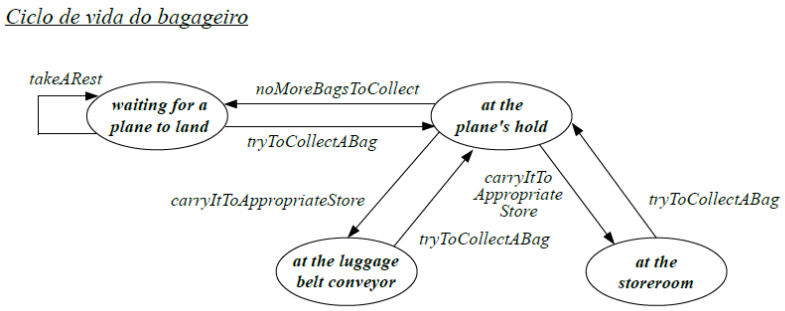

Figura 5.1: Esquema relativo ao Ciclo de vida do Bagageiro

A thread *Porter* executa as seguintes operações:

1. takeARest() - o bagageiro descansa até ao desembarque de todos os passageiros (invoca o método takeARest() que se encontra no monitor ZoneArrival);

2. noMoreBagsToCollect() - quando já não há mais malas para recolher, o bagageiro vai descansar;

3. tryToCollectABag() - quando o bagageiro recolhe as bagagens do porão do avião, uma a uma, e distribui-as pelas zonas de recolha de bagagens e de armazenamento temporário, conforme se trate de bagagens pertencentes a passageiros que terminam a sua viagem no aeroporto ou que estão em trânsito (invoca os métodos getLugNumber() e getLug() que se encontram no monitor ZoneArrival e noMoreBags() que se encontra no monitor ZoneLuggage);

4. carryItToAppropriateStore() - quando o bagageiro distribui as bagagens pertencentes a passageiros que ainda estão em trânsito ou que já terminaram a sua viagem (invoca os métodos putABagSR() ou putABagLBC(), respectivamente, que se encontram no monitor ZoneLuggage);

5.2 Plane

A PlaneThread está encarregue de, dentro de um ciclo com K iterações, simular as chegadas dos aviões, criando N passageiros por iteração, com 0 a M malas por passageiro. Uma mala tem uma chance de 20% de ser perdida e não ser colocada no Plane's Hold (caso o passageiro não esteja em trânsito; se estiver, a mala não é perdida). Os passageiros têm uma chance de 50% de estar em trânsito.

A cada ciclo, a PlaneThread espera que todos os passageiros tenham terminado e informa os monitores que vai chegar outro avião.

5.3 Passenger

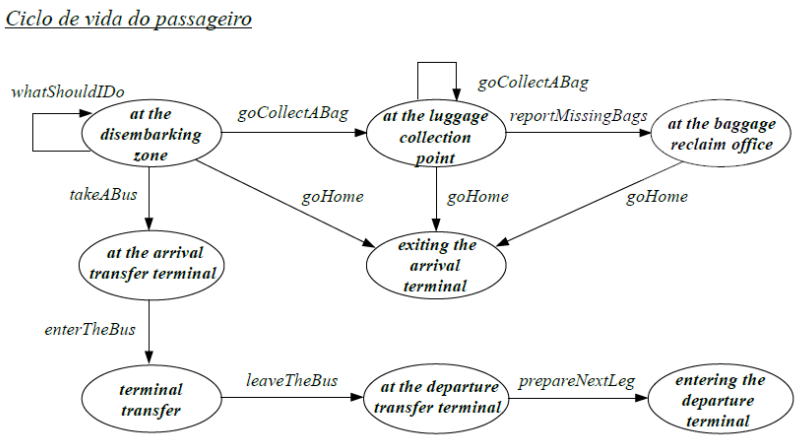

Figura 5.2: Esquema relativo ao Ciclo de vida do Passageiro

A thread *Passenger* executa as seguintes operações:

1. getJourney() - verifica se a viagem chegou efetivamente ao fim ou não;
2. whatShouldIDo() - decide o que o passageiro vai fazer dependendo das situações: se este não está em viagem e tem bagagens então vai recolhe-la; se não está em viagem mas não tem qualquer bagagem então vai-se embora caso contrário vai para a fila de espera para apanhar o autocarro (invoca o método iGotHere() que se encontra no monitor ZoneArrival);
3. goCollectABag() - quando o passageiro já não está em viagem e tem bagagens então vai recolhe-las e caso perca uma delas então terá de reportar o seu desaparecimento (invoca o método getLuggage() que se encontra no monitor ZoneLuggage);
4. reportLuggageMissing() - quando o passageiro reporta o desaparecimento de uma das bagagens perdidas (invoca o método reportLuggageMissing(id, bags) que se encontra no monitor ZoneLuggage)
5. goHome() - quando o passageiro se vai embora porque já não está em viagem e não possui qualquer bagagem (invoca o método goingHome(id) que se encontra no monitor ZoneEntryExit);

6. takeABus() - quando o passageiro deseja apanhar o autocarro (invoca o método takeABus(id) que se encontra no monitor ZoneTransferArrival);

7. enterTheBus() - quando o passageiro entra no autocarro e espera até que este termine (invoca o método finishBusTrip() que se encontra no monitor ZoneTransferDeparture);

8. leaveTheBus() - quando o passageiro abandona o autocarro (invoca o método leaveBus(id) que se encontra no monitor ZoneTransferDeparture);

9. prepareNextLeg() - quando o último passageiro de cada voo acorda todos os passageiros em EXITING_THE_ARRIVAL_TERMINAL ou ENTERING_THE_DEPARTURE_TERMINAL (invoca o método goingHome(id) que se encontra no monitor ZoneEntryExit)

5.4 Driver

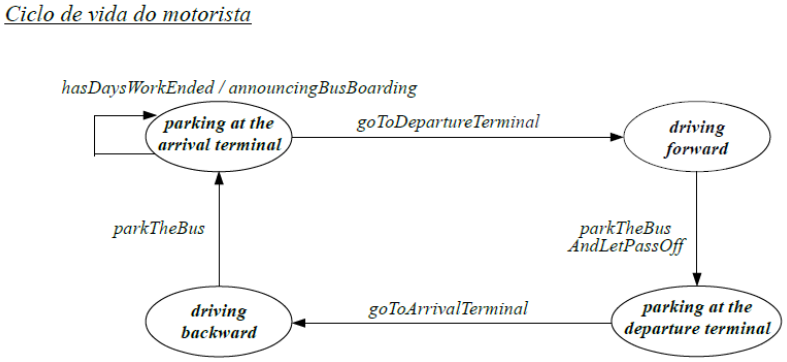

Figura 5.3: Esquema relativo ao Ciclo de vida do Motorista

Por último, a thread *Driver* executa as seguintes operações:

1. void goToDepartureTerminal() - quando o driver parte para a zona de Transferência de embarque (invoca o método depart() que se encontra no monitor ZoneTransferArrival);

2. void parkTheBusAndLetPassOf() - quando o driver estaciona o bus no terminal de embarque (invoca o método park(people) que se encontra no monitor ZoneTransferDeparture);

3. void goToArrivalTerminal() - quando o driver parte para a zona de Transferência de desembarque (invoca o método depart() que se encontra no monitor ZoneTransferDeparture);

4. void parkTheBus() - quando o driver estaciona o bus no terminal de desembarque;

5. boolean hasDaysWorkEnded() - verifica se o dia do motorista chegou ao fim ou não;

6. void announcingBusBoarding() - o driver espera até que hajam passageiros suficientes à espera de entrar no bus e acorda-os para estes entrarem (invoca o método announceBusBoarding() que se encontra no monitor ZoneTransferArrival);

Bibliografia

Referências

[1] *Nested Monitor Lockout*. URL: http://tutorials.jenkov.com/java-concurrency/nested-monitor-lockout.html.

[2] *Reentrant Lock in JAVA*. URL: http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html.

[3] *Synchronized Object*. URL: http://www.tutorialspoint.com/java/java_thread_synchronization.htm.

[4] *Using Reentrant Lock for Thread Synchronization*. URL: http://javabeanz.wordpress.com/2007/07/12/using-reentrant-locks-for-thread-synchronization/.

[5] *Using Reentrant Lock for Thread Synchronization*. URL: http://javabeanz.wordpress.com/2007/07/12/using-reentrant-locks-for-thread-synchronization/.