

1º Semestre 16/17 SI

Fundamentos de Sistemas Operativos

**Relatório do 3º Trabalho Prático**

Eng. Carlos Gonçalves

22/01/2017

**Trabalho realizado por Grupo 11:**

Diogo Fernandes nº39205

Rui Santos nº39286

Hugo Safara nº40614

Índice

[**Introdução** 3](#_Toc472894705)

[**Desenvolvimento** 4](#_Toc472894706)

[**Diagrama de Classes** 4](#_Toc472894707)

[**Tarefas dos Comportamentos** 5](#_Toc472894708)

[**Tarefa vaguear** 5](#_Toc472894709)

[**Tarefa evitar** 8](#_Toc472894710)

[**Caixa Correio** 11](#_Toc472894711)

[**Tarefa Seguir Parede** 12](#_Toc472894712)

[**Tarefa Gestor** 19](#_Toc472894713)

[**Robot Interface Threads** 22](#_Toc472894714)

[**Conclusão** 23](#_Toc472894715)

# **Introdução**

Na sequência da realização dos trabalhos práticos anteriores, tendo-se adquirido conhecimentos sobre comunicação entre processos por memoria partilhada em ficheiros, foi-nos proposto uma nova realização do mesmo trabalho, mas com aplicação de Multitarefas e adição de um novo comportamento ao RobotLego, comportamento esse Seguir Parede.

Os processos leves ou tarefas executam-se só num computador e são partes de um único processo designado por processo-pai que os lança e executa. Os processos leves cooperam dentro dum processo-pai para realizar uma funcionalidade desejada em pseudo-paralelismo num processador de um core ou em paralelismo num processador com múltiplos-cores. Os processos leves concorrem e partilham os recursos do computador e podem partilhar os dados do processo-pai.

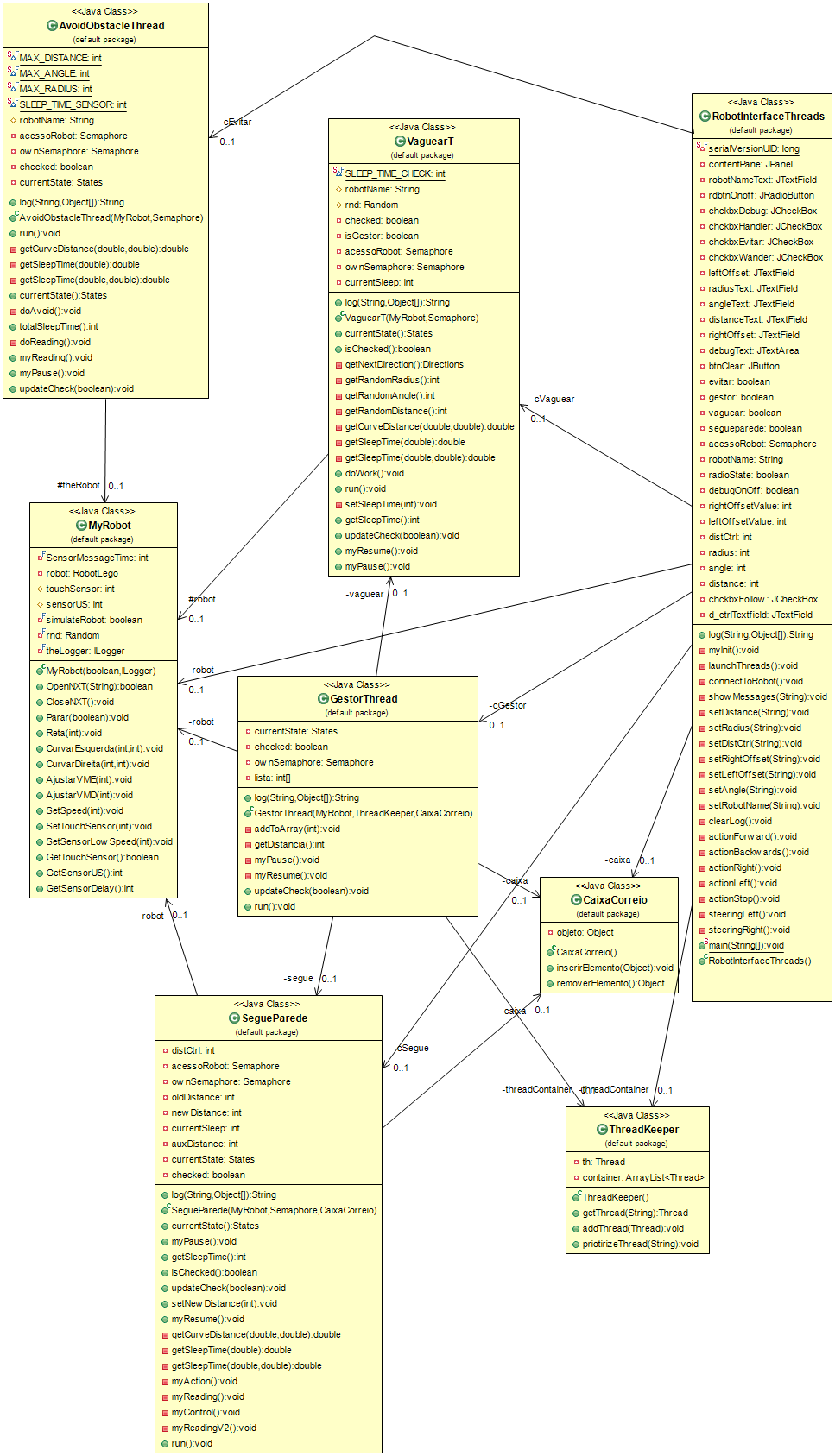
A comunicação entre tarefas exige sincronização, porque o tempo de processamento de cada tarefa e a sua execução é imprevisível num computador, as tarefas executam-se de forma assíncrona. Portanto, para se conseguir êxito na cooperação entre as várias tarefas é preciso haver pontos de sincronismo nas atividades. Um ponto de sincronismo, é um ponto no código em que uma tarefa espera por outra tarefa em relação às atividades a executar.

Para esta implementação iremos usar Semáforos, que é apenas um contentor de um valor inteiro com métodos que permitem decrementar *(acquire())* e incrementar *(release())* o valor atual, sendo que se o valor for igual a 0, irá bloquear o processo/tarefa que execute o método de decremento *(acquire())* até que seja efetuada uma alteração por um outro processo/tarefa *(release()).*

Uma outra implementação parte por o uso de Monitores, têm como objetivo garantir que métodos sejam executados em exclusividade por múltiplos processos e sem a complexidade inerente ao uso de semáforos que são fonte de erros na implementação de aplicações.

# **Desenvolvimento**

## **Diagrama de Classes**



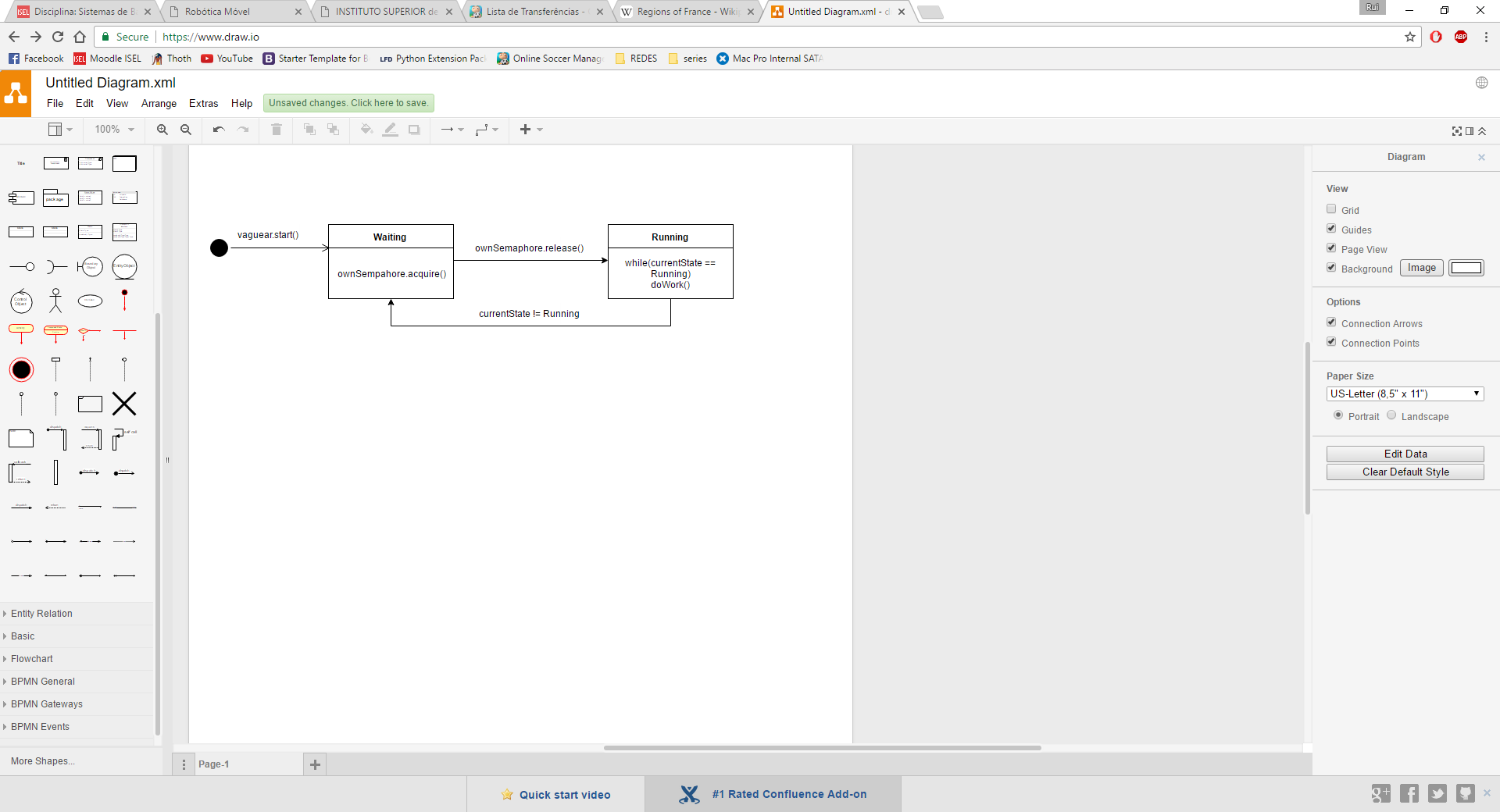
## **Tarefas dos Comportamentos**

Cada Classe em java que estenda de Thread, irá obrigatoriamente implementar o método *run()* sendo o método chamado quando fazemos *Thread.start()*.

Cada tarefa correspondente ao comportamento Vaguear, Evitar Obstáculo, Seguir Parede, tem uma máquina de estados que lhe permite efetivamente comandar no robot e ter um maior controlo sobre quando o fazer.

### **Tarefa vaguear**

Máquina de Estados Vaguear:



Em comparação à classe vaguear composta no segundo trabalho prático, sofre agora algumas alterações para programaticamente ter um comportamento diferente, começamos pela adição dos métodos de controlo da máquina de estados:

public void myResume**()** **{**

**this.**ownSemaphore**.**release**();**

**this.**currentState **=** States**.**Running**;**

**}**

Função principal, dar permissão ao vaguear para iniciar o seu comportamento.

o *ownSemaphore*, o objecto do tipo Semáforo que bloqueia qualquer ação proveniente da tarefa vaguear até que lhe tenha sido dada permissão por outros meios, Gestor ou GUI, com este método.

public void myPause**()** **{**

**this.**robot**.**Parar**(true);**

**this.**currentState **=** States**.**Waiting**;**

**}**

Função principal, parar a tarefa vaguear, e bloqueá-la no estado *waiting* até que lhe seja dada permissão novamente para exercer comandos sobre o Robot.  
Este cessar de permissões pode ser liderado por o Gestor ou a GUI.

public void updateCheck**(**boolean check**)** **{**

**this.**checked **=** check**;**

**if** **(!this.**checked**)** **{**

myPause**();**

**this.**log**(**"MANDEI PARA O WAITING VAGUEAR"**);**

**}** **else** **{**

myResume**();**

**}**

**}**

Este método tem como função ser o meio de “comando” da GUI sobre a tarefa vaguear, alterando o estado da máquina de estados de acordo com uma variável que, na GUI, é equivalente a uma *checkbox* que indica se o comportamento está ativo ou não.

@Override

public void run**()** **{**

**while** **(this.**currentState **!=** States**.**Ending**)** **{**

**switch** **(this.**currentState**)** **{**

**case** Waiting**:**

**this.**log**(**"ESTOU NO WAITING VAGUEAR"**);**

**try** **{**

**this.**ownSemaphore**.**acquire**();**

**}** **catch** **(**InterruptedException e**)** **{**

// TODO Auto-generated catch block

e**.**printStackTrace**();**

**}**

**break;**

**case** Running**:**

**this.**log**(**"ESTOU NO RUNNING VAGUEAR"**);**

**try** **{**

doWork**();**

**}** **catch** **(**Exception e**)** **{**

// TODO Auto-generated catch block

e**.**printStackTrace**();**

**}**

**break;**

**default:**

**break;**

**}**

**}**

**}**

Este é o método referido acima como o que executa a máquina de estados do vaguear, sempre que se usa *Thread.start()*.

public States currentState**()** **{**

**return** **this.**currentState**;**

**}**

public boolean isChecked**()** **{**

**return** **this.**checked**;**

**}**

Sendo *States* um enumerador comum a todas as tarefas, o método *currentState()* devolve o estado em que se encontra atualmente o autómato.

O método booleano *isCheck()* indica se o comportamento foi selecionado na GUI, sendo esta função chamada para termos de exclusão de ações no Gestor.

public VaguearT**(**MyRobot robot**,** Semaphore semaphore**)** **{**

**this.**setName**(**"Vaguear"**);**

**this.**currentDirection **=** Directions**.**Stop**;**

**this.**robot **=** robot**;**

**this.**acessoRobot **=** semaphore**;**

**this.**currentState **=** States**.**Waiting**;**

**this.**rnd **=** **new** Random**();**

**this.**checked **=** **false;**

**this.**ownSemaphore **=** **new** Semaphore**(**0**);**

**}**

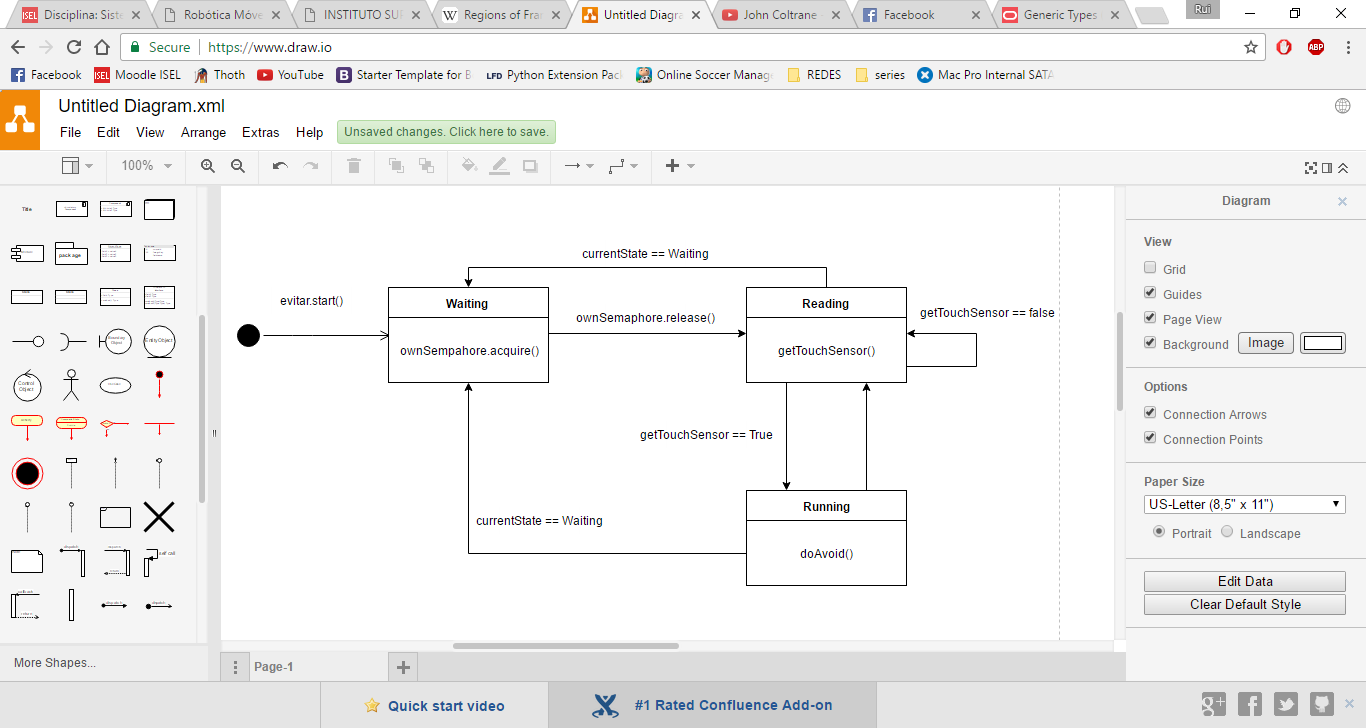
O construtor, face ao do trabalho anterior apresenta as seguintes alterações:  
- Um novo parâmetro de entrada, designado *acessoRobot* que é o semáforo correspondente por a sincronização dos comandos efetuados no robot pelas diferentes tarefas.

- O próprio semáforo *ownSemaphore*, que permite que o autómato da tarefa vaguear tenha um estado bloqueante.

- É dado um nome à *Thread* para mais tarde ser usada numa classe exterior útil ao Gestor.

### **Tarefa evitar**

Máquina de estados do Evitar:



Da mesma maneira que a tarefa vaguear se bloqueia no seu estado *waiting* a tarefa evitar usa o mesmo método.

public AvoidObstacleThread**(**MyRobot robot**,** Semaphore semaphore**)** **{**

**this.**setName**(**"Evitar"**);**

**this.**currentState **=** States**.**Waiting**;**

**this.**theRobot **=** robot**;**

**this.**ownSemaphore **=** **new** Semaphore**(**0**);**

**this.**acessoRobot **=** semaphore**;**

**this.**checked **=** **false;**

**}**

- Nome dado à *Thread* correspondente do comportamento;

- *ownSemaphore* semáforo próprio da tarefa;

- *acessoRobot* semáforo que indica o estado do acesso do robot;

- booleano *check* caso esteja selecionado o comportamento na GUI

@Override

public void run**()** **{**

**while** **(this.**currentState **!=** States**.**Ending**)** **{**

**switch** **(this.**currentState**)** **{**

**case** Waiting**:**

**this.**log**(**"ESTOU NO WAITING EVITAR"**);**

**try** **{**

**this.**ownSemaphore**.**acquire**();**

**}** **catch** **(**InterruptedException e**)** **{**

e**.**printStackTrace**();**

**}**

**break;**

**case** Reading**:**

// this.log("ESTOU NO READING EVITAR");

doReading**();**

**break;**

**case** Running**:**

// this.log("ESTOU NO RUNNING EVITAR");

doAvoid**();**

**break;**

**default:**

**break;**

**}**

**}**

**}**

O código executado sempre que se usa *Thread.start()*, correspondente à máquina de estados.

public void myReading**()** **{**

**this.**ownSemaphore**.**release**();**

**this.**currentState **=** States**.**Reading**;**

**}**

public void myPause**()** **{**

**this.**currentState **=** States**.**Waiting**;**

**}**

Métodos existentes para controlo do comportamento da *Thread* Evitar, apenas e exclusivamente usados pela GUI e não por nenhuma outra *Thread.*

public void updateCheck**(**boolean checkbox**)** **{**

**this.**checked **=** checkbox**;**

**if** **(!this.**checked**)** **{**

myPause**();**

**this.**log**(**"MANDEI PARA O WAITING"**);**

**}** **else** **{**

myReading**();**

**}**

**}**

Método que combina os anteriores de controlo, e sempre chamado quando é pressionada a *checkbox* correspondente ao comportamento Evitar se modo a dar update ás suas ações.

private void doReading**()** **{**

**if** **(this.**theRobot**.**GetTouchSensor**())** **{**

**try** **{**

**this.**acessoRobot**.**acquire**();**

**this.**theRobot**.**Parar**(true);**

**if** **(this.**currentState **!=** States**.**Waiting**)** **{**

**this.**currentState **=** States**.**Running**;**

**}**

**}** **catch** **(**InterruptedException e**)** **{**

// TODO Auto-generated catch block

e**.**printStackTrace**();**

**}**

**}**

// menor o sleep mais rapida a resposta

**try** **{**

// 1000 ms para robot virtual

// 100 para robot fisico

Thread**.**sleep**(**100**);**

**}** **catch** **(**InterruptedException e**)** **{**

// TODO Auto-generated catch block

e**.**printStackTrace**();**

**}**

**}**

Sendo a tarefa evitar independente de qualquer outra, estará sempre a ler o sensor de toque a obter o estado do mesmo, sempre que se regista o estado equivalente a 1 (true), logo em seguida começa por tomar controlo do Robot para se poder executar comandos.  
Lê-se o sensor a cada 100 milissegundos, para maior resposta num choque com um obstáculo, esforçando menos o motor do Robot se já tiver uma ação a decorrer.

Sempre que é tomada posse do robot, é executado o comando reta, esperando o tempo do mesmo, em seguida uma curva, a cada ação executado para robot deve-se dar *release()* do acesso, dizendo ás restantes tarefas competindo pelo acesso ao robot, que o canal se encontra livre.

private void doAvoid**()** **{**

**this.**theRobot**.**Reta**(-**MAX\_DISTANCE**);**

**try** **{**

Thread**.**sleep**((**int**)** **(**getSleepTime**(**MAX\_DISTANCE**)** **\*** 1000**));**

**}** **catch** **(**InterruptedException e**)** **{**

// TODO Auto-generated catch block

e**.**printStackTrace**();**

**}**

**this.**theRobot**.**CurvarEsquerda**(**MAX\_RADIUS**,** MAX\_ANGLE**);**

**try** **{**

Thread**.**sleep**((**int**)** **(**getSleepTime**(**MAX\_ANGLE**,** MAX\_RADIUS**)** **\*** 1000**));**

**}** **catch** **(**InterruptedException e**)** **{**

// TODO Auto-generated catch block

e**.**printStackTrace**();**

**}**

**if** **(this.**currentState **!=** States**.**Waiting**)** **{**

**this.**currentState **=** States**.**Reading**;**

**}**

**this.**acessoRobot**.**release**();**

**}**

### **Caixa Correio**

Esta caixa de correio, tem como principal objetivo, fornecer distâncias à tarefa Seguir Parede e receber distâncias do Gestor.  
Para o melhor funcionamento das tarefas com esta classe, usa-se monitores, sendo assim possível filtrar melhor os valores que aparecem em cada tarefa e a sincronização entre as trocas de, neste caso, valores inteiros de distâncias.

public class CaixaCorreio **{**

private Object objeto**;**

public CaixaCorreio**()** **{**

objeto **=** **null;**

**}**

public synchronized void inserirElemento**(**Object elem**)** **{**

**this.**objeto **=** elem**;**

**this.**notifyAll**();**

**}**

public synchronized Object removerElemento**()** **throws** InterruptedException **{**

**while** **(** **this.**objeto**==null** **)** **{**

**this.**wait**();**

**}**

Object result **=** **this.**objeto**;**

**this.**objeto **=** **null;**

**return** result**;**

**}**

**}**

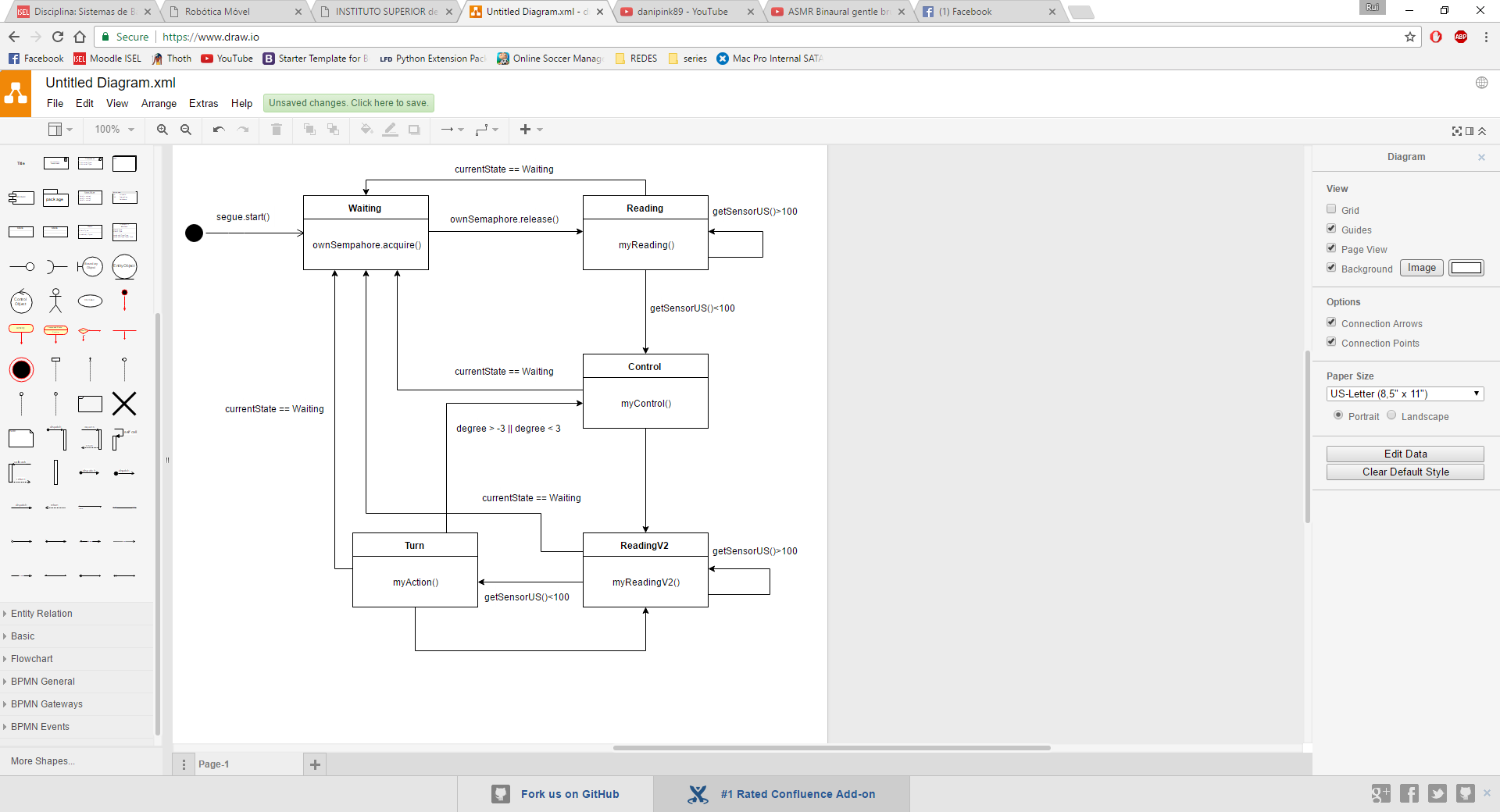
No método inserir, apenas guardamos o Objeto que recebemos como parâmetro de entrada, sem saber qual o seu tipo, e em seguida chamamos uma função própria do uso de monitores, o *notifyAll()* acorda todas as tarefas que estivessem a competir pelo acesso a esse novo objeto através do método *removerElemento()* que enquanto a variável for null, bloqueio a tarefa que está nesse momento a usar o método.

### **Tarefa Seguir Parede**

Como dito no enunciado, o comportamento Seguir Parede é a única tarefa que tem uma relação de dependência com o Gestor, visto ser necessário o uso do Sensor de Ultrassons (medidor de distâncias) em que tal só pode ser acedido pela tarefa Gestor.

O uso da caixa de correio, permite ao Seguir efetuar os cálculos necessários para a sua ação com maior precisão e controlo.

Máquina de Estados do Seguir Parede:



Como em todas as outras tarefas existem os métodos de controlo de fluxo da máquina de estados:

public void myResume**()** **{**

**this.**ownSemaphore**.**release**();**

**this.**currentState **=** States**.**Reading**;**

**}**

public boolean isChecked**(){**

**return** **this.**checked**;**

**}**

public void updateCheck**(**boolean check**)** **{**

**this.**checked **=** check**;**

**if** **(!this.**checked**)** **{**

myPause**();**

**}** **else** **{**

myResume**();**

**}**

Segue-se o Construtor da tarefa:

public SegueParede**(**MyRobot robot**,** Semaphore semaphore**,** CaixaCorreio caixa**)** **{**

**this.**setName**(**"Segue"**);**

**this.**robot **=** robot**;**

**this.**acessoRobot **=** semaphore**;**

**this.**oldDistance **=** 0**;**

**this.**newDistance **=** 0**;**

**this.**distCtrl **=** 0**;**

**this.**currentSleep **=** 0**;**

**this.**auxDistance **=** 0**;**

**this.**checked **=** **false;**

**this.**currentState **=** States**.**Waiting**;**

**this.**ownSemaphore **=** **new** Semaphore**(**0**);**

**this.**caixa **=** caixa**;**

**}**

Este construtor recebe como parâmetros em comum com as tarefas Vaguear, Evitar e Gestor, sendo que a CaixaCorreio apenas em comum com o Gestor.

Foram criadas duas variáveis *newDistance* e *oldDistance* que simbolizam as distâncias medidas para os cálculos do angulo a ser reajustado a cada movimento.

public void setNewDistance**(**int distanceRead**)** **{**

**this.**oldDistance **=** **this.**newDistance**;**

**this.**newDistance **=** distanceRead**;**

**}**

Método responsável pela troca dos valores das distâncias entre as variáveis, visto que já se sabe à partida quando a primeira distância é medida, e teremos uma segunda distância, essa será considerada a distância mais recente.

private void myReading**()** **throws** Exception**{**

auxDistance **=** **(**Integer**)**caixa**.**removerElemento**();**

**this.**log**(**"LI Primeira DISTANCIA: " **+** auxDistance**);**

**if** **(**auxDistance **<** 100 **&&** auxDistance **>** 20**)** **{**

**try** **{**

**this.**acessoRobot**.**acquire**();**

setNewDistance**(**auxDistance**);**

**}** **catch** **(**InterruptedException e**)** **{**

// TODO Auto-generated catch block

e**.**printStackTrace**();**

**}**

**if** **(this.**currentState **!=** States**.**Waiting**)** **{**

**this.**currentState **=** States**.**Control**;**

**}**

**}** **else** **{**

**if** **(this.**currentState **!=** States**.**Waiting**)** **{**

**this.**currentState **=** States**.**Reading**;**

**}**

**try** **{**

Thread**.**sleep**(**500**);**

**}** **catch** **(**InterruptedException e1**)** **{**

// TODO Auto-generated catch block

e1**.**printStackTrace**();**

**}**

**}**

**}**

Este método corresponde ao estado *Reading*, estado esse que apenas acontece uma vez enquanto não temos distâncias lidas, depois o autómato para saber uma nova distância o resto do seu funcionamento, depende apenas do estado *ReadingV2* que é igual ao *Reading*.  
A distância lida é obtida pela CaixaCorreio em comum com o gestor, estando sempre a pedir esta distância duplamente mais lento que a leitura do Gestor, respeitando o ritmo de nyquist, por isso se usa *Thread.sleep(500)* no final de cada pedido.

private void myControl**()** **{**

**this.**robot**.**Reta**(**20**);**

**this.**robot**.**Parar**(false);**

**this.**acessoRobot**.**release**();**

**try** **{**

Thread**.**sleep**(**getSleepTime**()** **\*** 1000**);**

**}** **catch** **(**InterruptedException e**)** **{**

// TODO Auto-generated catch block

e**.**printStackTrace**();**

**}**

**if** **(this.**currentState **!=** States**.**Waiting**)** **{**

**this.**currentState **=** States**.**ReadingV2**;**

**}**

**}**

Este método corresponde ao estado *Control*, que anda uma distância de controlo assim que obtém uma primeira medição válida, esta distância de controlo é percorrida de forma a obter uma segunda distância do sensor de Ultrassons que irá ser usada para calcular o ângulo e virar o robot de modo a que este fique paralelo à parede.  
Este angulo é calculado com dois catetos de um triângulo imaginário, utilizando a função *Arctg()*.

Este é o método que corresponde ao estado *Turn* em que o robot efetivamente calcula o angulo ajustar, e em seguida irá andar a distânbcia de controlo.

private void myAction**()** **{**

double radAngle **=** **((**double**)** **this.**oldDistance **-** **(**double**)** **this.**newDistance**)** **/** **(**double**)** **this.**distCtrl**;**

double radianos **=** Math**.**atan**(**radAngle**);**

double degrees **=** **(**radianos **\*** 360**)** **/** **(**2 **\*** Math**.**PI**);**

**this.**log**(**"--------------------------------- angulo calculado: " **+** degrees**);**

**if** **(**degrees **>** 3 **||** degrees **<** **-**3**)** **{**

**if** **(this.**oldDistance **>** **this.**newDistance**)** **{**

**this.**robot**.**CurvarEsquerda**(**0**,** **(**int**)** degrees**);**

**this.**robot**.**Parar**(false);**

**this.**acessoRobot**.**release**();**

**try** **{**

Thread**.**sleep**((**int**)** getSleepTime**((**int**)** degrees**,** 0**)** **\*** 1000**);**

**}** **catch** **(**InterruptedException e**)** **{**

// TODO Auto-generated catch block

e**.**printStackTrace**();**

**}**

**}** **else** **{**

**this.**robot**.**CurvarDireita**(**0**,** **(**int**)** degrees**);**

**this.**robot**.**Parar**(false);**

**this.**acessoRobot**.**release**();**

**try** **{**

Thread**.**sleep**((**int**)** getSleepTime**((**int**)** degrees**,** 0**)** **\*** 1000**);**

**}** **catch** **(**InterruptedException e**)** **{**

// TODO Auto-generated catch block

e**.**printStackTrace**();**

**}**

**}**

**if** **(this.**currentState **!=** States**.**Waiting**)** **{**

**this.**currentState **=** States**.**ReadingV2**;**

**}**

**}** **else** **{**

**if** **(this.**currentState **!=** States**.**Waiting**)** **{**

**this.**currentState **=** States**.**Control**;**

**}**

**}**

**}**

Método chamada a cada vez que se executa *segue.start()*.

@Override

public void run**()** **{**

**while** **(this.**currentState **!=** States**.**Ending**)** **{**

**switch** **(this.**currentState**)** **{**

**case** Waiting**:**

**this.**log**(**"ESTOU NO WAITING DO SEGUEPAREDE"**);**

**this.**log**(**"ESTADO A PRINTAR " **+** **this.**currentState**);**

**try** **{**

**this.**ownSemaphore**.**acquire**();**

**}** **catch** **(**Exception e**)** **{**

e**.**printStackTrace**();**

**}**

**break;**

**case** Reading**:**

**this.**log**(**"ESTADO A PRINTAR " **+** **this.**currentState**);**

**try** **{**

myReading**();**

**}** **catch** **(**Exception e**)** **{**

// TODO Auto-generated catch block

e**.**printStackTrace**();**

**}**

**break;**

**case** Control**:**

**this.**log**(**"ESTADO A PRINTAR " **+** **this.**currentState**);**

myControl**();**

**break;**

//

**case** ReadingV2**:**

**this.**log**(**"ESTADO A PRINTAR " **+** **this.**currentState**);**

**try** **{**

myReadingV2**();**

**}** **catch** **(**Exception e**)** **{**

// TODO Auto-generated catch block

e**.**printStackTrace**();**

**}**

**break;**

**case** Turn**:**

**this.**log**(**"ESTADO A PRINTAR " **+** **this.**currentState**);**

myAction**();**

**break;**

**default:**

**break;**

**}**

**}**

**}**

### **Tarefa Gestor**

public GestorThread**(**MyRobot robot**,** ThreadKeeper lista**,** CaixaCorreio caixa**)** **{**

**this.**setName**(**"Gestor"**);**

**this.**checked **=** **false;**

**this.**threadContainer **=** lista**;**

**this.**currentState **=** States**.**Init**;**

**this.**vaguear **=** **(**VaguearT**)** **this.**threadContainer**.**getThread**(**"Vaguear"**);**

**this.**segue **=** **(**SegueParede**)** **this.**threadContainer**.**getThread**(**"Segue"**);**

**this.**robot **=** robot**;**

**this.**ownSemaphore **=** **new** Semaphore**(**0**);**

**this.**lista **=** **new** int**[**5**];**

**this.**caixa **=** caixa**;**

**}**

Esta tarefa vai ter como função a definição de prioridades ás tarefas Seguir Parede e Vaguear sobre o controlo do robot das mesmas.  
Para isso ele tem de ter conhecimento das tarefas que ele deve controlar, a criação de uma classe *ThreadKeeper* que funciona como um contentor de *threads* para o nosso trabalho.

Gestor tem, como se havia referido, em comum a CaixaCorreio que lhe permite produzir distancias para o Seguir Parede as consumir.

É implementado aqui um filtro de mediana sobre as distâncias a ler do sensor, pois como qualquer componente hardware tem falhas nas suas leituras, o nosso filtro de mediana tem tamanho 5, significa que antes de colocar uma distância na Caixa de Correio para o seguir parede a consumir, ele lê 5 vezes do sensor e devolve a distância presente na posição do meio do array de distâncias lidas, após ter sido ordenado.

private int getDistancia**(){**

int aux**;**

aux **=**0**;**

//aplicar filtro de mediana

**while(**aux **<** lista**.**length**){**

addToArray**(this.**robot**.**GetSensorUS**());**

aux**++;**

**}**

int lista2**[]** **=** lista**;**

Arrays**.**sort**(**lista2**);**

**return** lista2**[**lista2**.**length**/**2**];**

**}**

É usado um método auxiliar, que coloca a nova distância lida na primeira posição e shifta uma posição a valores existentes.

Tal como os outros autómatos das tarefas, o Gestor tem também um estado *Waiting* com o seu próprio semáforo, para ser bloqueado à espera de permissão para trabalhar.

Estado *fillArray* – Este estado tem como função correr uma única vez, quando o gestor é iniciado, o seu array de valores é nulo, tendo necessidade de ser preenchido com valores.

**case** fillArray**:**

int aux**;**

aux **=**0**;**

//aplicar filtro de mediana

**while(**aux **<** lista**.**length**){**

addToArray**(this.**robot**.**GetSensorUS**());**

aux**++;**

**}**

**this.**currentState **=** States**.**SegueParede**;**

**break;**

Estado *SegueParede* :

**case** SegueParede**:**

distancia **=** getDistancia**();**

**if(**distancia **<**100**){**

**this.**caixa**.**inserirElemento**(new** Integer**(**distancia**));**

**this.**currentState **=** States**.**SegueParede**;**

**}else{**

**this.**currentState **=** States**.**Vaguear**;**

**if(this.**segue**.**isChecked**()){**

**this.**segue**.**myPause**();**

**}**

**if(this.**vaguear**.**isChecked**()){**

**this.**vaguear**.**myResume**();**

**}**

**}**

**try** **{**

Thread**.**sleep**(**250**);**

**}** **catch** **(**InterruptedException e**)** **{**

// TODO Auto-generated catch block

e**.**printStackTrace**();**

**}**

**break;**

A cada estado do Gestor ele tem de ler distâncias, de modo a conseguir perceber se deve permanecer no estado seguir parede ou deve avançar para o estado vaguear.

Se a condição do Seguir Parede estiver válida, então o gestor irá por em pausa o vaguear e resumir ações no SeguirParede.

Todas estas trocas são verificadas com um método de cada tarefa que nos indica se está autorizada a cumprir ações ou não.

Estado *Vaguear*:

**case** Vaguear**:**

distancia **=** getDistancia**();**

**if(**distancia **<**100**){**

**this.**caixa**.**inserirElemento**(new** Integer**(**distancia**));**

**this.**currentState **=** States**.**SegueParede**;**

**if(this.**segue**.**isChecked**()){**

**this.**segue**.**myResume**();**

**}**

**if(this.**vaguear**.**isChecked**()){**

**this.**vaguear**.**myPause**();**

**}**

**}else{**

**this.**currentState **=** States**.**Vaguear**;**

**}**

**try** **{**

Thread**.**sleep**(**250**);**

**}** **catch** **(**InterruptedException e**)** **{**

// TODO Auto-generated catch block

e**.**printStackTrace**();**

**}**

**break;**

Semelhante ao estado seguir parede, apenas trocam as prioridades dos comportamentos.

## **Robot Interface Threads**

Para que esta composição de multitarefas funcionasse, foi necessário a criação de um método que inicia-se todas as *threads* quando a conexão do robot fosse estabelecida.

private void launchThreads**()** **{**

// SET AOS SENSORES APENAS QUANDO LIGAMOS O ROBOT

**this.**robot**.**SetTouchSensor**(**RobotLego**.**S\_2**);**

**this.**robot**.**SetSensorLowSpeed**(**RobotLego**.**S\_1**);**

**this.**cVaguear **=** **new** VaguearT**(this.**robot**,** **this.**acessoRobot**);**

**this.**cVaguear**.**start**();**

**this.**cEvitar **=** **new** AvoidObstacleThread**(this.**robot**,** **this.**acessoRobot**);**

**this.**cEvitar**.**start**();**

**this.**cSegue **=** **new** SegueParede**(this.**robot**,** **this.**acessoRobot**,** **this.**caixa**);**

**this.**cSegue**.**start**();**

**this.**threadContainer**.**addThread**(this.**cVaguear**);**

**this.**threadContainer**.**addThread**(this.**cSegue**);**

**this.**cGestor **=** **new** GestorThread**(this.**robot**,** **this.**threadContainer**,** **this.**caixa**);**

**this.**cGestor**.**start**();**

showMessages**(this.**log**(**"lan篵 threads"**));**

**}**

Bem como para que fosse possível a intervenção do utilizador nos comportamentos do robot, a cada ação proveniente da interface gráfica manualmente, ocupa o canal definido como um semáforo *acessoRobot*, e em seguida liberta-o.

public void actionPerformed**(**ActionEvent e**)** **{**

**try** **{**

acessoRobot**.**acquire**();**

**}** **catch** **(**InterruptedException ex**)** **{**

// TODO Auto-generated catch block

ex**.**printStackTrace**();**

**}**

steeringLeft**();**

steeringRight**();**

actionForward**();**

acessoRobot**.**release**();**

showMessages**(**"Forward -> " **+** distance **+** "cm."**);**

**}**

# **Conclusão**

Na realização deste trabalho foram conseguidos os aspetos pedidos pelo enunciado do trabalho, ainda que com algumas dificuldades.

No inicio do dimensionamento da solução em torno do Gestor, pensava-se que uma simples troca de métodos entre as tarefas, sem qualquer uso de semáforo seria possível, mas além de desrespeitar o enunciado, foi posto em prática a solução e reparamos que por não haver sistema de sincronização entre as ações diferentes de cada tarefa, os resultados não saíram coerentes com o previsto estando sempre todas fora de tempo e a executar ações quando não deviam.

Após um sistema de sincronização implementado, pensou-se que seria suficiente, mas encontramo-nos com problemas na comunicação de distâncias a partir do Gestor para o Seguir Parede, para isso criou-se a caixa de correio com monitores, sendo uma maneira de sincronizar os pedidos e os envios de distâncias entre as duas tarefas.

Tentou-se criar uma maneira de seguir sem ser o Gestor a produzir as distâncias, mas assim iriamos desrespeitar o enunciado, para além de acordo com a nossa implementação, dificultar uma nova adição de comportamento.