MEIC-A
Sistemas de Automação
2011 / 2012

MobotArch

Projecto de
Arquitectura Móvel de Robôs

Manual de Programador

Grupo de Trabalho:

- Gil Lacerda Nr.º 64761
- Rui Lageira Nr.º 64860

I. Descrição da Arquitectura

Optou-se por descrever a Arquitectura por uma abordagem que percorre cada módulo que a compõe. Estes módulos são referidos como pacotes (seguindo a nomeclatura do Java de *package*), sobre os quais se refere o papel de cada um deles na Arquitectura e se descrevem as classes neles contidas.

1. Pacote de Utilidades (*util*)

Módulo que inclúi os parâmetros globais e funções auxiliares utilizados na arquitectura e encontram-se divididos por várias classes que agrupam utilidades por domínios lógicos:

CommunicationOptions – contém as variáveis que definem os portos de cliente e servidor utilizados na comunicação e uma flag de controlo para decidir se o módulo do lado do servidor ("mestre") também actua no motor (`remoteOnly = false`) ou se apenas os seus clientes ("escravos") actuam os seus motores (`remoteOnly = true`). Contém também uma cadeia de caracteres que determina os clientes aos quais o servidor envia as ordens: se a cadeia for "BROADCAST", o servidor difunde as ordens a todos os clientes de que tem conhecimento; se for um endereço IP, envia a ordem apenas ao cliente a ele associado; se for uma cadeia nula, não envia nada.

CommunicationUtil – mantém funções auxiliares ao módulo da comunicação.

EngineOptions – criado com a intenção de manter parâmetros globais associados a motores.

FrequencyOptions – contém parâmetros de tempos de espera das várias tarefas.

KeyboardOptions – contém parâmetros associados ao controlo por teclado.

ParsingUtil – contém funções auxiliares que analisam os argumentos definidos pelo utilizador, quando corre os programas principais.

SerialUtil – contém funções auxiliares de comunicação série: `getSerialPort` procura as portas séries disponíveis no sistema e devolve ou a primeira ou a especificada pelo programador (se for encontrada alguma); `send` e `receive` são as funções de envio e recepção de `bytes` de uma porta série, respectivamente.

SocketUtil – contém funções auxiliares de comunicação por sockets, particularmente sockets associados ao protocolo de transporte UDP, que é utilizado na comunicação entre servidor e cliente: `send` e `receive` são as funções de envio e recepção de datagramas UDP fazendo uso de `DatagramSocket`'s para o efeito.

TypeUtil – contém funções auxiliares de tradução de `bytes` noutro tipo ou vice-versa: `byte[]` para `double`, `double` para `byte[]`, `byte[]` para `Object`, `Object` para `byte[]`, materializando as primitivas que permitem serializar e desserializar objectos java (por exemplo comandos).

VisibilityOptions – contém *flags* globais de visibilidade do estado de execução do programa, informações para debug e janelas de imagem da câmara Web.

VisionOptions – contém parâmetros globais associados ao módulo de visão como id do dispositivo associado à câmara, dimensões da imagem da câmara e parâmetros de cálculo do centróide para efeitos de orientação do motor.

WebCameraUtil – contém funções auxiliares relacionadas com a câmara: `captureCamera` realiza a captura da câmara, utilizando bibliotecas de OpenCV; `computeCentroids` recebe uma imagem a preto e branco e devolve um mapa de centróides pelo cálculo de contornos e de rectângulos de área mínima que os incluem, cujo centro os definem.

2. Pacote de Excepções (***exception***)

Módulo que inclui as excepções que são lançadas aquando do funcionamento impróprio ou exceptional da arquitectura. As excepções são auto-explicativas, pelas mensagens que imprimem.

3. Pacote de DTOs (Objectos de Transferência de Dados) (***dto***)

Módulo em que se encontram todas as classes de objectos de transferência de dados. Estas classes servem para partilhar dados entre módulos independentes que controlam diferentes partes do robô.

- (***package dto***) encapsula os objectos de transferência de dados dos quais todos os outros herdam, encapsulando assim a funcionalidade comum a todos eles. O propósito destes objectos é a troca de informação entre os módulos independentes de um controlador.

DTO – classe de DTO mais genérica, possui apenas um booleano indicando o seu estado de actualização.

InputDTO – subclasse de DTO que encapsula as variáveis comuns a todos os dtos de entrada dos módulos. De momento encapsula apenas um booleano com o propósito de mandar uma ordem de desligar o módulo.

OutputDTO – subclasse de DTO que encapsula as variáveis comuns a todos os dtos de saída dos módulos. De momento encapsula apenas um booleano com o propósito de informar o controlador que o módulo se desligou.

Synchronized - classe que permite a leitura e escrita sincronizada de uma subclasse de DTO, permitindo a sua partilha entre módulos independentes. Para o funcionamento correcto desta classe, todos as classes de DTOs devem ter uma função que devolve uma cópia profunda da instância.

- (***package dto.communication***) encapsula os objectos de transferência de dados utilizados na partilha de dados com módulos de comunicação.

ClientInputDTO – subclasse de InputDTO que encapsula as variáveis de entrada dos módulos de cliente, receptores de mensagens.

ClientOutputDTO – subclasse de OutputDTO que encapsula as variáveis de saída dos módulos de cliente, receptores de mensagens.

ServerInputDTO – subclasse de InputDTO que encapsula as variáveis de entrada dos módulos de servidor, emissor de mensagens.

ServerOutputDTO – subclasse de OutputDTO que encapsula as variáveis de saída dos módulos de servidor, emissor de mensagens.

- (***package dto.engine***) encapsula os objectos de transferência de dados utilizados na partilha de dados com módulos de motores.

EngineInputDTO – subclasse de InputDTO que encapsula as variáveis de entrada dos módulos de motores.

EngineOutputDTO – subclasse de OutputDTO que encapsula as variáveis de saída dos módulos de motores.

- (***package dto.keyboard***) encapsula os objectos de transferência de dados utilizados na partilha de dados com módulos de teclado.

KeyboardInputDTO – subclasse de InputDTO que encapsula as variáveis de entrada dos módulos de teclado.

KeyboardOutputDTO – subclasse de OutputDTO que encapsula as variáveis de saída dos módulos de teclado.

- (***package dto.vision***) encapsula os objectos de transferência de dados utilizados na partilha de dados com módulos de visão.

VisionInputDTO – subclasse de InputDTO que encapsula as variáveis de entrada dos módulos de visão.

VisionOutputDTO – subclasse de OutputDTO que encapsula as variáveis de saída dos módulos de visão.

4. Pacote de Comandos (*command*)

Módulo que consta de vários níveis de abstracção:

- (***package command***) encapsula os comandos de qualquer entidade genérica passível de os realizar, permitindo a extensão de qualquer entidade comandável:

Command – interface de maior nível que representa um comando genérico.

Query – interface que representa um comando genérico com a particularidade de esperar uma resposta subjacente à sua execução.

VirtualCommand – interface que agrupa uma representação de um comando genérico.

- (***package command.engine***) encapsula os comandos de um motor genérico, permitindo a extensão de vários motor diferentes:

EngineCommand e ***EngineQuery***– interfaces análogas às `Command` e `Query`, mas representam comandos de um motor genérico.

SpecificCommand – interface que representa um comando especificado directamente pelo programador.

SpecificQuery – o mesmo que `SpecificCommand`, mas com a diferença análogo entre `Command` e `Query`.

VirtualEngineCommand e ***VirtualEngineQuery*** – enumeração de `EngineCommand's` e `EngineQuery's` respectivamente, que facilitam a criação de comandos dos motores.

Battery – é um comando `EngineQuery` que pretende obter informação genérica acerca do estado de bateria de um motor.

Direction – é um comando `EngineQuery` que pretende obter informação genérica acerca da direcção de movimento que um motor confere a uma entidade que se move.

Speed – é um comando `EngineQuery` que pretende obter informação genérica acerca da velocidade de movimento que um motor confere a uma entidade que se move.

Move – é um comando que pretende conferir movimento genérico a ser aplicado por um motor.

On – comando para ligar um motor.

Off – comando para desligar um motor.

EngineCommandFactory e ***EngineQueryFactory*** – fábricas que criam os comandos acima descritos.

- (***package command.engine.md25***) encapsula os comandos concretos de um motor guiado pelo controlador MD25 (ver ficha técnica):

MD25SpecificCommand e ***MD25SpecificQuery*** – implementam respectivamente `SpecificCommand` e `SpecificQuery`. São comandos em que o programador especifica directamente os `bytes` que um motor envia para o controlador MD25.

MD25Battery – utiliza um motor para enviar o comando *GET VOLTS* do controlador MD25. Fá-lo traduzindo a `MD25EngineInstruction` correspondente em `bytes` a ser enviados pelo motor para o controlador MD25.

MD25Direction – utiliza um motor para enviar os comandos *GET SPEED 1* e *GET SPEED 2* do seu controlador MD25. Obtém o módulo da diferença entre os valores das respostas.

MD25Speed – realiza o mesmo que MD25Direction mas obtém o menor dos valores de velocidades que o controlador comunica.

MD25Move – utiliza um motor para enviar os comandos *SET SPEED 1* e *SET SPEED 2* do seu controlador.

MD25On – simula a ligação de um motor.

MD25Off – simula o desligar de um motor.

MD25EngineCommandFactory e ***MD25EngineQueryFactory*** – fábricas que criam os comandos acima descritos.

5. Pacote de Módulos (*module*)

Módulo que contém classes que correspondem a periféricos físicos no robô. Estas classes enviam ao controlador percepções através das quais este se apercebe do ambiente e recebem dele instruções que controlam o seu comportamento:

- (***package module***) contém classes genéricas que definem tipos de módulos, estabelecendo a estrutura do seu comportamento.

IndependentModule – encapsula o comportamento básico de um módulo que funciona de forma paralela e independente do controlador, deixando apenas as porções mais específicas do seu comportamento por especificar.

- (***package module.communication***) contém classes de módulos dedicadas à comunicação de mensagens.

Client – subclasse de IndependentModule, adiciona ao comportamento herdado comportamento relacionado com a recepção de mensagens através de um porto virtual do computador.

Server – subclasse de IndependentModule, adiciona ao comportamento herdado comportamento relacionado com o envio de mensagens através de um porto virtual do computador.

- (***package module.communication.udp***) contém classes de módulos dedicadas à comunicação de mensagens através de UDP.

UDPClient – subclasse de Client, implementa a recepção de mensagens através de UDP, sendo estas as percepções partilhadas com o controlador.

UDPServer – subclasse de Server, implementa o envio de mensagens através de UDP, sendo estas as instruções enviadas pelo controlador.

- (***package module.engine***) contém classes de módulos dedicados a periféricos motores.

Engine – subclasse de IndependentModule, adiciona ao comportamento herdado comportamento relacionado com o controlo e monitorização do funcionamento de um motor genérico. Contém classes e funções que permitem a criação de comandos específicos do motor.

- (***package module.engine.md25***) contém classes de módulos dedicados a motores geridos pelo controlador físico MD25.

MD25Engine – subclasse de Engine, implementa o controlo e monitorização de um controlador de hardware MD25. Recebe instruções de movimento e consultas ao estado dos motores. O resultado destas consultas é partilhado com o seu controlador.

MD25EngineInstruction – enumeração dos comandos específicos de um controlador MD25. Tem o propósito de auxiliar a conversão de comandos virtuais genéricos para comandos reais específicos do controlador MD25.

MD25EngineTranslator – classe com o propósito de abstrair a tradução dos

comandos virtuais genéricos para comandos reais, específicos do controlador MD25, e também a tradução do retorno das consultas para um formato standard de forma a que possa ser lido pelo controlador virtual do robô. Contém assim a informação completa referente aos códigos binários que correspondem a instruções do controlador MD25.

- (***package module.keyboard***) contém classes de módulos dedicados à leitura do input de teclados.

Keyboard – subclasse de IndependentModule, adiciona ao comportamento herdado comportamento relacionado com a leitura do input de um teclado de uma forma genérica.

Key – classe que permite encapsular um tipo como sendo um código correspondente a uma tecla pressionada no teclado.

- (***package module.keyboard.windowed***) contém classes de módulos dedicados à leitura do input de teclados através de uma janela.

WindowedKeyboard – subclasse de Keyboard, implementa a leitura imediata do input do teclado através de uma janela de OpenCV, permitindo apresentar uma imagem com instruções. Envia ao controlador a tecla que foi pressionada a cada período.

- (***package module.vision***) contém classes de módulos dedicados à captura de imagens do ambiente e de dados visuais relacionados.

Vision – subclasse de IndependentModule, adiciona ao comportamento herdado comportamento relacionado com a obtenção e processamento de uma imagem que caracteriza o ambiente em que se encontra o robô.

CentroidMap – classe que permite encapsular um tipo como sendo um centróide usado como chave numa instância de Map que mapeia centróides para a sua área na imagem.

Image – classe que permite encapsular um tipo como sendo uma imagem.

- (***package module.vision.webcamera***) contém classes de módulos dedicados à captura de imagens do ambiente e de dados visuais relacionados através de uma câmara web.

WebCameraVision – subclasse de Vision, implementa a obtenção e processamento de uma imagem através de uma câmara web utilizando as biblitecas do OpenCV. Fornece ao controlador a imagem obtida assim como o mapa de centróides obtido através dessa imagem. Pode receber dele parâmetros de processamento da imagem.

6. Pacote de Controladores (*controller*)

Módulo que contém as classes de controlo responsáveis pelo controlo dos processos de análise do ambiente, de tomada de decisões relativas a acções a realizar e de execução dessas acções. A análise e execução de comandos é feita através da comunicação com módulos que representam os periféricos.

- (***package controller***) contém diferentes classes de controladores, genéricas e específicas a uma montagem.

Controller – encapsula o comportamento básico de um controlador que controla periféricos, através de módulos virtuais dedicados, num ciclo de consultas, tomada de decisão e comando dos mesmos.

MasterController – subclasse de Controller, estende o mesmo com um módulo de comunicações servidor e com o comportamento básico de criação e paragem deste.

SimpleMD25MasterController – subclasse de MasterController, adiciona um

módulo de motor MD25 e comportamento básico de criação e paragem deste e de controlo genérico da combinação de módulos actuadores de motor e servidor. O servidor envia qualquer instrução a ser transmitida aos motores aos seus clientes. Este envio e o envio da instrução aos motores são opcionais, podendo realizar-se apenas um ou ambos.

WCamMD25MasterController – subclasse de SimpleMD25MasterController, adiciona um módulo de visão através de câmara web, comportamento básico de criação e paragem deste e o código de controlo restante que permite analisar os centróides obtidos da imagem para controlar o comportamento dos motores do robô "mestre" e dos seus "escravos".

WKeyMD25MasterController – subclasse de SimpleMD25MasterController, adiciona um módulo de teclado utilizando uma janela, comportamento básico de criação e paragem deste e o código de controlo restante que permite analisar as teclas pressionadas para controlar o comportamento dos motores do robô "mestre" e dos seus "escravos".

SlaveController – subclasse de controller, extende o mesmo com um módulo de comunicações cliente e com o comportamento básico de criação e paragem deste.

MD25SlaveController – subclasse de SlaveController, implementa a recepção de decisões através de um UDPClient e um interpretador CommEngInterpreter e aplicação das mesmas sobre um controlador de motores MD25.

- (***package controller.interpreter***) contém classes que interpretam uma dada percepção e a transformam numa dada decisão de acção.

Interpreter – encapsula o comportamento básico de um interpretador que processa percepções e devolve acções. Possibilita também o retorno de uma acção sem percepção, como sendo o seguimento da acção anterior.

CommEngInterpreter – subclasse de Interpreter, processa as mensagens recebidas pelo cliente, devolvendo as instruções para os motores que estas representam. Nesta classe não é necessário o retorno de uma acção sem percepção.

WCamMD25Interpreter - subclasse de Interpreter, processa os mapas de centróides recebidos da câmara web, devolvendo as instruções para os motores consoante a posição e área destes. Nesta classe é necessário o retorno de uma instrução para os motores, de forma a manter a velocidade e direcção anteriores enquanto não se receber um novo mapa de centróides.

WKeyMD25Interpreter - subclasse de Interpreter, processa teclas recebidas do teclado, devolvendo as instruções para os motores que estas representam. Nesta classe é necessário o retorno de uma instrução para os motores, de forma a alinhar a sua direcção, para que seja possível obter um melhor controlo sobre o grau de viragem, e manter a velocidade para que os motores não parem caso não seja recebida uma nova tecla.

WKeyWKeyInterpreter - subclasse de Interpreter, processa teclas recebidas do teclado, devolvendo instruções correspondentes para o próprio teclado. Nesta classe não é necessário o retorno de uma acção sem percepção.

7. Pacote Principal (*main*)

Módulo que contém os programas principais (a ser executados pelo utilizador), onde são instanciados e corridos os `Controller` correspondentes. A arquitectura inclúi três programas principais:

WebCameraMasterMain – programa "mestre" que se guia a si próprio e os seus "escravos" se existirem, a partir de um câmara web, usando um controlador `WCamMD25MasterController`.

KeyboardMasterMain – programa "mestre" que utiliza as entradas de um teclado para se guiar a si próprio e os seus "escravos", usando um controlador `WKeyMD25MasterController`.

SlaveMain – programa "escravo" que recebe quaisquer ordens de um ou mais "mestres" e age conforme, usando o controlador **`MD25SlaveController`**.

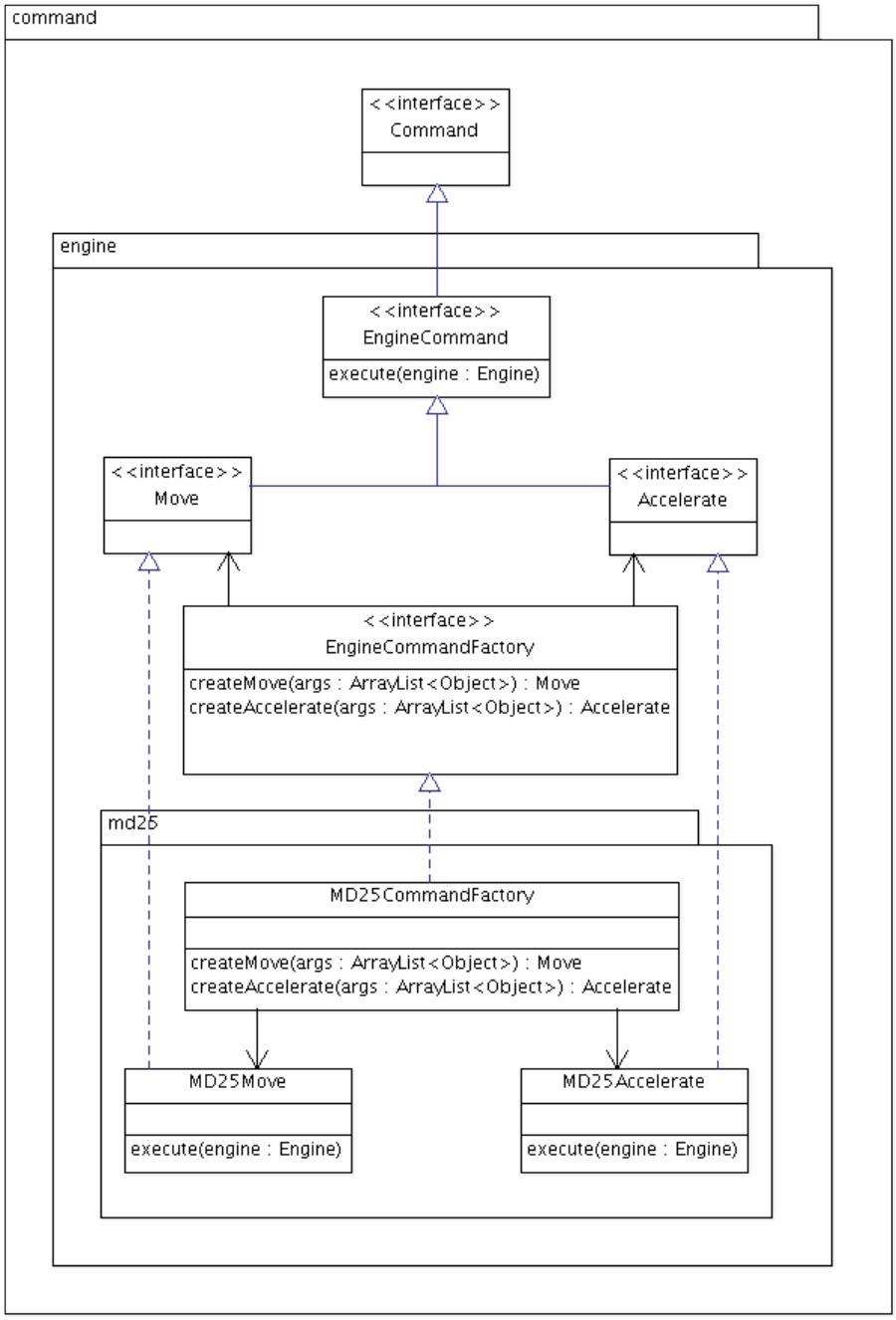
II. Exemplos de Extensão da Arquitectura

A Arquitectura encontra-se desenhada de forma a permitir boa flexibilidade e extensibilidade. Deixa-se nesta secção alguns exemplos de extensões que podem ser facilmente realizadas.

1. Extensão de comandos

O módulo de comandos da Arquitectura pode facilmente ser estendido com novos comandos. Suponha-se que se pretendia adicionar o comando `Accelerate` à panóplia de comandos que um motor genérico suporta e concretamente adicionar o comando `MD25Accelerate` aos comandos de um motor MD25, associando ao comando de baixo nível *SET ACCELERATION* do controlador MD25.

O esquema seguinte consta num diagrama de classes, que melhor exprime esta extensão.



No diagrama encontra-se também o comando Move que apresenta uma estrutura idêntica.

- Na *package* command.engine cria-se uma nova interface Accelerate (vazia) que estende de EngineCommand, para que um motor genérico passe a suportar o novo comando. Adiciona-se a declaração do método createAccelerate à fábrica de comandos de um motor genérico, para possibilitar a criação de uma instância de qualquer comando concreto que implemente Accelerate.

- Na *package* command.engine.md25 cria-se uma classe concreta MD25Accelerate que implementa o comando genérico Accelerate, como por exemplo:

```
package command.engine.md25;

public class MD25Accelerate implements Accelerate {

   public byte value;

   public MD25Accelerate(byte value) {
      super();
      this.value = value;
   }

   @Override
   public void execute(Engine engine) {

      MD25EngineInstruction instr = MD25EngineInstruction.SETACCELERATION;
      engine.send(MD25EngineTranslator.makeCmd(instr,value));

   }

}
```

- Neste exemplo o comando consta em induzir o motor a enviar os bytes correspondentes ao comando *SETACCELERATION* com o valor value.

- O passo seguinte é implementar o método createAccelerate na fábrica concreta de comandos de um motor MD25 (MD25CommandFactory). O método seria algo como:

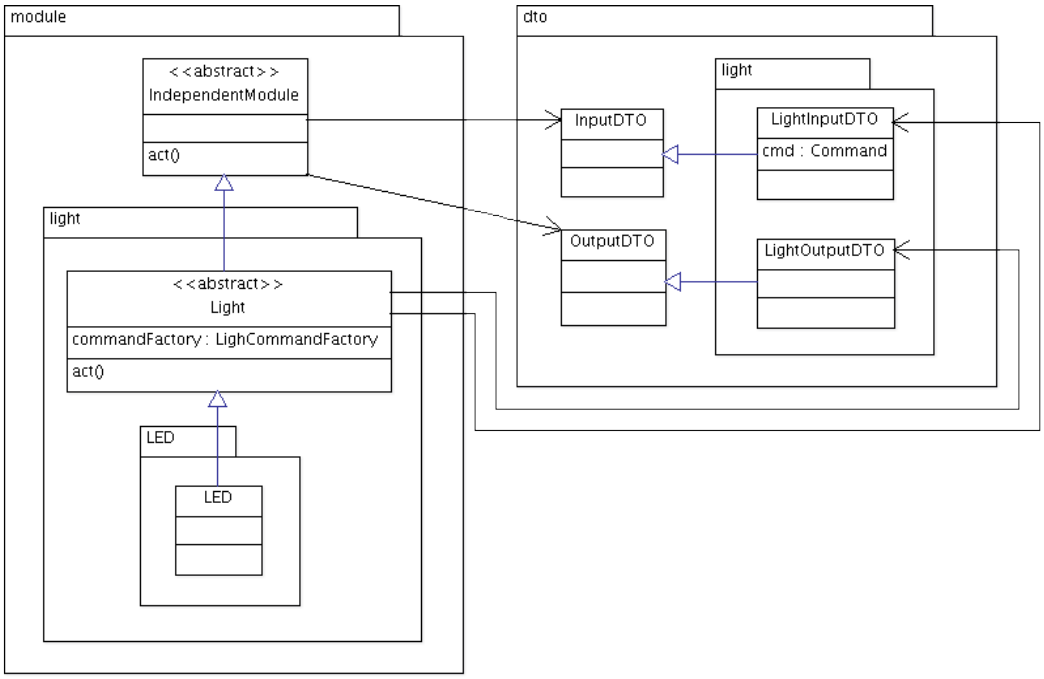
```
@Override
public Accelerate createAccelerate(ArrayList<Object> args) {
   return new MD25Accelerate((Byte)args.get(0));
}
```

- O último passo é adicionar à enumeração dos comandos virtuais de um motor (VirtualEngineCommand) o novo comando virtual ACCELERATE para que os controladores tenham acesso simplificado à construção do comando.

(para uma melhor legibilidade do código aqui exemplificado, estão omitidas as excepções lançadas e apanhadas)

2. Extensão de módulos

Para estender a arquitectura de forma a ter, por exemplo, mais um actuador, será necessário acrescentar um módulo independente (`IndependentModule`). Suponhamos que queremos adicionar um LED aos actuadores da arquitectura, cuja actuação é ligar ou desligar. Consideremos o seguinte esquema de extensão:



- Neste exemplo adiciona-se um módulo `command.light` e cria-se o módulo independente `Light` que abstrai qualquer luz. Esta classe implementa a função `act()` que pega no comando que o novo `LighInputDTO` transporta e executa-o . Será necessário também adicionar os novos comandos de luz ao módulo de comandos e criar uma fábrica que facilite a sua criação. A classe `LED` implementará a função específica de envio da informação necessária para ligar/desligar a luz.

- Para utilizar este módulo correctamente o controlador que o tem deverá ter também pelo menos um interpretador de percepções que as analise de forma a devolver instruções coerentes para este módulo. Um exemplo seria a detecção de obstáculos através da visão despoletar o acebder de uma luz de aviso.

III. Problemas e Trabalho futuro

Uma das melhorias que tencionamos fazer no futuro será construir um mecanismo que permita a realização de *queries* aos módulos independentes, como os motores, sem bloquear o seu funcionamento. Este mecanismo deve poder permitir que as *queries* devolvam dados de tipos diferentes.

O mesmo tipo de mecanismo deve ser aplicado ao módulo de comunicação do cliente pois este bloqueia enquanto não recebe uma nova mensagem e portanto não corre a sua função `act()` enquanto isto não se der. De momento isto não é um problema pois a função `act()` não tem propósito mas no futuro esta função, em módulos que são primariamente perceptores como o cliente, deverá servir para permitir ao controlador alterar as suas configurações.

Ainda falando em comunicações, outro aspecto que pode ser melhorado neste âmbito é que a comunicação em *broadcast* no caso de utilização de "follow the leader" poderia ser implementada através de *multicast* verdadeiro e não envio múltiplo sequencial de pacotes. No entanto a vantagem ganha em tempo através deste mecanismo só seria substancial no caso de haver muitos robôs envolvidos no esquema. No âmbito de projectos deste tipo esta situação não se dá, pelo que o esforço de programação ultrapassa esta vantagem largamente.

Considerando que no futuro deverão ser criados mais módulos específicos para lidar com novas situações, envolvendo não só os periféricos usados neste trabalho mas outros também , deverão ser introduzidas mais configurações neles e a também a capacidade de as alterar através da sua função `act()`, permitindo a sua alteração dinâmica. A utilidade disto é considerável pois permitirá a auto-configuração dos módulos do programa para lidar com alterações do ambiente.

Um exemplo disto que já está parcialmente implementado é no módulo da visão a função `act()` permitir a alteração dos parâmetros de análise de centróides da imagem. Para completar este exemplo seria necessário que o controlador analisasse a imagem e lista de centróides e desta análise derivasse novos valores para esses parâmetros. Isto seria muito útil para compensar alterações na luminosidade e cores do ambiente.

Outro caso em que isto seria muito útil é na compensação da discrepância de velocidades entre os robôs envolvidos no cenário de "follow the leader", recorrendo a algum tipo de feedback posicional, permitindo um seguimento mais preciso.

Outra tarefa relacionada com a configuração que tencionamos realizar no futuro é a separação das opções de configuração globais pelas classes interessadas. A razão para fazer isto desta forma é que, apesar da forma actual permitir um acesso simplificado e alteração simples por parte do programador, pode dar azo a erros pois permite que sejam alteradas a qualquer altura no decorrer do programa, o que não é recomendado para algumas e ineficaz para outras.

Um aspecto da arquitectura que poderia ser melhorado é o processo de decisão e a conversão das percepções em acções. De momento, os interpretadores fazem a ponte directamente entre o formato específico do *output* dos módulos perceptores com o formato específico do *input* dos módulos actuadores enquanto que os controladores controlam apenas a lógica de escolha e envio das acções. Apesar desta arquitectura ser prática para a natureza do trabalho, o facto dos interpretadores lidarem com as especificidades de ambos os formatos e tomarem conta de parte do processo de decisão torna-os classes vitais e muito sensíveis a alterações no projecto, além de dificultar bastante a comparação informada entre decisões por parte dos controladores, diminuindo a flexibilidade do processo de decisão.

Uma sugestão para melhorar a arquitectura neste sentido, seria a introdução de um formato genérico intermédio, usado no controlador para gerar e comparar decisões de acção. Nesta nova arquitectura, os interpretadores limitar-se-iam a traduzir as informações específicas dos perceptores para o formato genérico e um novo tipo de classes seria criado para lidar com a tradução das decisões de acção no formato

genérico para o formato específico de cada actuador.

Esta sugestão, no entanto, também tem os seus problemas, como por exemplo, introduzir um *overhead* maior no processo de decisão, atrasando o programa, complicar a lógica dos controladores, obrigando a uma estrutura de herança e separação de comportamento mais complexa neles e implicar um esforço considerável de programação para a sua implementação, extensão e manutenção. Devido a estes factores, consideramos que apenas para projectos em contextos mais ambiciosos é que esta alteração poderá ser vantajosa.

Finalmente, um aspecto menos relevante que poderia ser melhorado é a qualidade e quantidade do *feedback* disponibilizado ao utilizador pela arquitectura. Um exemplo seria imprimir as velocidades e direcções dos robôs a uma frequência predefinida. Isto seria especialmente útil em casos como o de controlo por teclado em que o utilizador controla directamente estas variáveis.

IV. MobotArch no Eclipse

Sugere-se a utilização do Eclipse IDE for Java Developers para a edição do código fonte da arquitectura. O arquivo comprimido *MobotArch.tar.gz* inclúi na pasta *source/eclipse_project* um projecto do Eclipse com o código fonte do MobotArch. Esse arquivo contém também as pastas *library_jars* e *native_libraries*, necessárias para realizar as instruções em baixo.

1. Importar MobotArch no Eclipse

- Seleccionar `File` → `Import...` → `General` → `Existing Projects into Workspace` → `Browse...`

- Procurar a pasta *source/eclipse_project/MobotArch* e confirmar.

- Seleccionar `Copy projects into workspace` se pretender que as modificações sejam feitas numa cópia do projecto e não no projecto original.

2. Associar Bibliotecas

O MobotArch utiliza bibliotecas externas de processamento de imagem e de comunicação série, pelo que é necessário associá-las ao projecto:

Com clique da tecla direita do rato no projecto aceder a `Properties` e :

- Seleccionar o separador `Java Build Path` → `Libraries` → `Add Library...`

- Seleccionar `User Library` e premir `OK` → `User Libraries...` → `New...`

- Inserir um nome e premir `OK` → `Add JARs...`

- Seleccionar todos os ficheiros presentes na pasta *library_jars* e confirmar a criação da biblioteca.

- De volta ao menu `Java Build Path`, expandir a biblioteca criada e seleccionar `Native library location` → `Edit...` → `External Folder...` → procurar a pasta *native_libraries* e confirmar.

3. Exportar ficheiros JAR executáveis

O Eclipse permite exportar ficheiros JAR executáveis, evitando o processo manual de compilação.

- Seleccionar `File` → `Export...` → `Java` → `Runnable JAR file`

- Em `Launch configuration` seleccionar o programa com a função `main` a partir do qual se pretende criar o JAR executável.

- Escolher a directoria destino onde se pretende guardar o JAR.

- Em `Library handling` seleccionar a segunda opção e premir `Finish`.