Faculdade de Engenharia da Universidade do Porto

Informática Industrial

Mestrado Integrado em Engenharia Electrotécnica e de Computadores

2019/2020

Automação de Linha de Produção

Junho de 2020

Diogo Alexandre Brandão da Silva

Fábio André da Rocha Morais

João Marcelo Casanova Almeida Tomé Santos

Marco António Mendonça Rocha

Tempo para concurso: 8:01

Introdução

O relatório aqui apresentado explana o desenvolvimento do projecto final da UC Informática Industrial que consiste na automatização do simulador da linha de produção flexível e no desenvolvimento de uma plataforma de controlo e monitorização do mesmo (MES - Manufacturing Execution System).

Relativamente ao controlo do nível mais baixo (controlo do simulador), este foi realizado recorrendo à tecnologia *Soft PLC* do Codesys, que dispõe de um ambiente de desenvolvimento para programas IEC 61131, e de um ambiente de execução dos programas.

No nível imediatamente acima(MES), o controlo/programação foi realizado recorrendo a uma linguagem orientada a objetos, tendo sido usado Java 1.8. Associado a este nível encontra-se o uso de uma base de dados remota em que se usou o sistema PostgreSQL para gestão num servidor da feup que é alojado *(http://db.fe.up.pt)*. Este nível recebe ordens vindas do *software* de gestão de recursos: ERP- Enterprise Resource Planning através de ficheiros XML a partir do protocolo UDP/IP.

Relativamente às comunicações: a comunicação entre o MES e o *Soft PLC* é feita usando o protocolo OPC-UA (através da biblioteca Milo), e a comunicação entre o *Soft PLC* e o simulador é feita por Modbus TCP.

É de realçar o uso do git para a gestão de versões, assim como a distribuição de tarefas.

Para além disso, foi usado o *Junit* para fazer testes automáticos ao sistema e validar as suas funções previstas.

Estrutura do Código

Primeiramente, é de realçar que ao longo do processo de desenvolvimento de todo este projeto foi seguido o princípio S.O.L.I.D. em que este tem os 5 princípios:

1. S - Single Responsibility Principle
2. O - Open-Closed Principle
3. L - Liskov Substitution Principle
4. I - Interface segregation Principle
5. D - Dependency Inversion Principle

Estes princípios ajudam a desenvolver um código mais limpo, separando as responsabilidades, facilitando *code refactoring* e favorecendo o reaproveitamento do código. Com isto tornamos o software mais robusto, escalável e flexível.

O MES neste sistema toma um papel de gestor que faz o escalonamento das ordens, determina as melhores ordens a ser executadas, determina as peças a serem enviadas e o caminho destas, recolhe também e guarda os dados do PLC na base de dados de forma a depois se obter as estatísticas e a persistência pretendida.

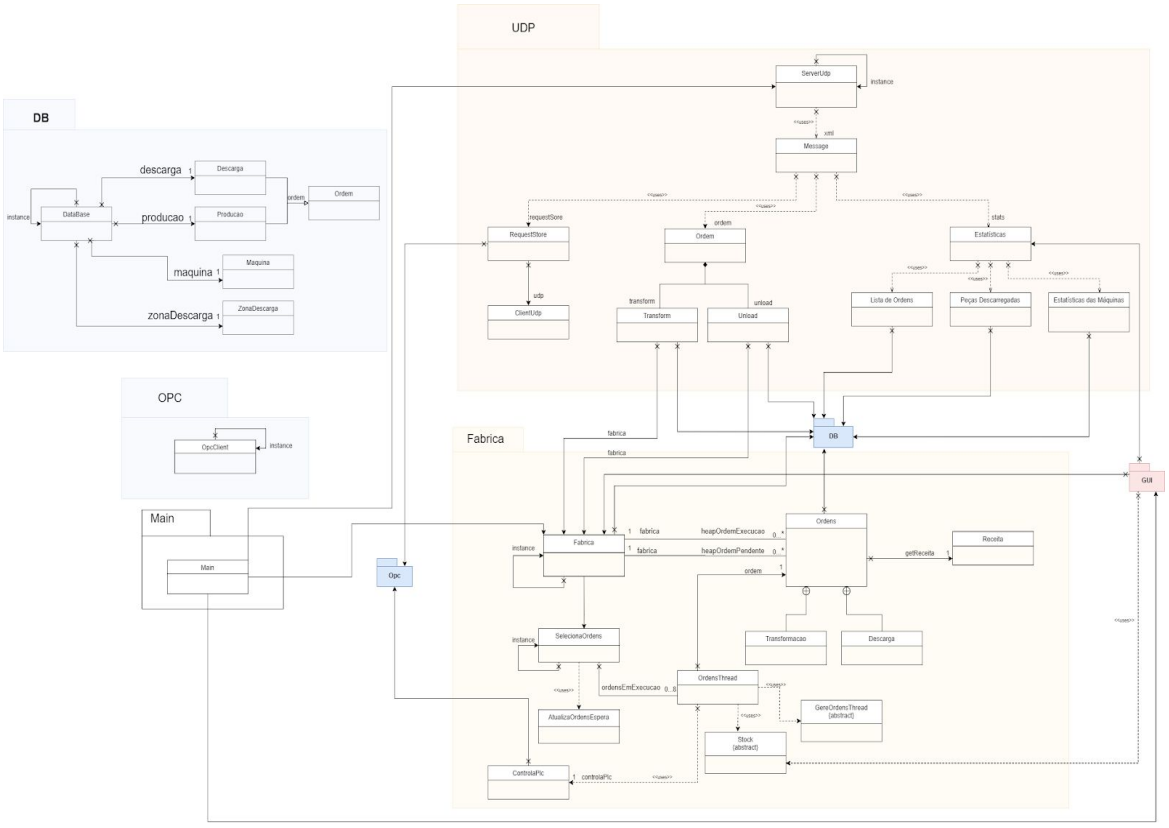

Fig. 1 - Diagrama de classes resumido do MES

Esta figura corresponde ao diagrama de classes de forma resumida. Para perceber melhor a ligação entre classes, pode ser encontrado nos ficheiros em anexo o diagrama de classes completo (UML/ClassDiagramMesComplete.png)

O sistema MES está dividido em 6 *packages*, como representado no diagrama da **Fig. 1,** em que temos um *package* referente a todas as classes pertencentes ao UDP, OPC, DB, Main, *Fabrica e Gui*.

A vantagem do uso de *packages* é conseguirmos obter um código organizado e obter métodos que só comunicam entre packages, desta forma não é necessário colocar o método ou variável como *public*.

A Main tem como objetivo instanciar a interface do utilizador, a classe *Fabrica* e o servidor UDP.

Os componentes principais deste software são a base de dados (*DataBase*), o escalonamento das ordens (*SelecionaOrdens*), a execução de ordens (*OrdensThread*), a determinação do caminho para as peças e o envio das informações para o PLC (*ControlaPlc*), a comunicação OPC (*OpcClient*), o servidor UDP (*serverUdp*), o processamento das mensagens que recebe via UDP (*Message*) e a interface com o utilizador (*GUI*).

Quando é recebida uma ordem pelo servidor UDP, esta é processada na classe *Message* que tem como função distribuir pelas classes *RequestStore*, *Ordem* e *Estatistica*, dependendo do tipo de comandos que receba no xml. Realça-se o facto de se conseguir pedir para exportar as

estáticas com o comando <Request_Stats> via UDP: por esse motivo temos as classes de estatística no package UDP.

A classe *Ordens* pode dar origem a múltiplos objetos, pois depende da quantidade de ordens recebidas via UDP vindas do ERP. Esta classe coloca os respetivos dados referentes a esta classe na base de dados, como por exemplo, quando uma ordem é executada, quando entra uma peça em produção...etc.

Em anexo está o diagrama de objetos que explica bem a situação acima descrita, onde está explicado para uma situação em concreto de 2 ordens de transformação em execução, 1 ordem de descarga em execução e 1 ordem de transformação pendente (UML/ObjectDiagram.png).

A classe *Fabrica* que funciona como um *singleton* tem a responsabilidade de mais alto nível no sistema. Esta classe possui a lista de ordens em execução e a lista de ordens pendentes, fazendo a sincronização quando o sistema é inicializado, isto é, coloca as ordens que estavam a ser executadas e as ordens pendentes nas respetivas *heaps*. Para além disso, instancia um objeto da classe *SelecionaOrdens* que funciona numa outra *Thread*, estando esta classe responsável pelo escalonamento das ordens, pela seleção das ordens que vão ser executadas, e pela gestão das ordens que precisam de ser trocadas. Esta classe conhece a classe *Fabrica* para que consiga aceder à heap de ordens pendentes.

As ordens são colocadas numa heap de ordens pendentes: desta forma o primeiro nó é sempre o elemento com mais prioridade. Ao utilizar uma heap para ordenar os dados conseguimos obter uma complexidade ao retirar a ordem de *O(1)* e ao ordenar como *O(log n)*.

A lista de possíveis transformações é feita através do Algoritmo Dijkstra, que determina o menor “caminho”, neste caso determina a transformação mais rápida a ser executada. Ver ficheiros em anexo (outro/receita.pdf).

A classe *GereOrdensThread*, que é uma classe abstrata, trata da partilha de dados dos objetos da classe *OrdensThread*, tem o estado das máquinas, os tempos das máquinas, o número de objetos instanciados de *OrdensThread*, semáforos, *flags*...etc. Como o nome indica esta tem a função de gerir as várias *Threads* que estão a ser executadas por ordem.

É de realçar o facto de termos limitado o número de Threads a serem criadas, pois iria afetar o desempenho do sistema se tivéssemos um número demasiado elevado de Threads em simultâneo. No entanto, limitar o número de Threads não será um problema em termos de eficácia, porque ter 8 ordens a executar em simultâneo (número de Threads superior a 8) não constitui uma vantagem visto que apenas existem 3 tipos de máquinas e 3 tipos de descarga.

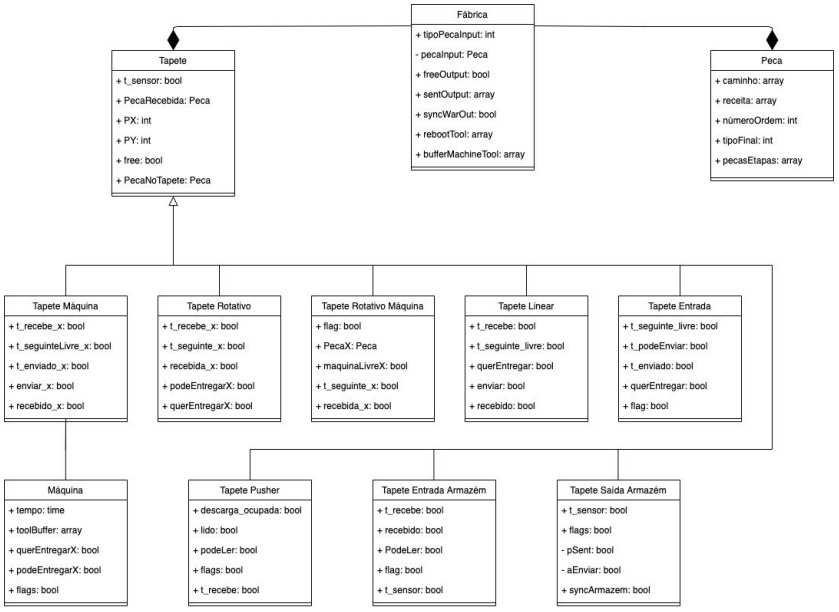

Fig. 2 - Diagrama de classes do PLC

O PLC é responsável por controlar diretamente o chão de fábrica. Este foi desenvolvido de modo a que os dados recebidos nas peças fossem sempre respeitados sem ocorrer colisões entre peças nem *deadlocks*, independentemente da falha dos outros sistemas. Além disto o PLC também é responsável por obter e enviar as estatísticas.

Como se pode ver na figura 2 a fábrica é constituída por duas classes principais: a Peça onde estão definidas todas as propriedades necessárias para que cada peça consiga percorrer a fábrica como foi planeado no MES; e o Tapete que consiste numa abstração que generaliza cada tapete individual que vai fazer um certo processo na peça: movimentação, transformação, carga ou descarga no armazém e entrada ou saída da fábrica.

Funcionamento do Código

Existe um servidor UDP com ciclo infinito numa thread à espera de mensagens via UDP. Caso este receba, é processada a mensagem e verifica se é uma ordem: caso seja então vai para uma classe que trata de ordens originadas via UDP. Nesta classe separa-se todos os elementos necessários e cria-se um objeto da classe *Transform* ou *Unload*, dependendo do caso, criando uma ordem do tipo *Ordens* que é colocada automaticamente numa *heap* de ordens pendentes e na base de dados. Consequentemente, existe uma classe que executa em *Thread* num ciclo infinito e que atualiza esta *heap* a cada segundo.

Em seguida, encontra-se o diagrama que explicita a sequência explicada, desde a recepção via UDP até à inserção na DB:

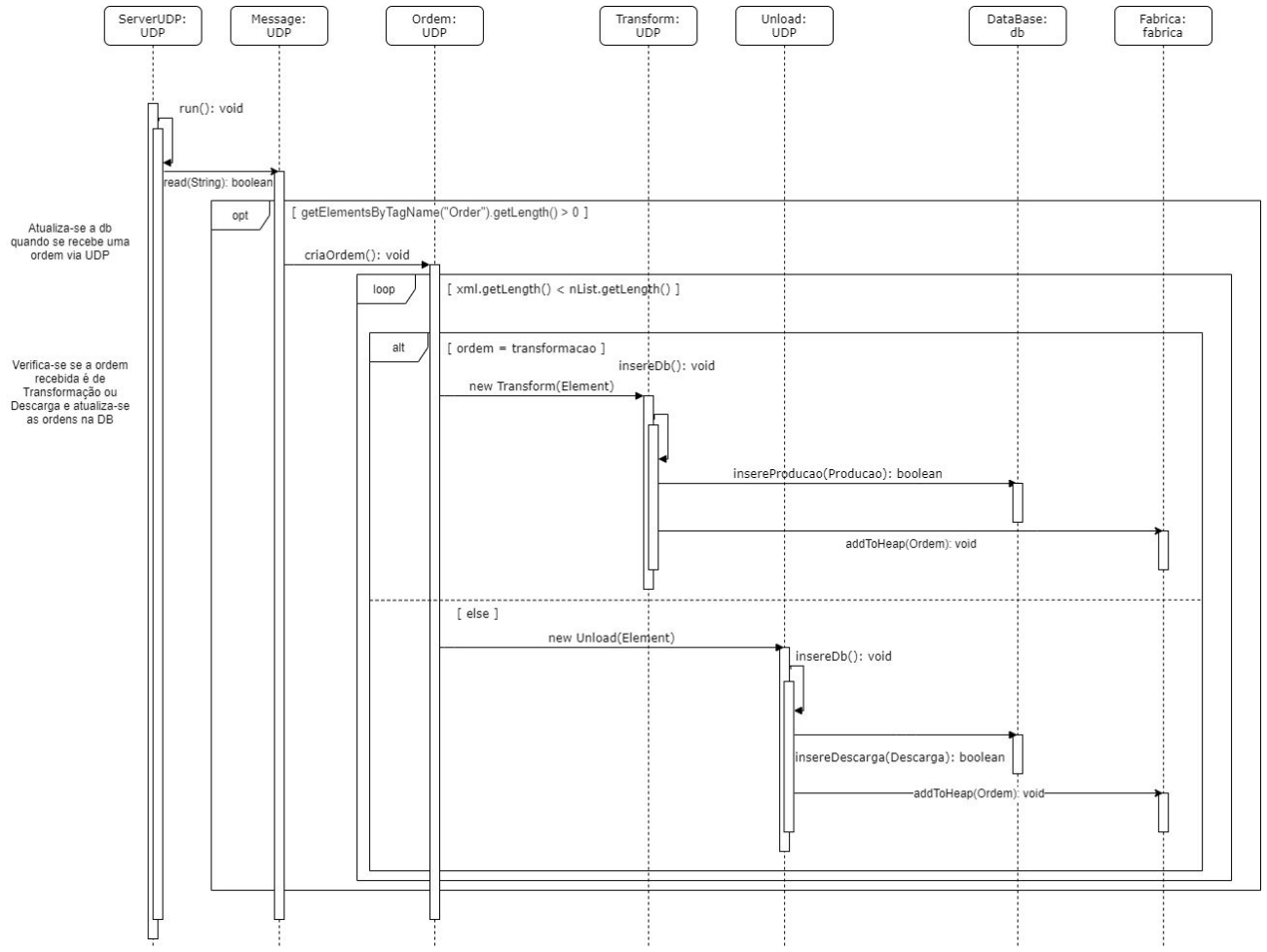

Fig. 3 - Diagrama de sequência do processo de recepção de ordens via UDP

Por outro lado existe a classe *SelecionaOrdens* que também funciona em *Thread* e tem como principal função fazer o escalonamento das ordens: o algoritmo determina se determinada ordem deve ou não ser executada.

O algoritmo para esta classe é um pouco complexo, pois depende do atraso máximo da ordem em questão, da ocupação das máquinas, do tipo de ordem e do tipo de transformação. Caso uma ordem esteja a utilizar os 3 tipos de máquinas (A,B,C) e uma ordem queira entrar para a máquina B, então esta entra e são executadas em paralelo estas 2 ordens. É também possível entrar uma ordem com mais prioridade que a ordem que está em execução, o que faz com que a ordem em execução fique em pausa e a nova ordem entre em funcionamento.

Caso a classe SelecionaOrdens coloque uma ordem em execução, é criado um objeto da classe OrdensThread que abre uma *Thread* para essa ordem e é adicionado a um array de ordens em execução. Nestes objetos, quando uma das ordens quer enviar uma peça ativa o semáforo, inativando a execução de envio de outras peças de outras ordens.

Nesta classe, para determinar se uma peça deve ou não ser enviada é analisado o estado da máquina que a peça quer ir (livre ou ocupado): o estado livre fica a *true* X segundos antes de acabar a tarefa dessa peça, este tempo foi medido para quando chega uma peça à máquina esta

entra imediatamente sem esperar, desta forma é evitado o congestionamento das linhas de produção.

Quando 1 peça da ordem X é enviada, esta passa por um algoritmo que determina qual a receita a fazer através do uso de grafos escolhendo a sequência de operações mais rápida, determina também a melhor máquina que deverá ir, obtém o melhor caminho via o algoritmo *dijkstra*, e todas as informações pertinentes são enviadas via OPC-UA.

Caso haja uma ordem que utilize 3 ferramentas diferentes na mesma máquina e que esteja como pendente, o sistema lança uma flag fazendo com que as últimas peças pendentes da ordem em execução deixem a primeira coluna livre, para entrar logo esta ordem.

Em seguida, encontra-se o diagrama que explicita a sequência de ordens explanadas:

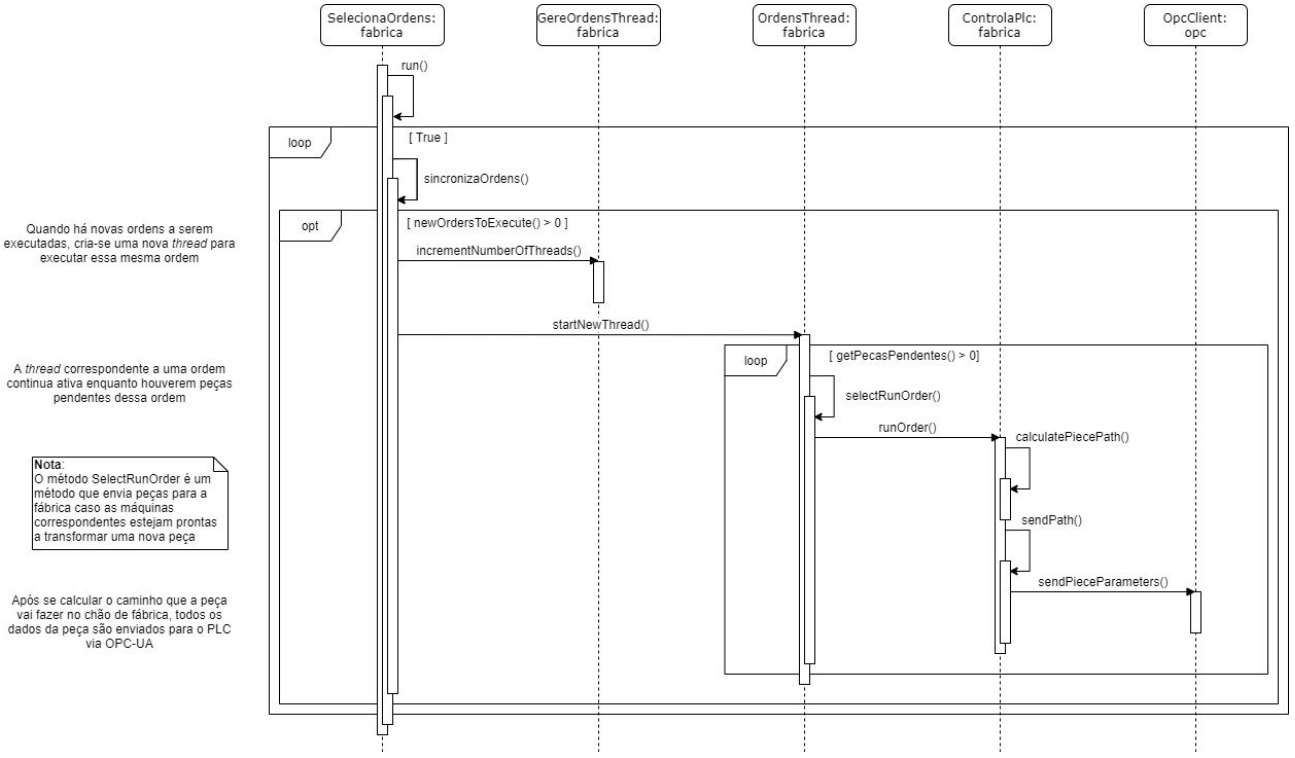

Fig. 4 - Diagrama de sequência do processo de selecção de ordens.

Relativamente ao PLC, as peças entram na fábrica por dois sítios diferentes: pelo armazém e pelos dois tapetes dos cantos direitos. Quando uma peça entra na fábrica, é criada uma instância de peça que contém a informação recebida pelo MES, ou no caso de ser uma peça que vai para o stock é o PLC que preenche os atributos da peça.

Sempre que há circulação da peça de um tapete para outro, as instâncias são passadas com a peça. O caminho realizado é o que se encontra nas variáveis PathX e PathY, sendo que cada tapete tem normalmente a sua posição (X, Y), facilmente se consegue comparar e decidir para onde é que a peça se destina.

As máquinas têm um buffer circular com as ferramentas que vão ser utilizadas quando as peças lá chegarem. Desta forma, quando o MES envia as informações da peça também atualiza

este buffer. A informação das operações que a peça tem de realizar também estão na classe peça, uma vez que quando a peça chega à máquina está à espera que esteja colocada a ferramenta correta, no entanto caso tenha acontecido algum erro a ferramenta muda novamente para a correta antes de começar a operação.

Outra vantagem de ser conhecido o caminho que a peça vai fazer enquanto esta percorre a fábrica é a facilidade em evitar os deadlocks. Isto é feito principalmente nos *Tapetes Rotativos Máquina*. Assim, os tapetes rotativos que dão acesso às máquinas verificam se as próximas localizações onde as peças querem ir estão livres e caso esteja permite que prossigam, caso contrário estas têm que esperar até que esteja livre. Isto tem um comportamento adjacente que é a formação de filas de espera caso os tapetes de baixo estejam ocupados. Este problema temporal foi parcialmente resolvido pelo algoritmo que decide a ordem dos pedidos a realizar (no MES).

Por último, relativamente à Base de Dados, a sua atualização é feita quando ocorre uma das seguintes situações: descarga de peça, ou carga de peça para o armazém, ou término de uma operação na máquina ou saída de uma peça do armazém para sofrer transformação.

Relativamente às 3 primeiras situações mencionadas, a classe *OpcClient* está responsável por atualizar a Base de Dados quando uma das variáveis correspondentes muda de estado (sendo monitorizadas por Subscrição usando OPC-UA).

O último caso de atualização da Base de Dados ocorre quando a classe *OrdensThread* manda sair do armazém uma nova peça de uma dada Ordem para ser transformada. Neste caso, a classe atualiza a ordem o número de peças pendentes e o número de peças em produção.

Em seguida, encontra-se o diagrama que explicita o processo de atualização da DB:

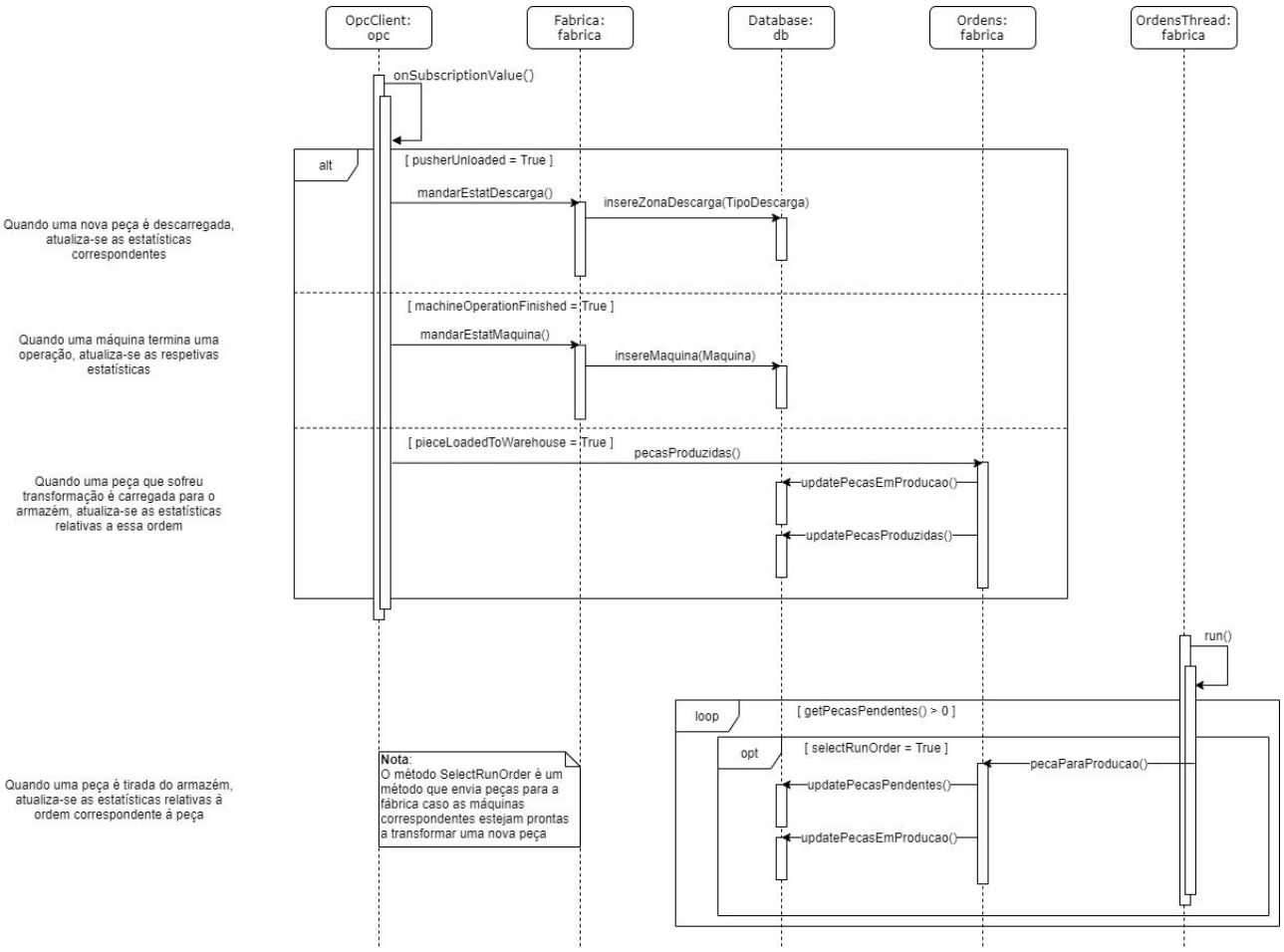

Fig. 5 - Diagrama de sequência do processo de atualização da DB

Implementação

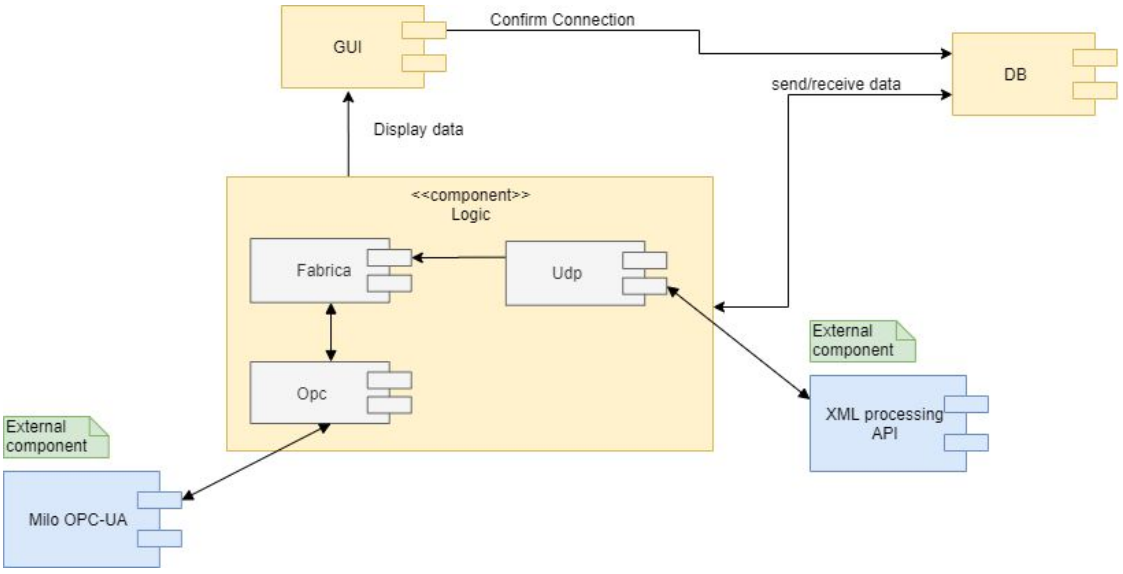

Fig. 6 - Component Diagram

Para a implementação do MES foi usado Java, pois é uma linguagem de programação orientada a objetos com diversas bibliotecas que permitem realizar este projeto de maneira sólida.

Como indicado na figura 6, foram implementadas 2 bibliotecas externas que ajudaram à realização deste projeto, sendo uma delas o Milo OPC-UA, que é uma biblioteca *open source* que implementa o OPC-UA (IEC 62541), para esta conexão é necessário indicar a porta e endereço IP onde queremos comunicar.

Foram implementados vários métodos para ajudar a manipular esta classe, métodos para dar *connect*, ler valores do Codesys ou enviar valores para o Codesys.

Para maximizar a eficiência do sistema foi utilizado também o modo de subscrição do OPC-UA, desta forma apenas são recebidos os valores que mudam o seu valor.

Para o uso do OPC-UA houve a necessidade de selecionar as variáveis que queríamos fazer ligação, pelo *Symbol Configuration* do codesys; do lado do MES é necessário escolher o *NodeId* que queremos ler/escrever.

Por ser uma classe de comunicação, e para dar alguma segurança, foi usada exclusão mútua para evitar que 2 processos ou *threads* tenham acesso em simultâneo a este recurso, foi usado o *synchronized* que evita este problema.

Uma outra técnica usada foi o *singleton*, que é um *design pattern* que permite que uma classe apenas tenha um objeto.

O uso da base de dados foi conseguida com a tecnologia *JDBC*, um API standard do java que reúne conjuntos de classes e interfaces que possibilita a conexão através de um driver especifico da base de dados desejada. Com este driver pode-se executar instruções SQL de qualquer tipo de banco de dados relacional.

O uso do XML foi conseguido através da técnica DOM (*Document Object Model*), onde os elementos são organizados numa árvore. Facilmente se endereça e manipula os objetos através de métodos.

A interface com o utilizador apenas mostra os dados que estão no componente *Logic* presente na figura anterior, desta forma conseguimos separar o controlador, comunicações e a interface do utilizador.

Foi usado o *Swing* para a interface do utilizador, escolhemos esta tecnologia por incluir muitos recursos úteis para fácil visualização e por ser leve e independente da plataforma.

Dada a complexidade do sistema seria impossível correr o sistema apenas numa única *Thread*, uma vez que tanto as comunicações como a *interface* gráfica têm exigências temporais necessárias para um bom funcionamento. Desta forma utilizamos várias Threads, como por exemplo para a *interface* do utilizador, para o servidor UDP, para execução de ordens...etc. Estas threads têm um delay variável de 1ms a 100ms dependendo da sua função, serve principalmente para não sobrecarregar o CPU do computador desnecessariamente.

Quando é executada uma nova ordem, esta abre uma nova *Thread* que fica responsável por enviar peças via OPC-UA.

Desta forma, tivemos o cuidado de utilizar semáforos para resolver a exclusão mútua falada anteriormente.

Relativamente à implementação do PLC, esta foi feita recorrendo ao software Codesys usando linguagens da norma IEC-61131. No nosso caso recorremos a: ST, SFC e FBD.

Para que toda esta implementação fosse possível começou-se por estabelecer a comunicação entre os diferentes elementos do sistema (MES, PLC e Simulador SFS). Inicialmente tratou-se da comunicação entre o PLC e o Simulador SFS: sendo esta feita recorrendo ao protocolo ModbusTCP, foi necessário especificar os endereços de todas os sinais de entrada e saída do Simulador, tendo ficado o Simulador como escravo que atualiza as suas variáveis ciclicamente com um ciclo de 100ms. Relativamente à comunicação entre o MES e o PLC, o PLC serve a função de servidor OPC-UA: desta forma disponibiliza as variáveis selecionadas no *Symbol Configuration*.

Como o Codesys disponibiliza todas as linguagens da norma IEC-61131, foi possível rentabilizar-mos a programação do PLC. Isto é, para cada elemento (tapete rotativo, tapete linear, máquina, ...) apenas se implementou um único *Function Block* que facilmente se replicava sempre que se pretendia instanciar um novo elemento. O que também nos traz vantagens em fase de *debug*: ao fazermos uma correção num único *Function Block*, todas as instâncias iguais a esse era corrigidas.

Por último, o MES nunca invoca métodos do PLC, apenas se faz uma sincronização entre eles via variáveis transmitidas por OPC-UA, o que permitia que não ocorressem conflitos entre os diferentes níveis de controlo.

Estrutura final vs. Inicial

Como em qualquer projeto a estrutura final foi diferente da estrutura inicial prevista. No entanto é consideravelmente importante o plano e desenho inicial do sistema, pois permite que a solução final esteja baseada na opinião conjunta de todos os elementos, ajudando também na divisão de tarefas.

No MES, a estrutura inicial relativa à base de dados, opc e ao udp manteve-se como prevista, porém no package *Fabrica* a sua estrutura mudou, pois foi necessário mais classes como por exemplo a classe *GereOrdensThread* ou a classe *Stock*.

No PLC, a principal diferença em relação ao planeado foi a introdução da classe tapete rotativo máquina. Esta classe surgiu para resolver os deadlocks, sendo que os deadlocks não ocorrem nos restantes tapetes rotativos, mas apenas nos que dão acesso às máquinas.

Outra diferença foi a mudança dos parâmetros das classes, sendo que estes parâmetros surgiram principalmente na otimização da operação da fábrica em tarefas como: a mudança da ferramenta antes da peça chegar à máquina, a persistência da fábrica e das estatísticas caso o MES se desligue, entre outras.

Começamos por definir o caminho pelo algoritmo A* onde escolhia o caminho mais curto, mas chegamos a um ponto que não tínhamos muita flexibilidade para mudar algumas coisas de maneira a otimizar mais. Foi então mudado o algoritmo e o melhor caminho não era apenas definido como o caminho mais curto, mas tendo em conta o tempo das máquinas, organização da fábrica...etc

Comparação com Padrões Existentes

Os padrões (*design patterns*) em Engenharia de *Software* representam soluções típicas, genéricas e reutilizáveis em problemas comuns de concepção de arquitetura. Estes padrões não representam um produto final que pode ser transformado em código imediatamente, apenas explicitam a arquitetura usada durante o seu desenvolvimento. Normalmente podem ser apresentados sob a forma de *template* adaptada à situação concreta a resolver.

Um dos padrões mais comumente usados é o ISA95: este define modelos destinados à organização de funções e de trocas de dados dentro de uma empresa. Este padrão tem como principal objetivo reduzir riscos, custos e possíveis erros durante a integração do novo *software* em ambiente fabril. E permite que *software* de diferentes fabricantes comuniquem entre si realizando trocas de dados corretamente.

O ISA95 define a sua hierarquia de controlo em 5 níveis principais:

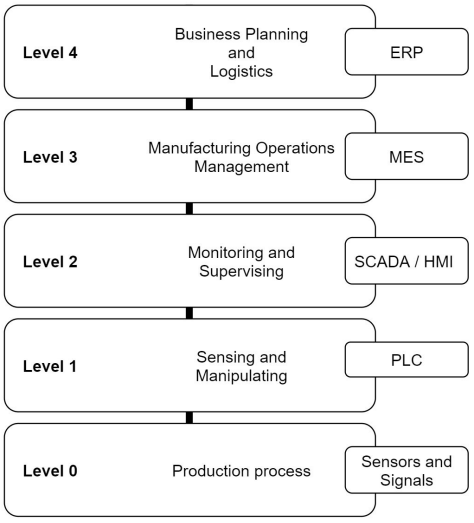

Fig. 7 - Hierarquia de Controlo ISA95

Existe um fluxo de informação trocado entre o *Enterprise* e a camada de baixo (*Manufacturing Control*), esta informação pode ser dividida em 4 classes: *Production Capability Information*, *Product Definition Information*, *Production Schedule* e *Production Performance*.
Fazendo a analogia ao nosso sistema, obtemos as seguintes semelhanças:

- **Production Capability Information** - GereOrdensThread, em que tem todo o estado da fabrica;

- **Product Definition** - *Receita*, classe onde tem o algoritmo que retorna a lista de transformações da peça;
- **Production Schedule** - OrdensThread, esta classe tem como função gerir quando uma ordem deve de enviar uma peça ou nao. Temos a classe SelecionaOrdem, que seleciona a ordem melhor a executar e que cumpre com os requisitos;
- **Production Performance** - *Estatisticas*, é onde obtemos toda a informação relativa ao que foi produzido, quanto e como.

Fazendo uma analogia entre o ISA95 e a arquitetura desenvolvida, este é o padrão que mais se assemelha ao produto final: o Nível 4 do ISA95 corresponde exatamente ao ERP do nosso sistema, o Nível 3 corresponde da mesma forma ao MES, o Nível 2 corresponde à interface do utilizador desenvolvida, o Nível 1 ao *Soft* PLC para controlo do simulador, e o Nível 0 corresponde ao conjunto de sensores e atuadores existentes no Simulador de Chão de Fábrica.

Com o desenvolvimento da Indústria 4.0 e a integração de novas tecnologias com o ambiente industrial surgiu a necessidade de normalizar (*standardisation*) estruturas de comunicação entre essas tecnologias. O que levou à criação do RAMI 4.0: que é um modelo de referência para arquitetura de *software* em ambiente industrial que têm como principal objectivo permitir a comunicação entre diferentes aplicações, tornando possível a criação de um "ecossistema" em que todos os objetos conseguem comunicar entre si.

No modelo RAMI 4.0, além de se ter em conta as camadas e a hierarquia existente entre elas, também se contabiliza o ciclo de vida do produto:

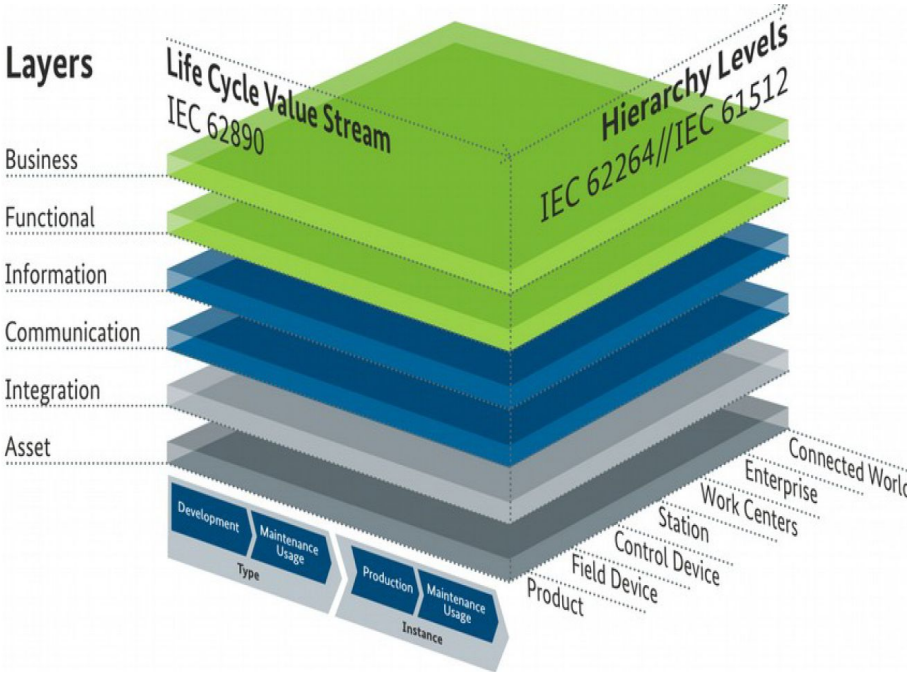

Fig. 8 - Modelo de Referência de Arquitetura RAMI 4.0

Comparando a nossa arquitetura à do RAMI 4.0, podemos associar as diferentes camadas do 3º eixo (referente à Arquitetura) a diferentes partes da nossa implementação. Desta forma, a camada *Business* do ISA95 é corresponde ao ERP do nosso sistema; as camadas *Communication* e *Integration* correspondem ao XML, à *DataBase* e ao *OpcClient* do nosso sistema, visto que estes têm a responsabilidade de aceder aos dados; e os *Assets* correspondem ao Simulador do Chão de Fábrica, dado que este simula os dispositivos reais.

Da mesma forma que a normalização do RAMI 4.0 surgiu de necessidades consequentes da evolução no controlo e automação industrial, outras soluções têm sido também analisadas. Consequentemente, estudos bem-sucedidos têm demonstrado que Abordagens Orientadas a Serviços trazem mais flexibilidade e interoperabilidade ao controlo e automação industrial.

Em Arquiteturas Orientadas a Serviços (SOA - Service-Oriented Architectures), o princípio fundamental baseia-se na implementação das funções de cada aplicação como um serviço.

Nestas arquiteturas, o maior desafio torna-se a definição da granularidade de cada serviço num sistema, sendo a granularidade o tamanho de cada um deles. E considera-se que um serviço tem boa ou má granularidade caso este maximize a modularidade do sistema e minimize a complexidade.

Desta forma, aplicando os conceitos de granularidade de serviços definidos no artigo "*Service Granularity in Industrial Automation and Control Systems*" , podemos classificar o nosso sistema como um híbrido uma vez que utilizamos dois tipos de granularidade: processo e máquina. A granularidade de máquina foi aplicada nos tapetes (PLC), sendo que algumas são mais finas uma vez que contêm mais serviços do que outras (máquinas ou tapetes lineares). A granularidade de processos foi aplicada no controlo (MES), uma vez que este pode ser dividido em vários serviços como *ControlaPLC* ou *Estatisticas*, sendo que estes são grosseiros e, ao contrário da granularidade de máquina, são pouco flexíveis e escaláveis, e o nível de monitorização e diagnóstico é alto (pouco detalhado).

Organização da Equipa

Todo o projeto foi realizado utilizando o modelo SCRUM, construímos o nosso *product backlog* com base nos requisitos do sistema e distribuímos tarefas para cada sprint (*sprint backlog*), todas as semanas reunimos para discutir o que cada pessoa fez e mostrar quais as coisas que poderiam ser melhoradas e barreiras que se obteve ao desenvolver o software.

Foi utilizado a plataforma *Github* para organização da equipa, criar *user stories* e distribuir pelos elementos do grupo.

Consequentemente, considerando em grupo as tarefas realizadas por cada um de nós, chegou-se ao consenso da seguinte distribuição percentual por elemento:

Elementos	Distribuição Percentual (%)
Diogo Alexandre Brandão da Silva	25
Fábio André da Rocha Morais	25
João Marcelo Casanova Almeida Tomé Santos	25
Marco António Mendonça Rocha	25

Testes e Resultados

Após se conseguir obter uma implementação robusta de todo o sistema realizaram-se os devidos testes. Inicialmente testou-se o conjunto de Ordens da Competição A fornecidos pelos professores, onde se obteve um tempo total de 8:01. Este tempo é contabilizado desde o momento em que a primeira peça sai do armazém até ao momento que a última peça completamente processada entra no armazém. Destaca-se o facto de que as Máquinas-Ferramenta descem e sobem a torre, na entrada e saída de cada peça da máquina respetivamente. Caso se optasse por uma implementação em que o movimento da torre não ocorria, obteríamos um tempo igual a 7:40.

Relativamente a questões temporais, também se realizaram testes com ordens de apenas 1 tipo de transformação, sendo possível desta forma avaliar as otimizações da gestão do chão de fábrica.

Teste	Transformação	Quantidade	Tempo Total
1	P1 -> P2	10	1 min 50 s
2	P2 -> P6	15	2 min 29 s
3	P3 -> P7	15	2 min 42 s
4	P1 -> P9	10	2 min 19 s

O sistema também passou com sucesso nos testes em que se carregava continuamente peças para os tapetes de carga enquanto o restante sistema processava as ordens em completa normalidade.

Relativamente à persistência do sistema, quando o sistema MES desligava completamente e voltava a reiniciar: as ordens que estavam a ser anteriormente processadas não são perdidas, isto é, o MES recorre à informação da Base de Dados para recuperar o estado anterior e continuar o processamento normal.

Testou-se também a possível falha de comunicação com a BD: quando esta situação ocorre o MES guarda num ficheiro .txt toda a informação que pretendia atualizar na BD, atualizando-a assim que a ligação se restabelece. Consequentemente, conseguiu-se garantir que, independentemente do estado do sistema, a BD será sempre corretamente atualizada. Ver em anexo (UML/diagramaEstadosDB.png)

Conclusões

Ao longo deste projeto foi nos possível adquirir e consolidar conhecimentos na área de *software* e gestão de uma linha de produção. Além do conhecimento técnico relativo à programação e gestão, também foi possível instruir-nos com uma melhor ética e organização de trabalho em equipas recorrendo à técnica SCRUM, dada a exigência e complexidade associada ao trabalho.

Relativamente ao produto final, obteve-se um sistema robusto, seguro e rápido. Também se conseguiu obter uma boa interface com o utilizador que permite facilitar o seu uso e entendimento do processo.

Optamos desde início por uma implementação em que todas as decisões são tomadas no MES e são enviadas para o PLC. Sendo o PLC apenas responsável por cumprir o que o MES rege e por garantir que não acontece nenhum evento indesejado como os *deadlocks*, por exemplo. Desta forma, conseguimos assegurar uma maior independência entre sistemas e uma maior flexibilidade na gestão das ordens processadas, mesmo na recepção de ordens mais urgentes que as que estão atualmente em produção. A independência entre sistemas permite-nos reduzir ao máximo comunicações desnecessárias, minimizando desta forma eventuais erros associados a falhas de comunicação.

Uma das maiores dificuldades ao longo do trabalho passou por garantir a sincronização (quando necessária) entre o MES e o PLC. Para tal, sempre que se pretendia sincronizar alguma informação realizava-se um *handshake* entre eles para garantir que a informação tinha sido enviada e recebida corretamente.

Por último, destaca-se o facto de que no nosso sistema a torre das ferramentas nas máquinas desce e sobe à entrada e saída de cada peça, pois consideramos que não faria qualquer sentido lógico a máquina processar uma peça com a ferramenta sempre em cima ou sempre em baixo. Dessa forma o nosso sistema tornou-se 0,5s mais lento por peça.

Concluindo, em grupo chegamos ao consenso que a realização deste trabalho tornou-se bastante enriquecedora do ponto de vista curricular e profissional, visto que foi feito num ambiente relativamente próximo ao vivido em ambiente industrial, desde a existência de um Caderno de Encargos que devemos cumprir até às reuniões semanais com o *Product Owner* (Professor).