Milestone 1 – Executive Summary

Técnicas Avançadas de Programação

Departamento de Engenharia Informática – Instituto Superior de Engenharia do Porto

Prof. Nuno Malheiro
Prof. Nuno Ferreira

24 de abril de 2025

Guilherme Cunha, nº 1201506
José Mendes, nº 1230195
Miguel Domingues, nº 1230200
Pedro Vilarinho, nº 1211149

Índice

Análise

Problema

Este projeto incide sobre o desenvolvimento de uma solução de escalonamento de tarefas de produção com base num ficheiro XML de entrada, que contém recursos físicos e humanos, tarefas, produtos e encomendas. O objetivo principal é transformar estes dados num plano de produção válido, representado num ficheiro XML de saída, respeitando todas as restrições de recursos e tempos.

Algoritmo

O algoritmo segue os seguintes passos principais:

1. **Extração e Validação de Dados (*Parsing*)**
   - Começamos por transformar o XML de entrada em classes de domínio fortemente tipadas (p.e. *Task*, *Product*, *Order, entre outros.*). Esta transformação é validada através do uso de tipos opacos (*opaque types*), que garantem que os identificadores e valores seguem os formatos e restrições estabelecidos.

2. **Processamento Sequencial das Encomendas**
   - Cada encomenda (*Order*) é tratada individualmente, e para cada instância do produto encomendado (respeitando a quantidade pedida), é iniciado um ciclo de escalonamento das tarefas.

3. **Escalonamento das Tarefas de um Produto**
   - Para cada tarefa associada ao produto, são alocados recursos físicos e humanos. O algoritmo:
     - Identifica os tipos de recursos físicos necessários.
     - Seleciona os recursos físicos disponíveis do tipo fornecido, evitando reutilizações.
     - Seleciona também os recursos humanos capazes de manusear os tipos de recursos físicos exigidos pela tarefa.

4. **Alocação de Recursos e Cálculo de Tempos**
   - Cada tarefa recebe um *start* e um *end*, com base na duração (*TaskTime*) e no tempo acumulado global (*globalTime*). Assim, o tempo de execução é contínuo e respeita a ordem de chegada das encomendas. Uma vez atribuída uma tarefa, o tempo global acumula e serve de referência para a próxima tarefa.

5. **Construção do XML de Saída**
   - Os dados de agendamento (***TaskSchedule***) são convertidos num XML conforme o *schema* fornecido, garantindo que cada tarefa fica associada à sua encomenda, número de produto, tempos, recursos físicos e humanos

utilizados.

O algoritmo assume que os recursos são ilimitados em termos de disponibilidade temporal, ou seja, não há sobreposição ou conflito de uso, cada recurso é usado apenas uma vez por tarefa. Isto evita a complexidade de escalonamento paralelo tal como era pedido para esta Milestone 1 - *"The execution of the artifact of milestone 1 should provide a maximum of production time for the input orders."*.

Principais decisões e justificações

- **Separação de responsabilidades:** Foi decidido dividir a lógica em componentes distintos (tipos, domínio, *parsing* XML, e lógica de *scheduling*) para manter uma arquitetura modular, testável e extensível.

- **Uso de tipos opacos (*opaque type*):** Os identificadores principais (como *TaskId*, *OrderId*, *PhysicalResourceId*, etc.) foram implementados como tipos opacos para garantir validações fortes em tempo de compilação e evitar erros.

- **Validações na transformação XML para Domínio:** Toda a validação semântica dos dados (como referências a tarefas inexistentes ou tipos inválidos) foi centralizada nos métodos de *parsing* (*XMLToDomain*), garantindo que os dados do domínio estão sempre num estado consistente.

Implementação

O algoritmo de escalonamento foi integrado num sistema já existente através do módulo *ScheduleMS01*, responsável por organizar sequencialmente as tarefas, alocar os recursos necessários e gerar o XML de saída com o agendamento final.

A função principal, *create*, já fazia parte do sistema e foi complementada com toda a lógica necessária. O seu objetivo é transformar um ficheiro XML de entrada num ficheiro XML de saída, com tarefas corretamente agendadas e recursos alocados de forma eficiente.

1. **Leitura e Validação dos Dados de Entrada**
   - A função ***scheduleDataRetriever*** é responsável por extrair os dados do XML de entrada e validá-los, transformando-os em objetos de domínio específicos, como *PhysicalResource*, *Task*, *HumanResource*, *Product* e *Order*. Cada um desses objetos é validado através da tipagem forte, o que significa que utilizamos *opaque types* e as funções de validação associadas para garantir que os dados estejam corretos e sigam as restrições estabelecidas.

   - A leitura e transformação dos dados é feita por meio de chamadas a funções do módulo *XMLToDomain*, que lidam com as diferentes partes do XML e as convertem em tipos de dados do domínio. Qualquer erro na leitura ou na

transformação dos dados resulta num *DomainError* adequado, retornado como parte do tipo *Result*.

2. **Alocação de Recursos**
   - Depois de verificar que todos os dados de entrada estão corretos, o algoritmo começa a distribuir os recursos necessários para cada tarefa. Primeiro cuidamos dos recursos físicos e só depois alocamos os recursos humanos. Para cada tarefa, identificamos exatamente o que ela precisa e reservamos esses recursos conforme sua disponibilidade e características, garantindo que tudo esteja pronto na hora certa.

   - As funções ***allocatePhysicalResources*** e ***allocateHumanResources*** são responsáveis por alocar recursos necessários para executar uma tarefa. Elas funcionam de forma similar a um processo de alocação onde, para cada tipo de recurso que uma tarefa precisa, o sistema tenta encontrar um recurso disponível correspondente. Caso, o recurso necessário não estiver disponível, a função retorna um erro, mais especificamente um ***DomainError.ResourceUnavailable***, informando que o recurso não pôde ser alocado para a tarefa.

   - O processo de alocação é tratado por meio de um tipo ***Result***. Quando os recursos são alocados corretamente, a função retorna um ***Right***. Caso contrário, se algo der errado, como a indisponibilidade do recurso, o retorno será um ***Left***, contendo o erro de domínio relacionado.

3. **Geração do Escalonamento**
   - A função principal para criar o escalonamento é a ***generateSchedule***. Ela recebe os recursos que já foram validados e organiza esses recursos em uma lista de objetos ***TaskSchedule***. Cada ***TaskSchedule*** contém informações importantes sobre o agendamento de uma tarefa, como o ***orderId***, o ***productNumber***, o ***taskId***, o horário de início e fim da tarefa, além dos ***physicalResourceIds*** e ***humanResourceNames***. As tarefas são organizadas de forma cronológica, ou seja, é definido quando cada tarefa começa e termina. Antes disso, os recursos necessários são alocados.

   - O algoritmo percorre as encomendas e seus respetivos produtos de forma sequencial. Para cada encomenda, o sistema verifica o produto associado e, em seguida, as tarefas relacionadas a esse produto. As tarefas são alocadas uma a uma, e os tempos de execução de cada tarefa são calculados de acordo com a ordem em que elas são executadas.

4. **Conversão para XML**
   - Uma vez que o escalonamento foi gerado com sucesso, a função ***toXml*** converte a lista de *TaskSchedule* num objeto XML, de acordo com o *schema* definido. Cada tarefa agendada é representada no XML: identificadores da

encomenda, número do produto, ID da tarefa, horários de início e fim, e os recursos físicos e humanos alocados para a tarefa.

5. **Tratamento de Erros com *Result* e *DomainError***
   - Uma das principais decisões na implementação foi a escolha de usar o tipo ***Result*[A]** (que é basicamente um ***Either[DomainError, A]***). Esse tipo de dado pode ser um ***Right[A]*** (quando o valor é válido) ou um ***Left[DomainError]*** (quando ocorre um erro relacionado ao domínio).Esta abordagem foi escolhida para garantir que os erros sejam tratados de forma clara e estruturada, sem recorrer ao uso de exceções ou manipulação de estado mutável.

   - Os erros são encapsulados na enumeração ***DomainError***, que define vários tipos de erros possíveis, como ***InvalidProductId***, ***InvalidPhysicalId***, ***InvalidOrderId***, e ***ResourceUnavailable***, entre outros. Ao longo do processo de escalonamento, sempre que ocorre um erro, ele é imediatamente reportado como um ***DomainError*** adequado, permitindo uma rastreabilidade clara e uma melhor compreensão do que deu errado no processo.

   - Esta abordagem foi escolhida para garantir que os erros sejam tratados de maneira clara e estruturada. Em vez de depender de exceções ou manipular o estado de forma mutável, esse tipo de estrutura permite lidar com erros de forma explícita, facilitando o controle do fluxo da aplicação e melhorando a legibilidade do código.

Alternativas

Na nossa implementação, optamos por utilizar ***opaque types*** para validações de domínio através de "*smart constructors*", porém uma alternativa seria o uso de case classes com validações (por exemplo o "*require*"), no entanto, consideramos que a nossa abordagem foi melhor, na medida em que os *opaque types* impedem a criação de valores inválidos sem que passem pelas validações definidas.

Atual implementação:

```scala
opaque type ProductId = String
  object ProductId:
    def from(id: String): Result[ProductId] =
      val pattern = "^PRD_.*$".r
      if pattern.matches(id) then Right(id)
      else Left(InvalidProductId(id))

    extension (id: ProductId)
      @targetName("ProductIdTo")
      def to: String = id
```

Alternativa:

```scala
case class ProductId(value: String) {
  require(value.matches("^PRD_.*$"), "ID inválido!")
}
```

Melhorias futuras

- Introdução de um mecanismo de paralelismo ou concorrência de tarefas, para otimizar o tempo total de produção quando existem recursos disponíveis.

- Possibilidade de tratar dependências entre tarefas (ex: tarefas com precedência), o que atualmente não é suportado.