MIS-PL / Management Information System

Product Line

CMS- College Management System

PR1

EDOM

2017/2018

Evaristo Figueiredo - 1010836

Pedro Rodrigues - 1131151

Índice

1. Organização trabalho

1. Criar ambiente de trabalho

1.1. Armazenamento: Google drive
1.2. Editor : Goggle docs
1.3. Diagramas: https://www.draw.io

2. Requisitos

2.1. Criar linha de produção (fabrica software)
  2.1.1. Grupo
2.2. DSL
  2.2.1. Pedro
2.3. PLANTUML
  2.3.1. Evaristo

3. Linha de produção

3.1. Engenharia de domínio

3.1.1. Criar meta-modelos

3.1.1.1. Tecnologias
  3.1.1.1.1. Ecore, OCL
3.1.1.2. Artefactos
  3.1.1.2.1. MindMap.Ecore
  3.1.1.2.2. Requisitos.Ecore
3.1.1.3. Passos (Exercício 2)
  3.1.1.3.1. Cria projecto ecore model (para usar editor gráfico)
  3.1.1.3.2. Criar mm.ecore
    3.1.1.3.2.1. Utilizar base
    3.1.1.3.2.2. Criar no editor gráfico a representação do ecore
  3.1.1.3.3. Criar OCL para validar mm.ecore
    3.1.1.3.3.1. Titulo preenchimento obrigatório
    3.1.1.3.3.2. Data fim tem que ser depois de início
    3.1.1.3.3.3. Ver outros (Ex.2)
  3.1.1.3.4. Criar req.ecore
    3.1.1.3.4.1. Utilizar base
    3.1.1.3.4.2. Criar no editor gráfico a representação do ecore
  3.1.1.3.5. Criar genModel para os 2 ecores
  3.1.1.3.6. Criar projectos de plugin para o genModel

3.1.2. Fazer transformações entre eles

3.1.2.1. Tecnologias
  3.1.2.1.1. ATL
3.1.2.2. Artefactos
  3.1.2.2.1. mm2req.atl
  3.1.2.2.2. req2uc.atl
3.1.2.3. **Passsos (Exercício 3)**

3.1.2.3.1. Adicionar ao eclipse o plugin para ATL: ATL SDK, ATL EMFTVM
3.1.2.3.2. Usar a linguagem ATL para transformar componentes de um modelo noutro
3.1.2.3.2.1. Criar ficheiro mm2req.atl
3.1.2.3.2.2. Criar ficheiro req2ana.atl
3.1.2.3.2.3. Criar ficheiro ana2des.atl

3.1.3. Criar gerador para criar os código necessário para implementar (ACCELEO)

3.1.3.1. Manutenção de contactos e estudantes
3.1.3.2. Usando o formato da plataforma GWT
3.1.3.3. Não é preciso o passo intermédio com o UML design

4. Engenharia aplicacional

4.1. Criar instâncias

4.1.1. Tecnologias
4.1.1.1. Os plugins da engenharia de domínio
4.1.1.2. Run configurations ATL e ACCELEO
4.1.2. Artefactos
4.1.2.1. Instâncias de mindmap e de requisitos
4.1.3. Passos
4.1.3.1. Preencher os dados necessários para criar um mindmap que capture o domínio do problema
4.1.3.2. Transformar as instâncias do modelo mindmap para o modelo de requisitos
4.1.3.3. Completar os dados necessários para criar a especificação de requisitos de acordo com os dados que dispomos do mindmap
4.1.3.4. Transformar o modelo de requisitos no de análise
4.1.3.5. Completar os dados necessários para criar a especificação da análise de acordo com os dados que dispomos dos requisitos
4.1.3.6. Transformar o modelo de análise no modelo de design
4.1.3.7. Melhorar e completar este último modelo
4.1.3.8. Por fim executar a geração de código JAVA recorrendo ao ACCELEO

5. Criar uma análise teórica sobre a utilização da linha de produção na criação de outras aplicações recorrendo á referida linha

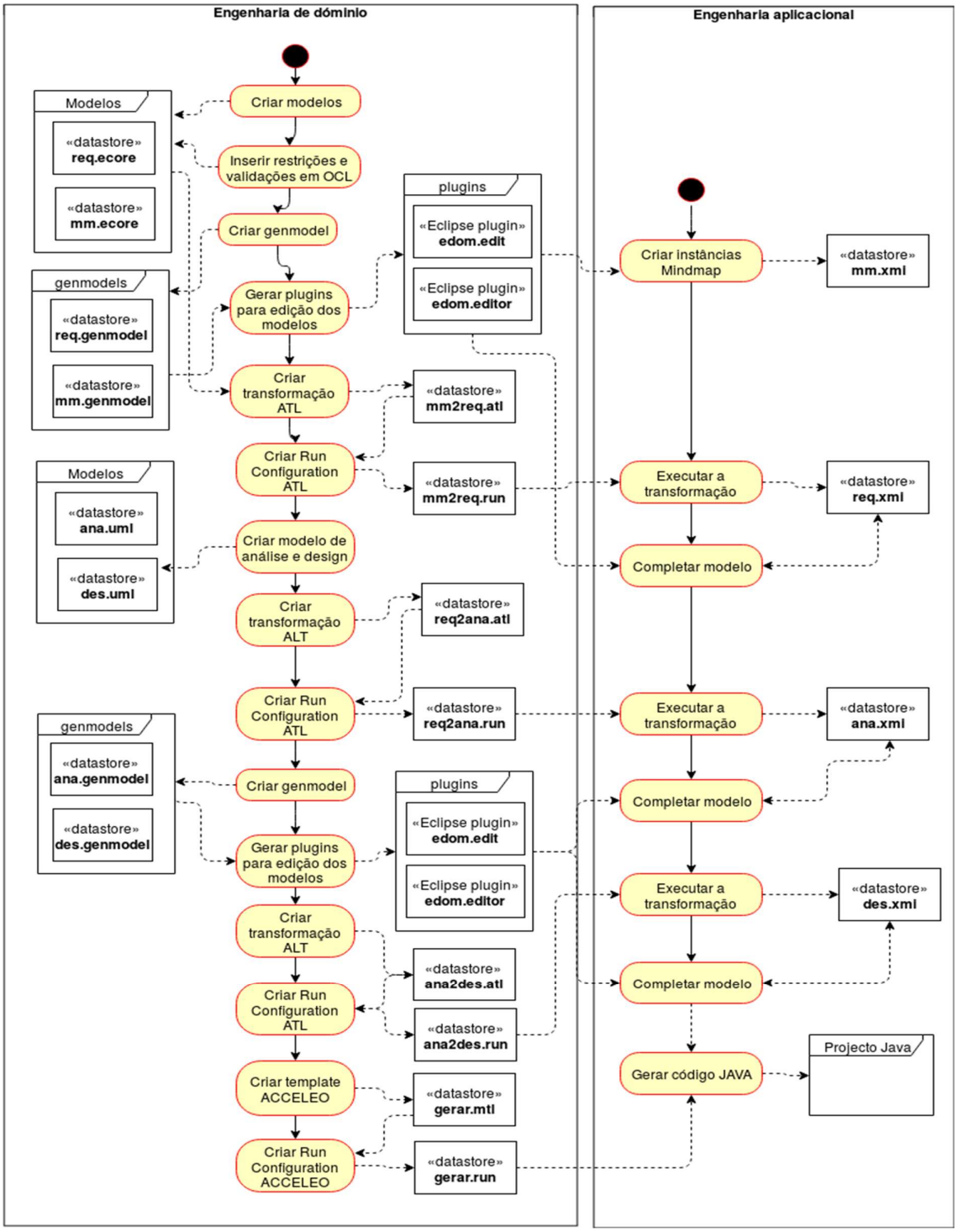
5.1. Se criarmos uma aplicação com professores e cursos em vez das entidades actuais
5.1.1. Quais modificações seriam necessárias

6. Distribuição do trabalho

Tarefa	**Aluno**
Linha de produção	
Modelos mm e req	Pedro
OCL para mm	Pedro
ATL mm2req	Pedro
ATL req2ana	Evaristo
ATL ana2des	Evaristo
ACCELEO	Evaristo
DSL	Pedro
PlantUML	Evaristo

2. Linha de produção

2.1 Diagrama de Atividades



2.2 Descrição

2.2.1 Modelos

O primeiro passo da engenharia de domínio, consiste na criação de dois metamodelos, utilizando a tecnologia ecore. São eles: o modelo Mindmap e o modelo Requirements.

O modelo mindmap é composto pelos seguintes elementos:

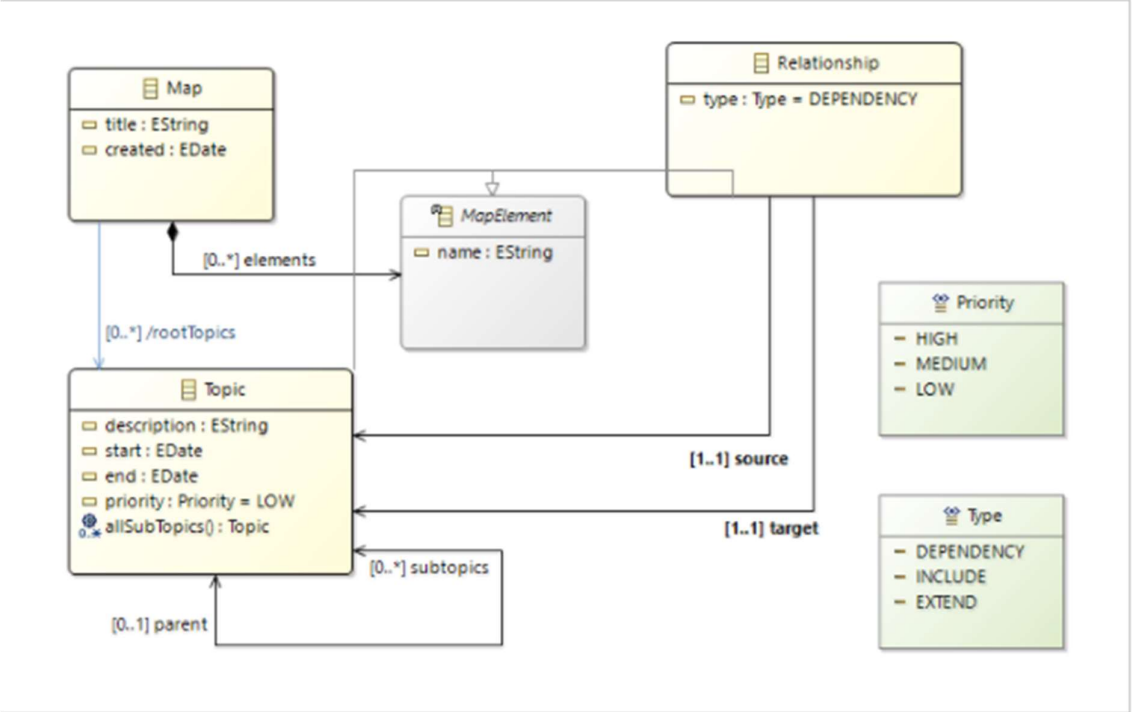


Figura 2 - Modelo Mindmap

- Entidade Map, com os atributos title e created. Cada map irá conter vários objetos do tipo MapElement.
- Topic e Relationship são classes "filho" de MapElement, logo vão herdar o atributo name.
- Existem duas ligações entre Topic e Relationship. Assim poderá existir uma relação entre dois topics, em que um é o target e outro é o source.
- Relationship contém um atributo do tipo Type, que indica o tipo de relação.

- Topic, contém uma relação bidireccional consigo mesmo, que irá permitir ao Topic ter um atributo parent, que indica o seu tópico pai e à sua respectiva lista de subtopics (tópicos filhos);

- Topic ainda contém o método, allSubTopics(), que irá permitir listar todos os seus tópicos filhos.

Em relação ao modelo requirements, este possui os seguintes elementos:

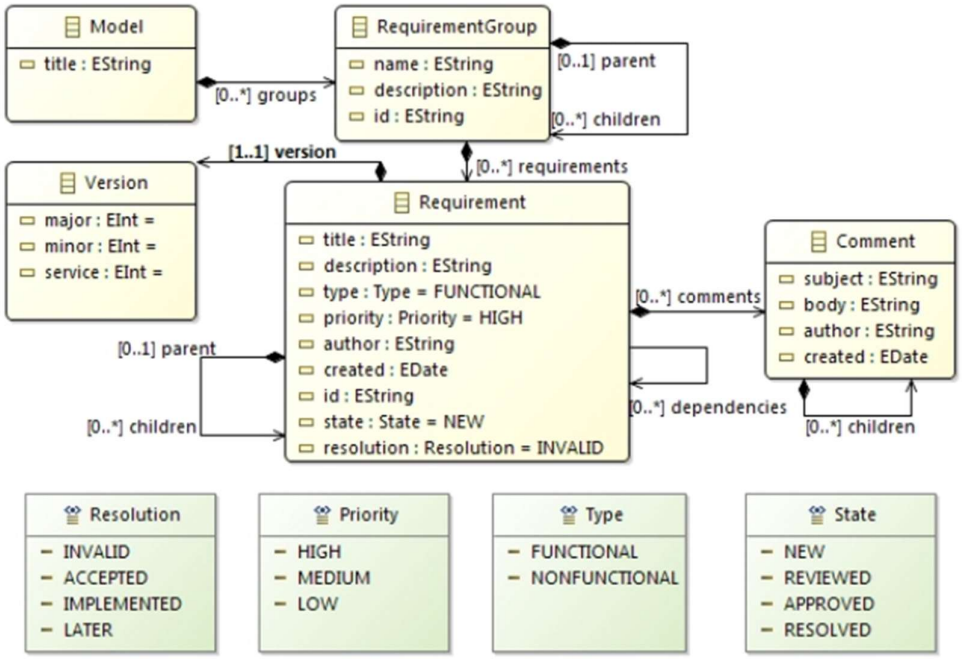


Figura 3 - Modelo Requirements

- Entidade Model, com os atributos title. Cada model irá conter vários objetos do tipo RequirementsGroup.

- Cada Requirement irá ter duas referências parent e children, que permite que um RequirementGroup seja pai ou filho de outro RequirementGroup.
- Cada RequirementGroup, poderá também ter um conjunto de Requirements filhos.

- Para além dos seus atributos, os Requirements têm também duas referências parent e children, que permitem estabelecer ligações de herança entre os Requirements.

Em cada modelo ecore, é necessário também editar as configurações OCL, declarando regras que permitem garantir que os metamodelos atendem a todos os requisitos de negócio necessários.

Depois de definidos os metamodelos é necessário proceder para a criação de instâncias, de maneira a modelar o problema em questão.

A criação das instâncias do mindmap segue a seguinte estrutura:

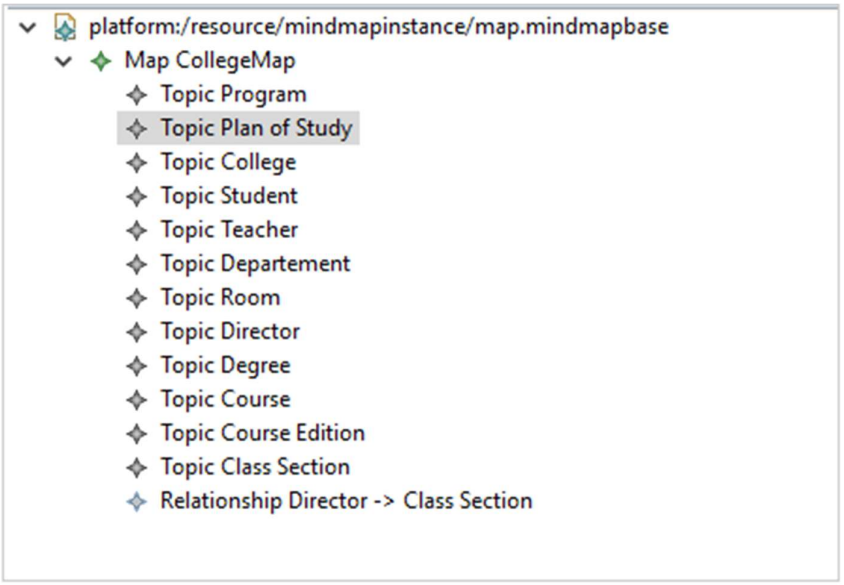


Figura 4 - Instâncias Mindmap (Exemplo: College)

Como se pode observar, cada tópico é irmão do outro, mesmo sendo seu pai, logo as relações pai- filho são estabelecidas através dos atributos parent e subtopics.

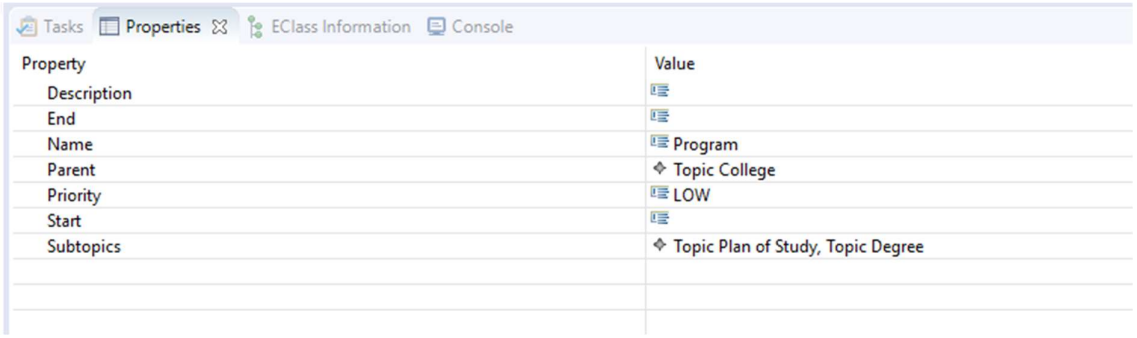


Figura 5 - Propriedades de um Topic no Mindmap

Isto deve-se ao facto do Map conter vários MapElements, e de um Topic não ter uma relação de composição com outro topic, algo que irá ser alterado na solução alternativa.

O próximo passo, será a criação de transformações ATL. Estas transformações irão permitir transformar instâncias mindmap em instâncias requirements e transformar instâncias requirements em modelos Use Case.

Em relação à transformação ATL: Mindmap para Requirements esta possui as seguintes regras:

2.2.2 Transformação de Map (Mindmap) para Model (requirements):

```
--implementation of requirement 1
rule Map2Model {
    from
        s: MMMindmap!"Map"
    to
        t: MMRequirements!Model (
            title <- s.title,
            groups <- s.elements->select(x | x.oclIsKindOf(MMMindmap!Topic) and x.oclAsType(MMMindmap!Topic).parent.oclIsUndefined()
        )
}
```

Figura 6 - Regra ATL Map2Model

Esta regra irá transformar as instâncias Map para Model. Cada model irá ter um título que será igual ao título do map, e uma referência groups que irá conter todas as instâncias Topic que terão que ser transformadas em RequirementGroup.

2.2.3 Transformação de Topics em RequirementGroup:

```
--implementation of requirement 2 : for topics that are not root topics
rule Topic2RequirementGroup {
    from
        s: MMMindmap!Topic (not s.parent.oclIsUndefined())

    to
        t: MMRequirements!RequirementGroup (
            name <- s.name,
            description <- s.description,
            children <-  s.subtopics,
            parent <- s.parent
        )

    do {
        -- ('With Parent: '+s.name).debug();
        -- s.parent.name.debug();
        thisModule.contIdRG <- thisModule.contIdRG.inc();
        t.id <- 'G' + thisModule.contIdRG.toString();
    }
}
```

Figura 7 - Regra ATL Topic2RequirementsGroup

Neste caso a regra Topic2RequirementGroup, irá transformar os Topics do Map para RequirementGroup. Cada RequirementGroup irá receber do Topic, um nome, uma descrição, uma referencia "children" com os seus RequirementGroup filhos e uma referencia pai que contêm os seus RequirementGroup pai. Também cada RequirementGroup irá ter um id, sendo o seu valor incrementado pelo helper "contIdRG", cada vez que um RequirmentGroup é criado.

No final, o resultado da transformação é o seguinte:

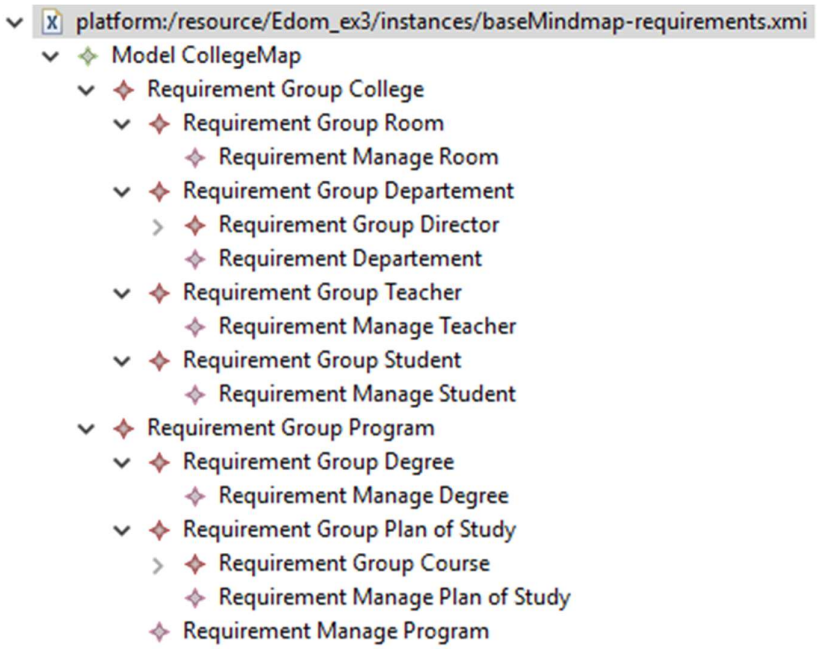


Figura 8 - Resultado transformação ATL Mindmap para Requirements

Em comparação com o modelo mindmap, podemos observar que neste caso a principal diferença consiste no facto de que um elemento filho não é irmão de um pai como era no mindmap, permitindo assim evidenciar melhor as relações hereditárias entre as entidades.

Em relação à transformação ATL: Requirements para Use Case:

2.2.4 Requirements Model para UC Model:

```
--implementation of requirement 1
rule ModelReq2ModelUML {
    from
        s: MMRequirements!Model
    to
        t: MMUML!Model (
            name <- s.title,
            packagedElement <- s.groups
        )
}
```

Figura 9 - Regra ATL ModelReq2ModelUML

Esta regra irá permitir a conversão do modelo do Requirement para um modelo Use Case. O modelo Uc vai ter um título e uma referência "packgedElement" que irá conter todos os requirementsGroups do modelo Requirement.

2.2.5 RequirementGroup para Component:

```
--implementation of requirement 2
rule ReqGroup2CompUML {
    from
        s: MMRequirements!RequirementGroup ( not s.children.isEmpty() )
    to
        t: MMUML!Component (
            name <- s.name,
            packagedElement <- s.children
        )
}
```

Figura 10 - Regra ATL ReqGroup2CompUML

Neste caso, esta regra irá permitir transformar os requirementGroup em componentes do Uc, criando através da referência "children" de requirementGroup, as respectivas ligações de parentesco entre os componentes.

2.2.5 Transformação de Requirement para Use Case:

```
--implementation of requirement 3
rule OtherRequirementGroup2Component {
    from
        s : MMRequirements!RequirementGroup ( s.children.isEmpty() )
    to
        t: MMUML!Component (
            name <- s.name  ,
            ownedUseCase <- s.requirements -> select (r | (r.type = #FUNCTIONAL) )
        )
}


rule Req2UC {
    from
        s : MMRequirements!Requirement (s.type=#FUNCTIONAL)
    to
        t : MMUML!UseCase (
            name <-  s.title,
            ownedUseCase <- s.children -> select (r | (r.type = #FUNCTIONAL) )
        )
}
```

Figura 11 - Regra ATL OtherRequirement2Component

Esta última transformação irá permitir a conversão de requirement para use case. Logo a regra OtherRequirementGroup2Component vai buscar todos requirement filhos de cada requirementGroup, através da referência "requirements" e converte-los em UseCase através da regra Req2UC. Esta última regra irá também transformar os filhos requirement em UseCase através da referência "children".

No final o resultado da transformação é o seguinte:

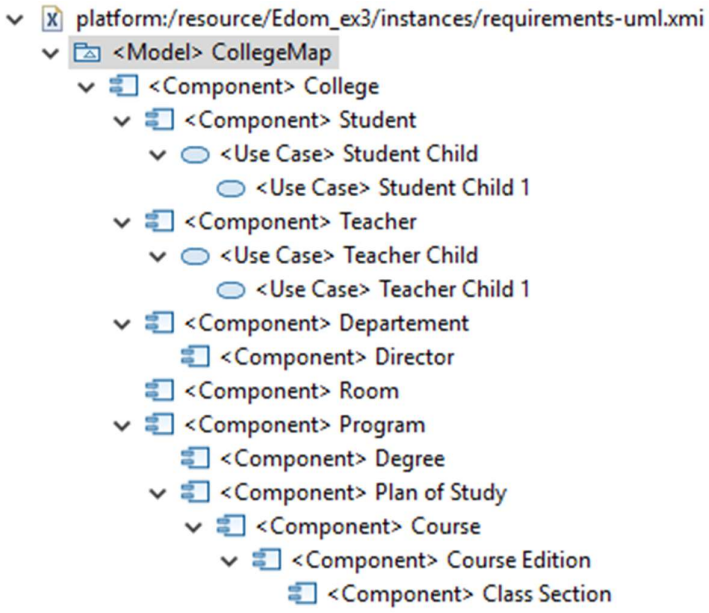


Figura 12 - Regra Resultado da transformação ATL para Use Case

2.2.6 Análise e design

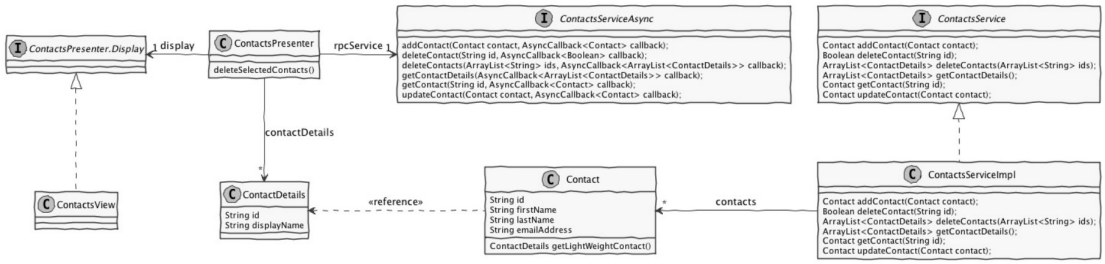
No enquadramento subjacente a realização deste projeto, foi assumido a etapa do processo, designada de "analise" deveria ser capaz de capturar o domínio do problema de uma forma bastante abrangente. Sendo que na fase seguinte ("design") é requerido que de entre o grupo alargado de entidades previamente consideradas, seja extraída informação referente aquelas para as quais é pretendido gerar de forma automática o código JAVA para as opções de CRUD.

Por convenção foi assumido que, a forma de modelar as premissas previamente enunciadas, deveria assumir a forma da instanciação de um elemento do tipo "USE CASE" para cada uma das entidades a serem consideradas relevantes para o "design".

Segundo a mesma convenção, foi também acordado que o relacionamento entre os referidos elementos "USE CASE" e as respetivas entidades deveria ser representado através da subordinação de um elemento "COMMENT" aos "USE CASE" onde a propriedade "source" do mesmo deve conter a expressão "http://pt.isep.edom/crud".

Das condicionantes previamente enunciadas, emerge a necessidade de que a principal transformação ATL, apenas se aplique ao subconjunto de instâncias existentes no modelo de "análise" que, respeitem as mesmas.

Após salvaguardar a condicionante anterior, a referida transformação "ATL" deve ser capaz incorporar a informação do modelo de analise e fazendo uso da mesma, construir o modelo de design recorrendo a elementos UML padrão, (como classes, propriedades e operações) por forma a criar um diagrama de classes que represente a pattern arquitetural utilizada pelo framework GWT para modelar a implementação das operações de CRUD.



A regra de transformação principal recorre a um conjunto de regras auxiliares que suportam a realização de operações acessórias, como a instanciação de novos elementos e ou importação dos atributos das classes de análise para as suas congéneres de design.

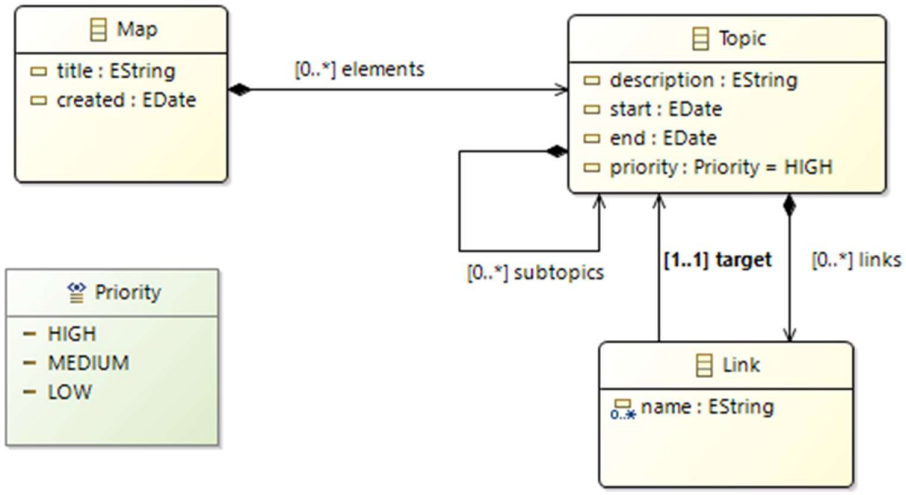
2.2.7 Geração de código JAVA

Nesta fase do projeto, tendo em conta que optamos por criar um modelo intermédio de design, apenas é necessário criar um template ACCELEO que realiza a transformação desse modelo de design em código java. Como já temos um design "detalhado" o nosso acceleo pode ser "genérico". Basta que saiba transformar classes e interfaces de UML em classes e interfaces java.

2.3 Alternativas

2.3.1 Modelos

Como cenário opcional a criação dos ecores, poderia ter sido usado uma versão do mindmap, que permite implementar as mesmas funcionalidades do modelo requirementes relativamente as ligações pai-filho.



Pois como podemos observar, este mindmap mais simplificado, irá permitir aos Topics terem uma relação consigo mesmo, que irá permitir aos topics conterem outros topics, criando assim a relação pai->filho, algo que não acontecia no mindmap base.

2.3.2 Geração de código

Como cenário opcional ao anteriormente descrito, foi considerada a possibilidade de abdicar da transformação do modelo de análise no modelo de design e realizar a geração de código JAVA, diretamente a partir do primeiro.
Nesta abordagem, com recurso ao ACCELEO, seria gerado diretamente código JAVA para todas as classes, interfaces e os seus respetivos elementos constituintes.
Apesar de ser possível resolver o problema desta forma, uma vez que é código que se pretende gerar, é conjunto finito e bem conhecido de estruturas JAVA, claramente identificadas pela pattern que a plaforma GTW utiliza para estruturar as funcionalidades CRUD, que se pretendem ver implementadas de forma automatizada.

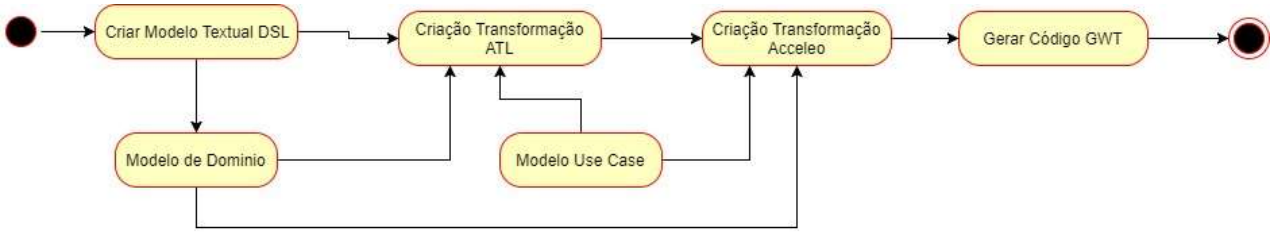
No entanto esta abordagem, embora seja mais direta, foi preterida relativamente à que realiza a transformação intermédia do modelo de análise no de design, pois esta última não acarreta praticamente mais custos de engenharia aplicacional e garante a segurança e flexibilidade que nos são aportados pela existência do modelo de design que pode ser validado e manipulado antes de gerar o código JAVA. Sendo que partindo deste modelo

intermédio o ACCELEO apenas precisa de implementar uma estratégia genérica de conversão de elementos padrão UML em código.

Esta forma é menos propensa a erros e a transformação ACCELEO sendo genérica pode ser aproveitada noutras linhas de produção.

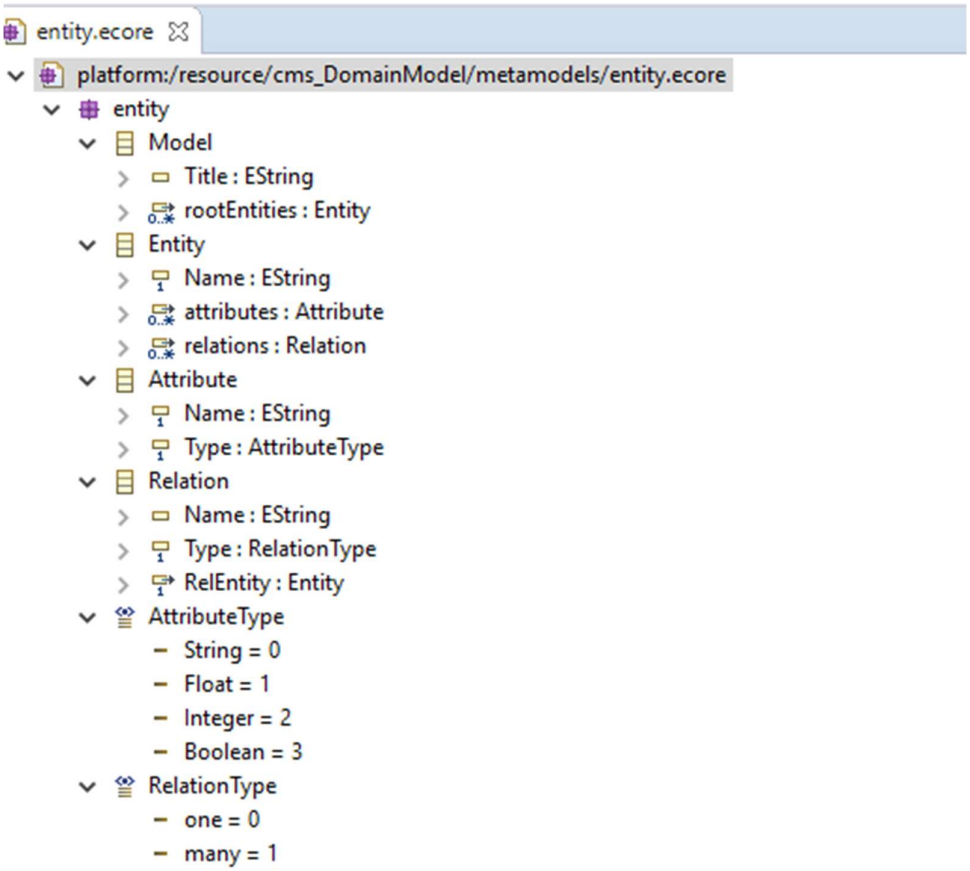
3. Textual DSL for Domain Modeling

3.1 Diagrama de Atividades



3.2 Análise

O objetivo desta tarefa era criar um DSL para o modelo de domínio, evitando assim a modelação do mesmo em UML. Logo para isso foi criado um novo modelo dsl onde as entidades, a sua estrutura e os seus relacionamentos, foram modelados em um modelo dsl chamado “entity”.



Depois de acordo com o dsl, foi criada uma gramática, para que fosse possível criar instâncias do tipo “entity”.

```
RULES {
    @SuppressWarnings(minOccurenceMismatch, maxOccurenceMismatch)
    Model ::= "Model" "{" (
        "Title" ":" Title['"','"']
        ("has entities" ":" rootEntities: Entity)*
    )"}";

    @SuppressWarnings(minOccurenceMismatch, maxOccurenceMismatch)
    Entity ::= "Entity" "{" (
         "Name" ":" Name['"','"']
        ("attributes" ":" attributes: Attribute)*
        ("relations" ":" relations: Relation)*
    )"}";

    @SuppressWarnings(minOccurenceMismatch, maxOccurenceMismatch)
    Attribute ::= "Attribute" "{"
        ("Name" ":" Name['"','"']
        ("Type" ":" Type[String:"String", Float:"Float", Integer:"Integer", Boolean:"Boolean"]
    )"}";

    @SuppressWarnings(minOccurenceMismatch, maxOccurenceMismatch)
    Relation ::= "Relation" "{"
        ("Name" ":" Name['"','"']
        "Type" ":" Type[one:"one", many:"many"]
        ("RelEntity" ":" RelEntity[])*
    )"}";
```

Passou-se à criação da instância do modelo de domínio, tendo sido criadas as entidades, contact e student.

```
Model{

    Title:"College"
    has entities: Entity {
        Name: "Student"
        attributes: Attribute {
            Name: "name"
            Type: String
        }
        relations: Relation {
            Name: "contacts"
            Type: many
            RelEntity: Contact
        }
    }

    has entities: Entity {
        Name: "Contact"
        attributes: Attribute {
            Name: "name"
            Type: String
        }
        relations: Relation {
            Name: "students"
            Type: many
            RelEntity: Student
        }
    }

}
```

Em seguida foi criada a transformação ATL, checkAnalysisModel, que recebendo o analysisModel original irá criar outro, em que os useCases não possuem entidades que não são referenciadas no modelo de dominio. Para isso na regra UseCase2UseCase, é compara cada o nome das entidades do Use Cases com os nomes das entidades do modelo de domínio.

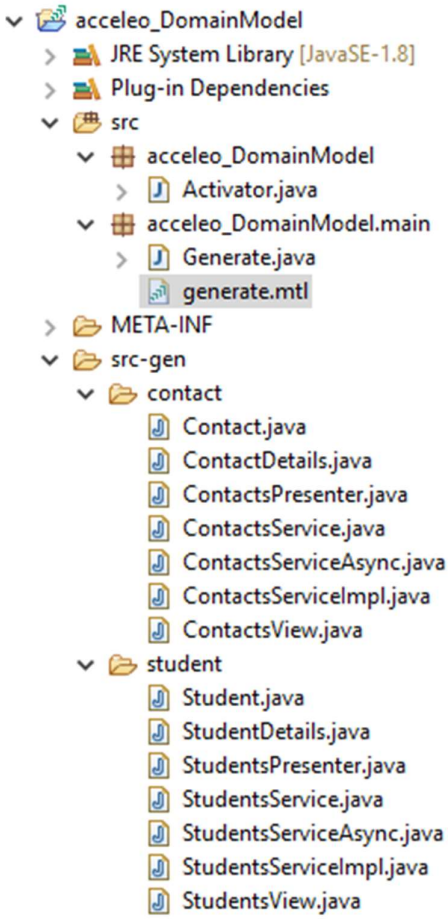
```
module checkAnalysisModel;
create OUT : NewAnalysis from IN : Analysis, IN1 : Domain;


rule Model2Model {
    from
        s : Analysis!Model
    to
        t : NewAnalysis!Model (
           name <- s.name,
           packagedElement <- s.packagedElement,
           ownedType <- s.ownedType
        )
}

rule UseCase2UseCase {
    from
        s : Analysis!UseCase (
            Domain!Entity.allInstances()->select( d | d.Name = s.name.split(' ').last())
            ->asSet()->notEmpty()
    )
    to
        t : NewAnalysis!UseCase (
            name <-  s.name,
            eAnnotations <-  Sequence{tt} --A sequence of created annotations goes to use cas
        ),

        tt: NewAnalysis!EAnnotation (  --rule to create eAnnotations
            source <- 'http://pt.isep.edom/crud'
            )
        do{
        }
}
```

Depois de ter este analysis Model, é feita novamente uma transformação atl, para que o modelo de domínio usado só possua entidades referenciadas no Use Case.

```
-- @atlcompiler emftvm

-- @nsURI DomainModel=http://org/eclipse/dsl/entity
-- @nsURI Analysis=http://www.eclipse.org/uml2/5.0.0/UML
-- @nsURI NewDomainModel=http://org/eclipse/dsl/entity

module syncEntities;
create OUT : NewDomainModel from IN : DomainModel, IN1 : Analysis;

rule AnalysisToModel {
    from
        m1 : DomainModel!Model
    to
        m2 : NewDomainModel!Model (
            Title <- 'College',
            rootEntities <- m1.rootEntities
        )
}

-- Map Entities
rule entitySync {
    from
        e1 : DomainModel!Entity (
            Analysis!UseCase.allInstances()->select( us | us.name.split(' ').last() = e1.Name)
            ->asSet()->notEmpty()
        )
    to
        e2 : NewDomainModel!Entity (
            Name <- e1.Name,
            attributes <- e1.attributes,
            relations <- e1.relations
        )
}
```

Finalmente com o modelo de dominio retificado, é gerado o seu respectivo código através de uma transformação acceleo.

```
[module generate('http://org/eclipse/dsl/entity')]


[template public generateElement(aClass : Entity)]

[comment Model Class @main /]
[file (aClass.Name.toLowerCase() + '/' +aClass.Name+'.java', false, 'UTF-8')]
package [aClass.Name.toLowerFirst()/];

public class [aClass.Name/] {
    [for (a: Attribute | aClass.attributes) separator('\n')]
    private [a.Type/] [a.Name/];
    [/for]

    [for (r: Relation | aClass.relations) separator('\n')]
    [if r.Type.toString().strcmp('many') = 0]
    private ArrayList<[r.RelEntity.Name/]> [r.Name/];
    [else]
    private [r.RelEntity.Name + ' ' + r.Name/];
    [/if]
    [/for]

    private [aClass.Name+'Details'/] details;

    public [aClass.Name + 'Details'/] getLightWeight[aClass.Name/]() {
        return details;
    }

}
[/file]

[comment Detail Class @main /]
[file (aClass.Name.toLowerCase() + '/' + aClass.Name+'Details.java', false, 'UTF-8')]
```



3.3 Alternativas

Uma alternativa para este ponto, poderia ser a criação de uma outra sintaxe para o modelo de domínio DSL. Por exemplo criar uma sintaxe em português mais compacta.

4. PLANTUML

4.1 Problema

1. Este requisito pretende tornar possível a criação de projeções em forma de diagrama de modelos UML, fazendo uso do PlantUML.
2. O projeto base usa UML para o "**modelo de análise**", onde estão representados **modelos de caso de uso**, bem como **modelos de entidade** (usando classes).
3. O objetivo deste requisito é gerar diagramas para ambos os modelos usando PlantUML.
    3.1. De forma idêntica ao exercicio 5, a solução deve ser capaz de gerar Plantuml, isto é, arquivos textuais que representam diagramas dos **modelos de caso de uso**.
    3.2. Também deverá ser possível gerar arquivos textuais de Plantuml que representam **diagramas de classes** para o modelo de entidades.
    3.3. Isto deve ser feito através de transformações Acceleo.
4. A **transformação inversa** também deve ser possível, ou seja, a geração de modelos UML a partir de ficheiros textuais de Plantuml .
    4.1. Deve ser desenvolvida uma DSL textual que suporta o subconjunto de PlantUML usado para estas preocupações.
    4.2. Esta DSL então pode ser usada para "ler" ficheiros de texto contendo PlantUML, sendo que estes serão usados como entrada para uma transformação de ATL que pode produzir modelos UML de arquivos PlantUML.
5. Para este requisito é necessário, fazer o **design** e **implementação** de:
    5.1. Uma **transformação Acceleo** que pode ser usada para **gerar diagramas de caso de uso PlantUML** a partir de modelos de **caso de uso em UML2**;
    5.2. Uma **transformação Acceleo** que pode ser usada para **gerar diagramas de classe PlantUML de modelos de entidade representada no UML2;**
    5.3. Uma **nova DSL textual** que pode representar um **subconjunto da linguagem PlantUML** que ofereça suporte para **diagramas de casos de uso** e **diagramas de classes.**
        5.3.1. A DSL deve ser capaz de modelar os mesmos diagramas como aqueles resultantes dos 2 itens anteriores
    5.4. Uma **transformação de ATL** que pode ser usada para gerar instâncias de **modelos UML2** a partir da **DSL** anterior.

4.2 PlantUML

PlantUML é um projeto Open Source que permite escrever rapidamente:
• Diagrama de sequência,
• Diagrama de casos de uso,
• Diagrama de classes,
• Diagrama de atividades,
• Diagrama de componentes,
• Diagrama de estado,
• Diagrama de objetos.
Diagramas são definidos usando uma linguagem simples e intuitiva.

4.2.1 Diagramas de classes

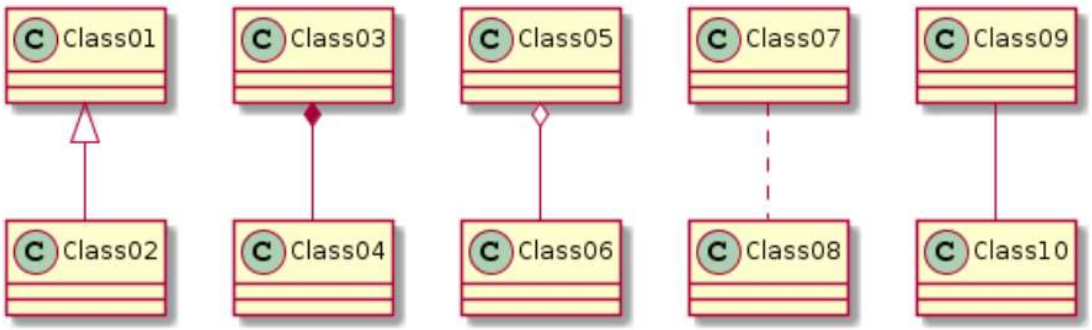
As relações entre classes dão definidas usando os seguintes simbulos:

Extension	<|--	
Composition	*--	
Aggregation	o--	

É possível substituir **"--"** por ".." para obter uma linha de pontos.

Exemplo

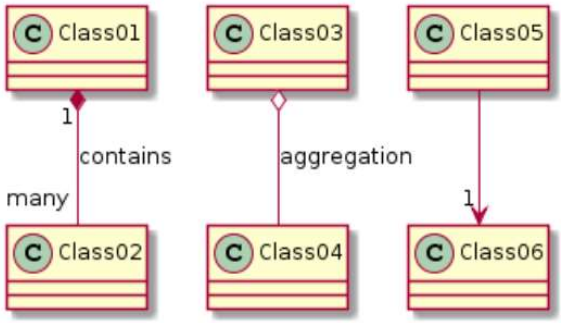
```
@startuml
  Class01          <|--          Class02
  Class03          *--           Class04
  Class05          o--           Class06
  Class07          ..            Class08
  Class09          --            Class10
@enduml
```



É possível adicionar uma etiqueta a uma relação através do uso do caracter "**:**", seguido pelo texto da etiqueta.
Para definir cardinalidade, é possível usar aspas em cada uma dos lados da relação

Exemplo

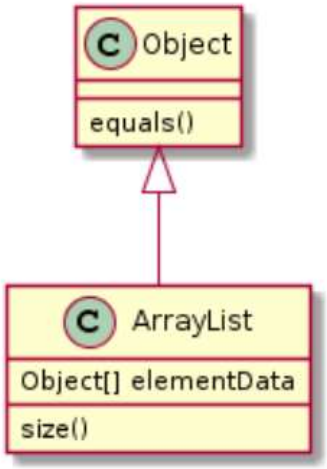
```
@startuml
  Class01 "1" *-- "many" Class02 : contains
  Class03 o-- Class04 : aggregation
  Class05 --> "1" Class06
@enduml
```



Para declarar campos e métodos, é possível usar "**:**" seguido dos nomes dos campos e dos métodos.
Nesta linguagem a distinção entre campos e métodos é definida através do uso de parêntesis depois do nome do método

Exemplo

```
@startuml
Object <|-- ArrayList
Object : equals ()
ArrayList : Object [] elementData
ArrayList : size ()
@enduml
```

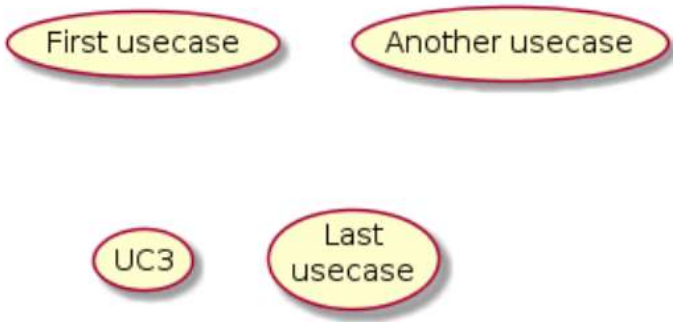


4.2.2 Diagramas de casos de uso

Os casos de uso são inseridos entre parênteses (porque dois parênteses são como uma elipse).
Deve ser usada a palavra-chave "**usecase**" para definir um caso de uso.
E você pode definir um nome usando a palavra-chave "**as"**. Este nome será utilizado mais adiante, quando for necessário definir as relações entre componentes do diagrama.

Exemplo:

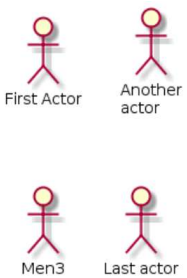
```
@startuml
  (First usecase)
  (Another usecase) as (UC2)
  usecase UC3
  usecase (Last\nusecase) as UC4
@enduml
```



Os atores são delimitados por caracteres doi"
É possível utilizar a palavra chave "**actor"** para definir um ator.
É necessário definir um nome usando a palavra chave "**as"**. O nome será utilizado na definição dos relacionamentos.
As definições de atores são opcionais.

Exemplo

```
@startuml
 :First Actor:
 :Another\nactor: as Men2
 actor Men3
 actor :Last actor: as Men4
@enduml
```

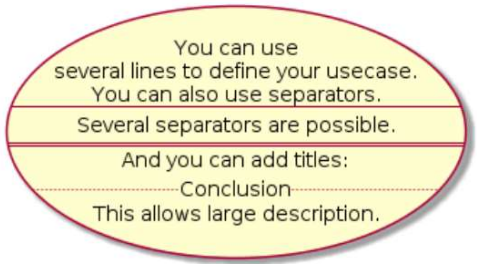


Se for necessário criar uma descrição que utilize várias linhas basta utilizar aspas.

Também é possivel utilizar os seguintes separadores: -- .. == __.
É possível colocar títulos nos separadores.

Exemplo

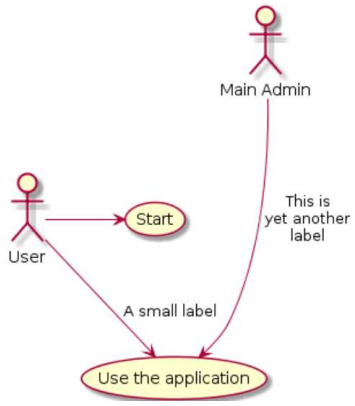
```
@startuml
  usecase UC1 as "You can use
  several lines to define your usecase.
  You can also use separators.
  --
  Several separators are possible.
  ==
  And you can add titles:
  .. Conclusion ..
  This allows large description ."
@enduml
```



Para ligar atores e casos de uso, é usada a seta **-->**.
Quanto mais traços "-", na seta, mais longa a seta será e é possivel adicionar um rótulo na seta, adicionando um caractere **" :"** na definição da seta.

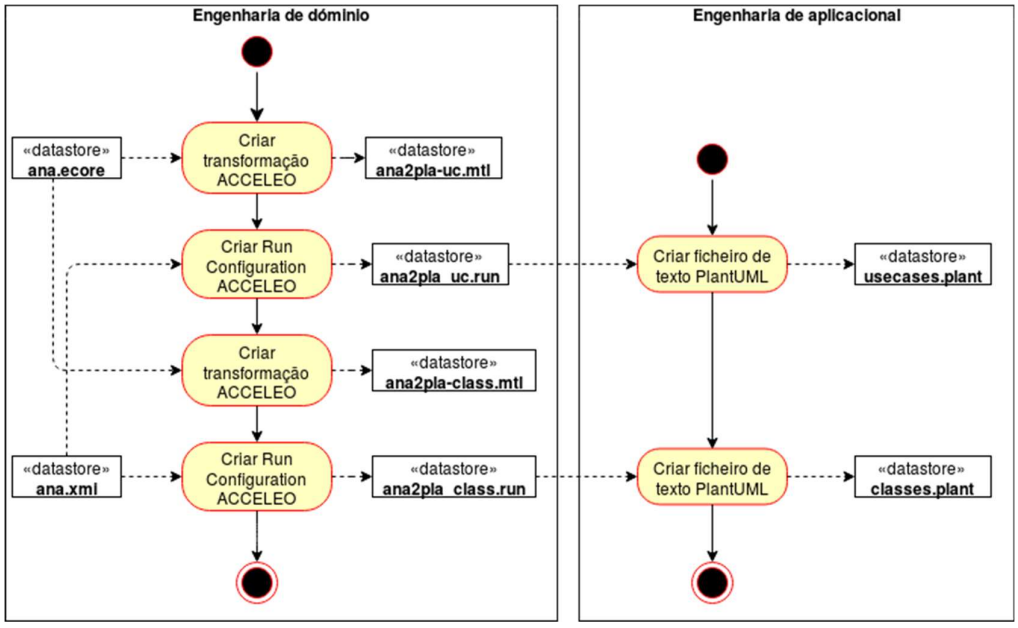
Exemplo

```
@startuml
  User -> (Start)
  User --> (Use the application) : A small label
  :Main Admin: ---> (Use the application) : This is\nyet another\nlabel
@enduml
```

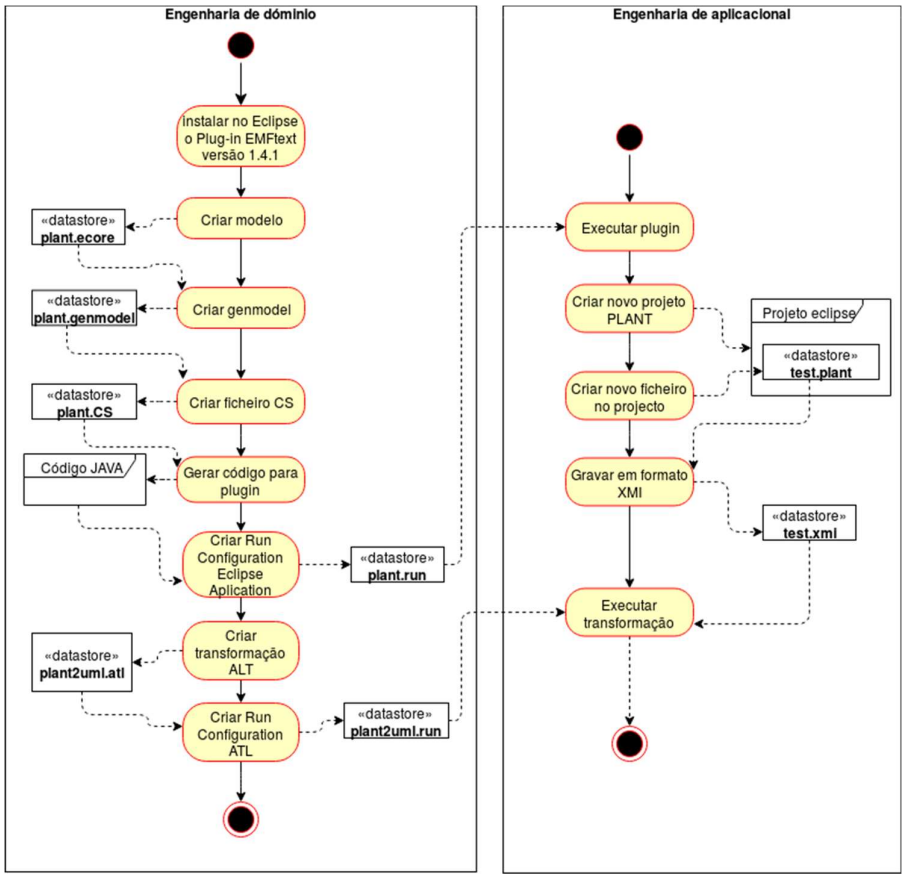


4.3 Diagrama de actividade

4.3.1 UML2PLANT

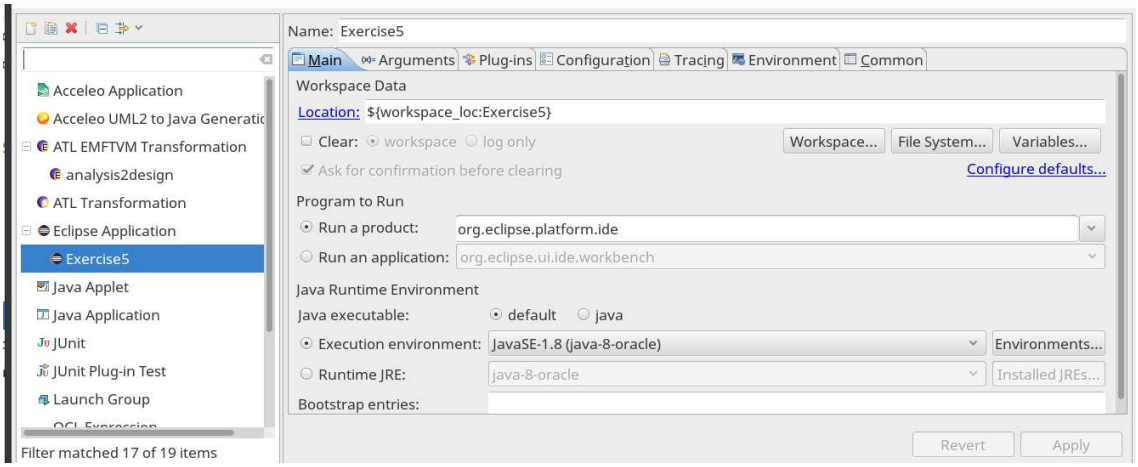


4.3.2 PANT2UML

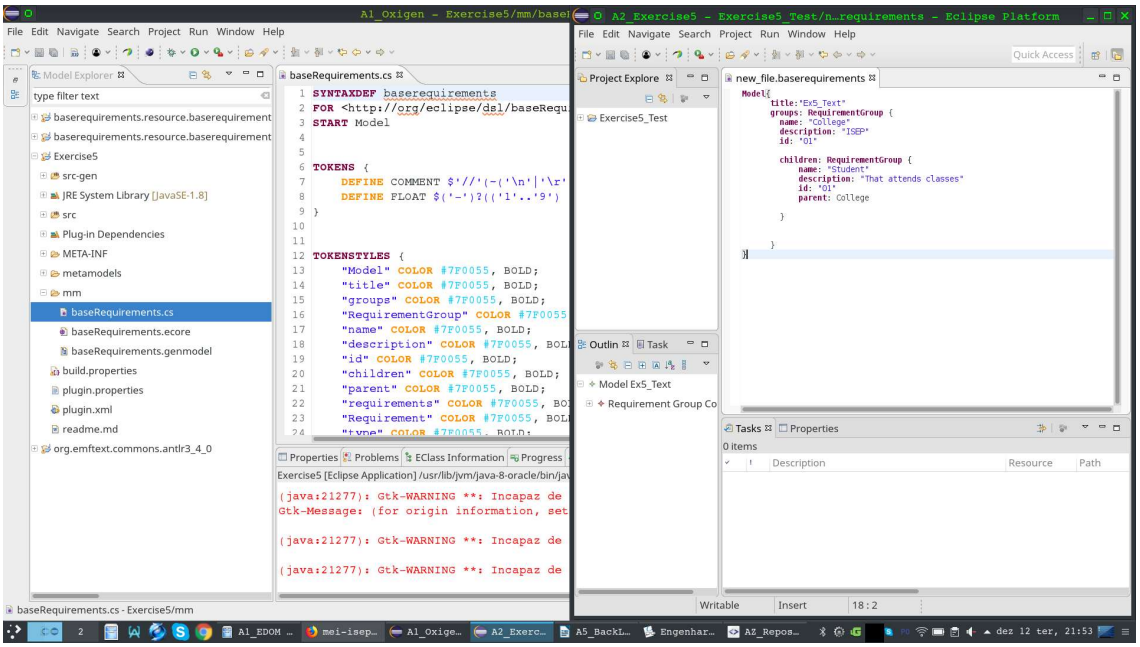


4.4 Descrição

1. **ECORE**
   1.1. Através do editor gráfico “Sirius”, deverá ser criado um modelo que represente a parte da gramática do PlantUML, que permite desenhar diagramas de casos de uso e diagramas de classes.
2. **GENMODEL**
   2.1. Sobre a pasta onde criarmos o ecore escolhemos a opção “New” do menu de contexto.
   2.2. De seguida escolhemos a opção “Other”, Eclipse Modeling FrameWork e EMF Generator Model
   2.3. Escolhemos como origem o ficheiro “ecore”
3. **CS**
   3.1. Sobre o ficheiro “.genmodel” escolhemos a opção “Generate sintaxe” do menu de contexto.
   3.2. De seguida escolhemos a opção “HUTN Sintaxe”
4. **Geração de código**
   4.1. Para gerar o código para os plugins que permitem editar o texto da nossa linguagem.
   4.2. Primeiro abrimos o ficheiro .genmodel
   4.3. Sobre o package principal escolhemos no menu de contexto a opção "Generate Model Code".
   4.4. Por fim sobre o ficheiro .cs, escolhemos no menu de contexto a opção "Generate Text Resource".
5. **Executar o plugin**
   5.1. Para executar o nosso editor é necessário criar uma RUN configuration



   5.2. Executando esta configuração acedemos a uma nova versão do eclipse
      5.2.1. Nesta versão é possível criar um projeto específico que permite agregar ficheiros de texto PlantUML



6. Transformar PlantUML em modelo UML

6.1.1. Depois de criar o ficheiro de texto com o editor especifico para a linguagem PlantUML é possível gravá-lo em formato XMI.

6.1.2. Recorrendo a uma transformação ATL criada pela a equipa de engenharia de domínio é possível, automaticamente transformar as instâncias existentes no ficheiro XMI, num modelo UML2.

5. Análise teórica

Com esta linha de produção em funcionamento, a utilização da mesma para criarmos uma aplicação com professores e cursos em vez das entidades actuais, seria direta.

Assumindo que durante as fases de criação do mindmap e da definição de requisitos as novas entidades também foram contempladas, os colaboradores da área de engenharia aplicacional apenas teriam que definir no modelo de análise novos USE CASES para as novas entidades e relacioná-los com as mesmas.

Todas as tarefas que foram automatizadas pela engenharia de domínio, como as transformações de modelos e de geração de código JAVA, continuam a ser válidas sem requerer alterações.

Desta forma com recurso à engenharia de domínio é possível optimizar substancialmente o processo de desenvolvimento de software, garantir a qualidade do trabalho realizada através das validações automáticas e restrições impostas pelos modelos e estruturar o processo de desenvolvimento.

Em conclusão, podemos afirmar que a abordagem MDD acrescenta um enorme valor ao processo de desenvolvimento e que a adopção da mesma na prática contribui claramente para a diminuição de custos, de tempo de desenvolvimento e para o aumento da qualidade do produto final.