



Universidade do Minho  
Departamento de Informática

Engenharia de Linguagens

Projecto Integrado

*Software* para Análise e Avaliação de Programas

*Grupo 2:*

José Pedro Silva	pg17628
Mário Ulisses Costa	pg15817
Pedro Faria	pg17684

Braga, 12 de Março de 2011

# Abstract

Este relatório trata da primeira fase do Projecto Integrado (PI) da UCE de Engenharia de Linguagens. O PI consiste num sistema de submissão de trabalhos dos alunos. Este sistema deve ser capaz de aceitar registos de alunos e professores. Deve ainda permitir a submissão de código por parte dos alunos e a submissão de exercicios por parte dos professores. O sistema deve ainda ser capaz de fazer uma análise detalhada sobre o código submetido pelo aluno e gerar um relatório para o professor ler sobre esse mesmo código.

Nesta fase o grande objectivo é modelar todo este sistema. Decidiu-se fazê-lo de um ponto de mais técnico, com alguns diagramas e a representação da estrutura dos ficheiros XML que vão ser úteis para a nossa implementação, mas também decidiu-se fazer uma modelação do ponto de vista mais formal, de forma a nesta fase clarificar o processo de modelação e fazer com que se dê maior foco à modelação em si, sem ter de pensar nos sistemas complexos que estão por trás disso.

# Índice

<b>Índice</b>	<b>ii</b>
Índice de Figuras . . . . .	iii
<b>1 Introdução</b>	<b>1</b>
1.1 Motivação . . . . .	1
1.2 Objectivos . . . . .	1
1.2.1 Ferramentas . . . . .	2
1.3 Descrição do Sistema . . . . .	2
1.3.1 Utilizadores . . . . .	2
1.3.2 Funcionalidades do sistema . . . . .	2
1.4 Estrutura do Relatório . . . . .	3
 <b>I Milestone I</b>	 <b>5</b>
<b>2 Modelação do Problema</b>	<b>6</b>
2.1 Modelação Informal . . . . .	6
2.2 Modelação Formal . . . . .	8
2.3 Modelo de Dados . . . . .	11
2.4 Importação de dados . . . . .	12
 <b>II Milestone II</b>	 <b>18</b>
<b>3 Implementação</b>	<b>19</b>
3.1 Criação de grupos, docentes e concorrentes . . . . .	19
3.2 Linguagens de programação . . . . .	19
3.3 Compilação . . . . .	20
3.4 Execução . . . . .	20
3.5 Guardar resultados . . . . .	20
<b>4 Métricas</b>	<b>21</b>
4.1 Análise Estática . . . . .	21
4.2 Análise Dinâmica . . . . .	22
4.3 Métricas de qualidade de <i>software</i> . . . . .	23
4.3.1 Patterns . . . . .	23
<b>5 Aplicação Web</b>	<b>24</b>
<b>6 Interface pelo terminal</b>	<b>25</b>
6.1 Perl . . . . .	25
6.2 Menus e exemplos . . . . .	25
<b>7 Scripts de avaliação</b>	<b>28</b>
<b>8 Front-end</b>	<b>29</b>
<b>9 Conclusão e Trabalho Futuro</b>	<b>30</b>

## Índice de Figuras

2.1	Arquitectura do sistema . . . . .	7
2.2	Modelo de dados - Grupo e Docente/Admin . . . . .	12
2.3	Modelo de dados - Concurso, tentativa e enunciado . . . . .	13
2.4	diagrama do schema para o enunciado . . . . .	16
2.5	diagrama do schema para a tentativa . . . . .	17
6.1	Menu Principal . . . . .	26
6.2	Linguagens disponíveis . . . . .	26
6.3	Procura e alteração do <code>user</code> Zella Douglas . . . . .	27

# 1

## Introdução

### Conteúdo

<b>1.1</b>	<b>Motivação . . . . .</b>	<b>1</b>
<b>1.2</b>	<b>Objectivos . . . . .</b>	<b>1</b>
<b>1.2.1</b>	<b>Ferramentas . . . . .</b>	<b>2</b>
<b>1.3</b>	<b>Descrição do Sistema . . . . .</b>	<b>2</b>
<b>1.3.1</b>	<b>Utilizadores . . . . .</b>	<b>2</b>
<b>1.3.2</b>	<b>Funcionalidades do sistema . . . . .</b>	<b>2</b>
<b>1.4</b>	<b>Estrutura do Relatório . . . . .</b>	<b>3</b>

Este relatório descreve todo o projecto desenvolvido para o módulo Projecto Integrado da UCE de Engenharia Linguagens do Mestrado em Engenharia Informática da Universidade do Minho.

Pretende-se que este projecto seja um *Software* para Análise e Avaliação de Programas (SAAP), tendo este *software* como objectivo criar um ambiente de trabalho, com um interface Web, que permita a docentes/alunos avaliar/submeter programas automaticamente.

### 1.1 Motivação

Este trabalho é muito interessante do ponto de vista de produto final e como obra didáctica para a sua concretização. Para a sua feitura os autores terão de aplicar conhecimentos adquiridos na área de arquitectura de um sistema de informação, desenvolvimento para a web, linguagens de scripting, bases de dados, processamento de textos, etc. . .

Como se pode facilmente concluir, este é um projecto que do ponto de vista técnico é muito motivador devido à sua magnitude e inovação técnica que implica aos seus autores.

### 1.2 Objectivos

Este projecto tem como objectivos consolidar conhecimentos adquiridos nos diferentes módulos da UCE de Engenharia de Linguagens Assim sendo, vamos recorrer a tecnologia aprendida durante as aulas, bem como tecnologia já conhecida pelos autores, aprendida durante a formação académica ou à parte desta. Um dos grandes objectivos é consolidar e desenvolver ainda mais o uso de tecnologias variadas, para arranjar uma solução elegante, funcional e que cumpra os requisitos do sistema descrito. Assim, este projecto é mais do que um projecto de mestrado, passando a ser encarado como um desafio ao conhecimento e a pôr em prática conhecimentos adquiridos.

### 1.2.1 Ferramentas

As ferramentas/tecnologias que estamos a utilizar são as seguintes:

- DB2
- Perl
- Ruby on Rails (RoR)
- Haskell

Fazer-se-à uso de DB2 por suportar nativamente XML e ainda XPath e XQuery para fazer travessias no XML. O uso de Perl prende-se com o facto de ser incrivelmente fácil criar sem muito esforço pequenas ferramentas que acreditamos serem úteis para identificar padrões em texto ou cortar pequenos pedaços de texto.

O uso de RoR deve-se ao facto da simplicidade em criar ambientes web. Decidimos ainda utilizar Haskell eventualmente em tarefas mais complexas, por ser uma linguagem de rápida implementação e bastante segura.

## 1.3 Descrição do Sistema

O SAAP - Software para Análise e Avaliação de Programas é um sistema disponível através de uma interface web, que terá como principal função submeter, analisar e avaliar automaticamente programas.

O sistema estará disponível em parte para utilizadores não registados, mas as suas principais funcionalidades estarão apenas disponíveis para docentes e grupos já registados no sistema.

O sistema poderá ser utilizado para várias finalidades, no entanto estará direccionado para ser utilizado em concursos de programação e em elementos de avaliação universitários.

### 1.3.1 Utilizadores

Existem três entidades que podem aceder ao sistema.

O administrador, que além de poder aceder às mesmas funcionalidades do docente, funcionalidades essas que já iremos descrever, é quem tem o poder de criar contas para os docentes.

O docente tem acesso a todo o tipo de funcionalidades relacionadas com a criação, edição e eliminação de concursos e enunciados, assim como consulta de resultados dos concursos e geração/consulta de métricas para os ficheiros submetidos no sistema.

O grupo, que pode ser constituído por um ou mais concorrentes, terá acesso aos concursos disponíveis, poderá tentar registar-se nos mesmos, e submeter tentativas de resposta para cada um dos seus enunciados.

Além do que já foi referido, todos os utilizadores podem editar os dados da sua conta.

Falta referir que um utilizador não registado (guest), pode criar uma conta para o seu grupo, de modo a poder entrar no sistema.

### 1.3.2 Funcionalidades do sistema

As várias funcionalidades do sistema já foram praticamente todas mencionadas, vamos no entanto tentar explicar a sua maioria, com um maior nível de detalhe.

#### Criação de contas de grupo e de docente

**Criação de conta de grupo:** Qualquer utilizador não registado poderá criar uma conta no sistema para o seu grupo, através da página principal do sistema. Terá de preencher dados referentes ao grupo e aos respectivos concorrentes.

**Criação de conta de docente:** Como já foi referido, o administrador terá acesso a uma página onde poderá criar contas para docentes.

#### **Criação de concursos e enunciados**

Tanto o administrador como o docente podem criar concursos. Depois de preencher todos os dados do concurso podem passar à criação de enunciados. Nesta altura caso prefiram, podem importar enunciados previamente criados em xml. O concurso só ficará disponível na data definida aquando da criação do concurso.

#### **Registo e participação nos concursos**

Um grupo que esteja autenticado no sistema pode tentar registar-se num dos concursos disponíveis, e será bem sucedido se a chave que utilizar for a correcta.

Depois de se registar no concurso, o tempo para a participação no mesmo inicia a contagem decrescente e o grupo poderá começar a submeter tentativas para cada um dos enunciados. O sistema informará, pouco depois da submissão, se a resposta estaria correcta ou não. Depois do tempo esgotar o grupo não pode submeter mais tentativas.

#### **Geração das métricas para os ficheiros submetidos**

O docente ou o administrador a qualquer altura podem pedir ao sistema que gere as métricas para as tentativas submetidas.

#### **Consulta dos ficheiros com as métricas e dos resultados do concurso**

O docente pode aceder aos logs que contêm a informação sobre as tentativas submetidas, ou se desejar, apenas aceder a informações mais específicas, tal como qual exercício tentou submeter determinado grupo, e se teve sucesso ou não.

Pode também visualizar os ficheiros com as informações das métricas.

## **1.4 Estrutura do Relatório**

Este documento está provisoriamente estruturado em duas partes referentes a cada uma das Milestones até à data alcançadas. Dentro de cada parte estará descrito de forma sucinta todo o trabalho realizado separado logicamente por secções.

*Capítulo 1º* O leitor é introduzido ao tema abordado, e é justificada a utilidade do projecto.

### **Parte I**

*Capítulo 2º* Neste capítulo é explicada a modelação e todas as decisões tomadas para a modelação do sistema a que nos propomos a concretizar.

### **Parte II**

*Capítulo 3º* Neste capítulo são expostos algumas soluções relacionadas com a implementação do problema proposto.

*Capítulo 4º* São discutidas os vários tipos de métricas que existem e quais as que interessam para este trabalho.

*Capítulo 5º* É dada especial atenção ao sistema front-end Web que foi desenvolvido.

*Capítulo 6º* Descreve uma implementação de um sistema front-end pelo terminal que tem como finalidade ser usado por administradores.

**Capítulo 7º** São mostrados e explicados a criação e uso de alguns scripts desenvolvidos para fazerem pequenas tarefas para a aplicação.

**Capítulo 8º** É explicado o sistema front-end C feito em Haskell que foi utilizado para avaliar o código submetido.

**Capítulo 9º** No capítulo final são tecidos alguns comentários relativos ao trabalho efectuado e motivação para trabalho futuro.



Parte I

Milestone I

# 2

## Modelação do Problema

### Conteúdo

<b>2.1</b>	<b>Modelação Informal</b>	<b>6</b>
<b>2.2</b>	<b>Modelação Formal</b>	<b>8</b>
<b>2.3</b>	<b>Modelo de Dados</b>	<b>11</b>
<b>2.4</b>	<b>Importação de dados</b>	<b>12</b>

Neste capítulo, pretende-se dar uma descrição textual da várias modelações que foram feitas ao sistema, mais precisamente, a modelação informal, a modelação formal, o modelo de dados e, por fim, a importação de dados.

### 2.1 Modelação Informal

Com o diagrama da arquitectura do sistema, figura 2.1, pretende-se mostrar as várias entidades que podem aceder ao sistema, assim como as várias actividades que cada uma pode realizar e tarefas para o sistema processar. Também é realçada a ideia de que alguns dos recursos do sistema só estão disponíveis ao utilizador depois de passar por outros passos, ou seja, o diagrama dá a entender a ordem pelas quais o utilizador e o sistema podem/devem executar as tarefas.

Para começar, como temos duas entidades diferentes que podem aceder ao sistema (o docente e o aluno/concorrente), dividiu-se o diagrama em duas partes distintas (uma para cada entidade referida), de modo a facilitar a leitura.

Em ambos os casos, o login é a primeira actividade que pode ser realizada. Caso o login não seja efectuado com sucesso, é adicionado ao log file uma entrada com a descrição do erro. Caso contrário, e o login ser efectuado com sucesso, consoante as permissões do utilizador em questão, tem diferentes opções ao seu dispor.

No caso do login pertencer a um docente, este terá acesso aos dados de cada um dos grupos, podendo verificar os resultados que estes obtiveram na resolução das questões do(s) concurso(s), assim como ao ficheiro que contém a análise das métricas dos vários programas submetidos pelos mesmos. Poderá também criar novos concursos e os seus respectivos exercícios, assim como adicionar baterias de testes para os novos exercícios, ou para exercícios já existentes.

No caso do login pertencer a um aluno/concorrente, o utilizador terá a opção de se registar num concurso ou de seleccionar um no qual já esteja registado. Já depois de seleccionar o con-

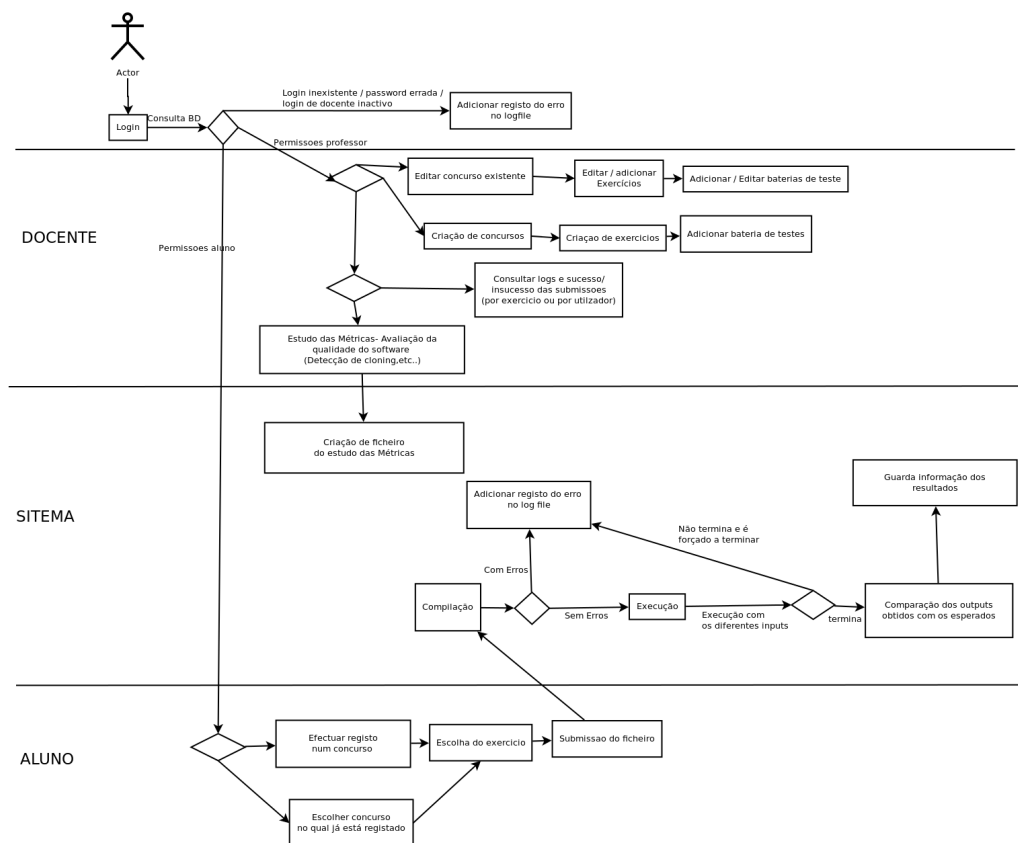


Figura 2.1: Arquitectura do sistema

curso, pode ainda escolher o exercício que pretende submeter. Depois de submeter o código fonte do programa correspondente ao exercício escolhido, e já sem a interacção do utilizador, o sistema compilará e tentará executar os diferentes inputs da bateria de testes do exercício, e comparar os resultados obtidos com os esperados. No fim de cada um destes procedimentos, serão guardados os resultados / erros. Para terminar, será feito um estudo das métricas do ficheiro submetido, tendo como resultado a criação um ficheiro com os dados relativos a essa avaliação.

## 2.2 Modelação Formal

Afim de haver algum rigor na definição do sistema decidiu-se fazer um modelo mais formal do ponto de vista dos dados e das funcionalidades que o sistema apresenta. A ideia desta modelação é ainda ser um modelo formal da especificação atrás descrita. Para isso utilizou-se uma notação orientada aos contratos (design by contract), com a riqueza que as pré e pós condições de funções oferecem.

Assim sendo, os contratos estão descritos da seguinte forma:

$$\begin{array}{c} \{P\} \\ C \\ \{R\} \end{array}$$

em que  $P$  define uma pré condição,  $C$  uma assinatura de uma função e  $R$  uma pós condição. De notar que tanto a pré como a pós condição têm de ser elementos booleanos e devem-se referir à assinatura da função. Assim sendo definiu-se que apenas o contrato  $C$  é válido se a sua pré e pós condições devolverem *true*. A pré e pós condição podem ser vazias.

Neste sistema, definiu-se que a assinatura  $C$  da função pode ter uma particularidade, que é a instânciação de um elemento que pertença a um determinado tipo. Ou seja, pode-se dizer

$$soma :: a \sim Int \rightarrow b \sim Int \rightarrow Int$$

para expressar que a função *soma* recebe dois parametros  $a$  e  $b$  que são inteiros e devolve um elemento do tipo inteiro.

Decidiu-se também, que por motivos de facilidade de leitura e afim de evitar a complexidade formal, não especificar o estado interno do sistema, como por exemplo o estado do Apache, da base de dados entre outros componentes do sistema. Especificar isso não iria trazer nada de interessante ao que se pretende mostrar aqui. Assim, neste modelo formal apenas se vê de forma clara, os contratos que queremos que o nosso sistema tenha.

Começou-se então por definir o contrato da função *login* que permite a um determinado utilizador entrar no sistema. Esse contrato estipula que recebendo um par  $Username \times Hash$  e um  $SessionID$  devolve ou um *Error* ou um novo  $SessionID$  que associa o utilizador à sua sessão no sistema. Afim de haver privacidade sobre os dados críticos do utilizador, como a password, decidiu-se apenas receber do lado do servidor a *Hash* respectiva da sua palavra-passe, sendo esta hash gerada do lado do cliente. Esta técnica não tem nada de novo, mas por incrível que pareça ainda há sistemas online que não usam este tipo de mecanismos.

$$\begin{array}{l} \{existsInDatabase(u)\} \\ login :: u \sim Username \times Hash \rightarrow SessionID \rightarrow Error + SessionID \\ \{\} \end{array}$$

Apresenta-se de seguida o modelo de dados formal que o sistema usa. Considerou-se que um *Exercicio* tem um *Enunciado* e um dicionário a relacionar *Input's* com *Output's*, um concurso

tem um nome, um tipo e um conjunto de exercícios.

```
data Dict a b = (a × b)*
data Exercicio = Exercicio Enunciado (Dict Input Output)
data Contest = Contest Nome Tipo Exercicio*
```

Para criar um novo concurso, é necessário assegurar que o utilizador que requisita este serviço é um professor, visto não sendo requisito alunos criarem concursos. É necessário ainda assegurar que o concurso que se vai criar tem no mínimo um exercício.

```
{existeSession(s) ∧ isProf(s) ∧ (notEmpty ∘ getExercice) c}
createContest :: s ∼ SessionID → c ∼ Contest → 1
{(notEmpty ∘ getDict) c}
```

Para criar um novo exercício, o utilizador tem de ser um professor e o exercício em questão não pode ser repetido no sistema. Decidiu-se desse modo para evitar redundância na informação que se tem armazenada.

```
{existeSession(s) ∧ isProf(s) ∧ (not ∘ exist)(Exercicioed)}
createExercice :: s ∼ SessionID → e ∼ Enunciado → d ∼ (Dictab) → 1
{exerciceCreated(Exercicioed)}
```

Pode-se ainda consultar os logs de um concurso específico. Aqui apresenta-se como argumento todo o concurso - *Contest* para apenas evitar a definição evidente de identificadores. É claro que a implementação desta acção, irá receber como parâmetro o identificador do concurso, como em outros parâmetros deste modelo formal.

```
{existeSession(s) ∧ isProf(s) ∧ contestIsClosed(c)}
consultarLogsContest :: s ∼ SessionID → c ∼ Contest → LogsContest
{}
```

De seguida mostra-se a especificação de efectuar um registo no concurso, pretende-se apenas que o concurso não esteja cheio.

```
{existSession(s) ∧ contestNotFull(c)}
registerOnContest :: s ∼ SessionID → c ∼ Contest → Credenciais
{}
```

Pode-se ainda submeter um exercício, o que implica que esta acção tenha como consequência devolver um relatório com os resultados interessantes para mostrar ao participante de um concurso.

```
{existeSession(s) ∧ exerciceExist(e)}
submitExercicio :: s ∼ SessionID → e ∼ Exercicio → res ∼ Resolucao → rep ∼ Report
{rep = geraReportseres}
```

## Modelação de acções de selecção

Existem acções do sistema proposto que envolvem a selecção de itens nos forms ou então métodos que geram efeitos secundários, assim decidiu-se explicar aqui esse conjunto de contractos. Estes métodos são interessantes de modelar porque, assim fica mais claro ver os parâmetros que recebemos para os concretizar.

Tem-se então a acção de escolha de um exercício numa panóplia de exercícios disponíveis no concurso que o utilizador está actualmente inscrito e a participar.

De relembrar que o uso do tipo 1 significa o tipo unitário, ou seja, do ponto de vista formal esta operação não devolve nada, apenas altera o sistema. Sistema esse que no início explicou-se que por motivos de complexidade não tinha interesse especificar formalmente.

$$\{ \text{existeSession}(s) \wedge \text{exerciceExist}(e) \}$$

$$\text{escolheExercicio} :: s \sim \text{SessionID} \rightarrow e \sim \text{Exercicio} \rightarrow 1$$

$$\{ \}$$

Tem-se ainda a escolha do concurso para um grupo que está já registado.

$$\{ \text{existSession}(s) \wedge \text{existContest}(c) \wedge \text{userRegistadoNoContest}(s, c) \}$$

$$\text{escolheConcursoJaRegistado} :: s \sim \text{SessionID} \rightarrow c \sim \text{Contest} \rightarrow 1$$

$$\{ \}$$

Apresenta-se de seguida o contracto da função que gera um relatório ao concorrente.

$$\{ \}$$

$$\text{geraReport} :: e \sim \text{Exercicio} \rightarrow res \sim \text{Resolucao} \rightarrow \text{Report}$$

$$\{ \}$$

A título de exemplo, da potencialidade deste tipo de modelação, serviu-se agora da linguagem do Haskell para especificar a definição desta operação em maior detalhe, como se pode ver a partir das seguintes definições:

$$\text{geraReportBugCompile} :: \text{Exercicio} \rightarrow \text{Error} \rightarrow \text{Report}$$

$$\text{geraReportBugCompare} :: \text{Exercicio} \rightarrow \text{Errado} \rightarrow \text{Report}$$

$$\text{geraReportNoBug} :: \text{Exercicio} \rightarrow \text{Resolucao} \rightarrow \text{Report}$$

$$\text{execute} :: \text{Program} \rightarrow \text{Exercicio} \rightarrow \text{ResolucaoProposta}$$

A *ResolucaoProposta* é a resolução submetida pelos concorrentes e aprenstenta o seguinte tipo:  
 $\text{data ResolucaoProposta} = \text{Dict Input Output}$

Tem-se ainda a função que recebe uma resolução como input e devolve ora um programa pronto já compilado, ora um erro no caso de surgir algum problema na compilação.

$$\{ \}$$

$$\text{compile} :: \text{Resolucao} \rightarrow \text{Error} + \text{Program}$$

$$\{ \}$$

E ainda uma função de comparação que recebe uma resolução e um exercicio e devolve se o Output pretendido é o mesmo que o que o programa submetido origina.

$$\{ \text{length}(\text{Exercicio}) == \text{length}(\text{ResolucaoProposta}) \}$$

$$\text{compare} :: \text{Exercicio} \rightarrow \text{ResolucaoProposta} \rightarrow \text{Certo} + \text{Errado}$$

$$\{ \}$$


---

```

1 geraReport :: Exercicio -> Resolucao -> Report
2 geraReport exer res = do
3     case compile res of
4         (Left error) -> geraReportBugCompile error res

```

---

```

5         (Right p) ->
6             let resProps = execute p exer
7             in case (compare exer resProps) of
8                 (Left certo) -> geraReportNoBug e res
9                 (Right errado) -> geraReportBugCompare errado res

```

---

Se fosse pretendido este código de um ponto de vista de composição de funções a sua conversão poderia ser facilmente atingida pelas seguintes definições:

---

```

1 geraReport exer res =
2     compile res >>= \p -> compare exer (execute p exer)
3     >>= \c -> geraReportNoBug exer res

```

---

O contrato da função que gera o relatório final baseando-se na participação de uma equipa num determinado concurso:

$$\{existSession(s) \wedge existContest(c)\}$$

$$geraFinalReport :: s \sim SessionID \rightarrow c \sim Contest \rightarrow DictExercicioResolucao \rightarrow Report$$

$$\{\}$$

## 2.3 Modelo de Dados

Definiu-se que existirão três tipos de utilizadores: o administrador, o docente e o grupo.

- Administrador - é a entidade com mais poder no sistema. É o único que pode criar contas do tipo docente. É caracterizado por:
  - Nome de utilizador;
  - Nome completo;
  - Password
  - e-Mail
- Docente - entidade que tem permissões para criar concursos, exercícios, aceder aos resultados das submissões, (...). Os seus atributos coincidem com os do Administrador.
- Grupo - entidade que pode registar-se em concursos e submeter tentativas para os seus diferentes enunciados.

Decidiu-se que o sistema terá a noção de grupo, e um grupo não é mais do que um conjunto de concorrentes. No entanto, o grupo é que possui as credenciais para entrar no sistema (nome de utilizador e password). Além disso também tem um nome pelo qual é identificado, um e-mail que será usado no caso de haver necessidade de se entrar em contacto com o grupo, e um conjunto de .

É importante incluir a informação de cada concorrente no grupo para, se possível, automatizar várias tarefas tais como lançamento de notas. Cada concorrente é caracterizado pelo seu nome completo, número de aluno (se for o caso), e e-mail.

Para finalizar, pretende-se explicar em que consistem os concursos, enunciados e tentativas.

Um concurso, resumidamente, é um agregado de enunciados. Contém outras propriedades, tais como um título, data de início e data de fim (período em que o concurso está disponível para que os grupos se registem), chave de acesso (necessária para o registo dos grupos), duração do concurso (tempo que o grupo tem para resolver os exercícios do concurso, a partir do momento que dá início à prova), e por fim, regras de classificação.

Um enunciado é um exercício que o concorrente tenta solucionar. Como seria de esperar, cada exercício pode ter uma cotação diferente, logo o peso do enunciado é guardado no mesmo. Para cada enunciado existe também um conjunto de inputs e outputs, que servirão para verificar se o

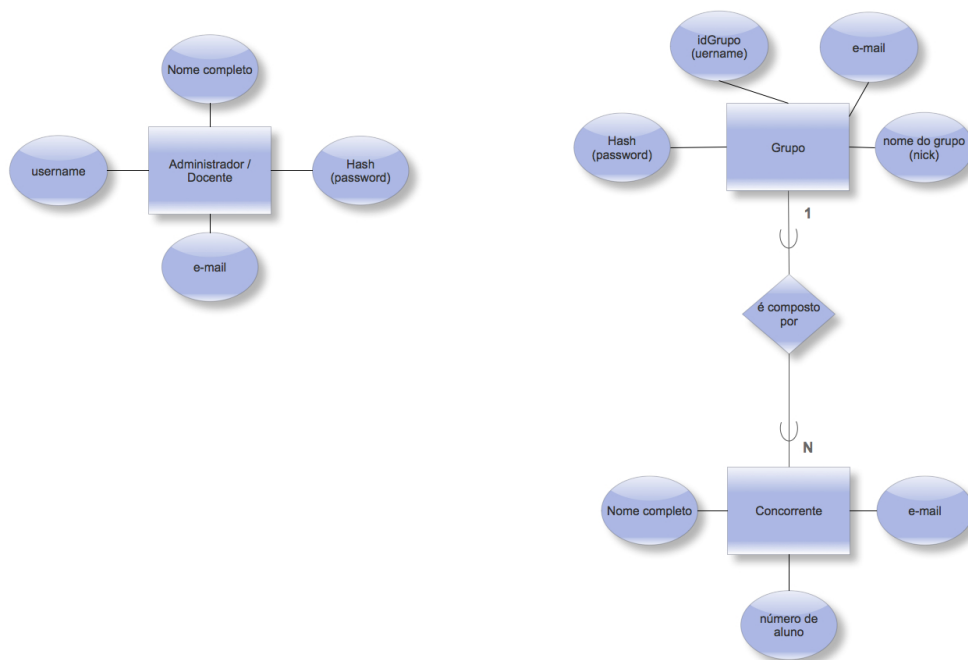


Figura 2.2: Modelo de dados - Grupo e Docente/Admin

programa submetido está correcto. Contém ainda uma descrição do problema que o concorrente deve resolver, assim como uma função de avaliação, função esta que define como se verifica se o output obtido está de acordo com o esperado.

Uma tentativa é a informação que é gerada pelo sistema, para cada vez que o grupo submete um ficheiro.

Além de conter dados sobre o concurso, enunciado e grupo a que pertence, a tentativa também contém o caminho para o código fonte do programa submetido, data e hora da tentativa, dados referentes à compilação e um dicionário com os inputs esperados e os outputs gerados pelo programa submetido.

## 2.4 Importação de dados

Uma das funcionalidades requeridas ao sistema proposto é a importação de enunciados e tentativas no formato xml. Esta funcionalidade será bastante útil para demonstrar e testar o sistema, sem que se tenha de criar manualmente os enunciados usando a interface gráfica, ou se tenha que submeter ficheiros com código fonte, de modo a serem geradas tentativas. Os campos presentes no xml de cada uma das entidades, *enunciado* e *tentativa*, são praticamente os mesmos que estão descritos no modelo de dados das respectivas entidades.

No xml do enunciado, não há nada muito relevante a acrescentar, além de que não contém um id para o enunciado, pois este será gerada automaticamente pelo sistema. Passamos agora a apresentar um exemplo do mesmo.

```

1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <Enunciado xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:noNamespaceSchemaLocation="enunciado.xsd">

```



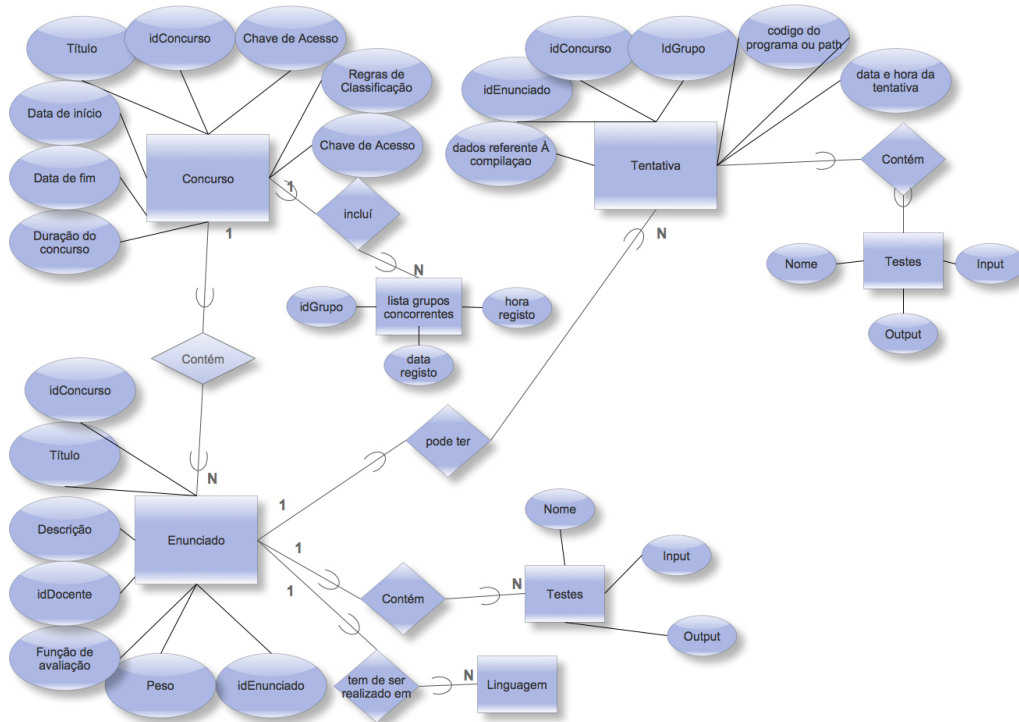


Figura 2.3: Modelo de dados - Concurso, tentativa e enunciado

```

5 <idConcurso> 1 </idConcurso>
6 <Peso>20</Peso>
7 <Titulo> Exercício 1 </Titulo>
8 <Descricao> Some os numeros que lhe sao passados como argumento , e
   apresente o resultado. </Descricao>
9 <Exemplo>Input: 1 1 1 1 1   Output: 5</Exemplo>
10 <Docente> PRH </Docente>
11 <FuncAval>Diff </FuncAval>
12 <Linguagens>
13   <Linguagem>C</Linguagem>
14 </Linguagens>
15 <Dict>
16   <Teste>
17     <Nome>Lista vazia</Nome>
18     <Input></Input>
19     <Output>0</Output>
20   </Teste>
21   <Teste>
22     <Nome>Lista c/ 1 elem</Nome>
23     <Input>1</Input>
24     <Output>1</Output>
25   </Teste>
26   <Teste>
27     <Nome>Lista c/ varios elem</Nome>
28     <Input> 2 3 4 5 </Input>
29     <Output>14</Output>
30   </Teste>
31 </Dict>
32
33 </Enunciado>

```

Quanto ao xml para a *tentativa* há que realçar o facto de que o código fonte do programa vai dentro de uma tag xml. Além da tag xml, o código fonte terá de ir cercado de uma secção CDATA. Isto acontece para que o código fonte não seja processado com o restante xml que o contém.

---

```

1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <Enunciado xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:noNamespaceSchemaLocation="tentativa.xsd">
5   <idConcurso>1</idConcurso>
6   <idEnunciado>1</idEnunciado>
7   <idGrupo>36</idGrupo>
8
9   <data>2010-12-08</data>
10  <hora>16:33:00</hora>
11
12  <compilou>1</compilou>
13
14  <Dict>
15    <Teste>
16      <Nome>Lista vazia</Nome>
17      <Input></Input>
18      <Output>0</Output>
19    </Teste>
20    <Teste>
21      <Nome>Lista c/ 1 elem</Nome>
22      <Input>1</Input>
23      <Output>1</Output>
24    </Teste>
25    <Teste>
26      <Nome>Lista c/ varios elem</Nome>
27      <Input> 2 3 4 5 </Input>
28      <Output>14</Output>
29    </Teste>
30  </Dict>
31
32  <pathMetricas>sasas</pathMetricas>
33
34  <codigoFonte>
35    <nome>prog.c</nome>
36    <codigo>
37      <![CDATA[
38        #include <stdio.h>
39
40        ... restante codigo ...
41
42      ]]>
43    </codigo>
44  </codigoFonte>
45
46 </Enunciado>

```

---

Para que todos os dados contidos nos ficheiros xml possam ser facilmente validados, foram criados dois *XML schema*. Neste schema, definiu-se quais as tags que devem existir em cada xml, o tipo de dados e até a gama de valores que serão contido por cada tag e a multiplicidade das tags.

Nesta fase inicial do projecto, ainda não foram sempre especificados os tipos de dados que serão contidos por cada tag.

No entanto, para alguns dos casos em que tal aconteceu apresenta-se alguns exemplos e explicações.

No xsd referente ao *enunciado* encontra-se o elemento *Peso*, que é um exemplo de uma tag que contém restrições. O *Peso* terá de ser um inteiro e terá um valor entre 0 e 100.

---

```

1 <ed:element name="Peso" default="25">
2   <ed:simpleType>
3     <ed:restriction base="ed:integer">
4       <ed:minInclusive value="0"/>
5       <ed:maxInclusive value="100"/>
6     </ed:restriction>
7   </ed:simpleType>
8 </ed:element>

```

---

Já o elemento *Linguagem* é também restringido, mas de uma forma ligeiramente diferente. A *Linguagem* será uma string, mas apenas poderá tomar um dos valores enumerados no xsd.

---

```

1 <ed:element name="Linguagem" maxOccurs="unbounded">
2   <ed:simpleType>
3     <ed:restriction base="ed:string">
4       <ed:enumeration value="C"/>
5       <ed:enumeration value="Java"/>
6       <ed:enumeration value="Haskell"/>
7     </ed:restriction>
8   </ed:simpleType>
9 </ed:element>

```

---

No xsd para a *tentativa* podemos evidenciar a multiplicidade das tags, ou seja, quantas vezes algumas delas se podem repetir. Na *tentativa*, existe um *Dict*, que contém uma ou mais tags *Teste*. Para definir que possam existir mais de que uma tag *Teste* dentro de *Dict*, adicionou-se o atributo *maxOccurs*, na entidade *Teste*, com o valor “unbounded”. O valor mínimo não é necessário definir, porque é um por default.

---

```

1 <tt:element name="Dict">
2   <tt:complexType>
3     <tt:sequence>
4       <tt:element name="Teste" maxOccurs="unbounded">
5         <tt:complexType>
6           <tt:sequence>
7             <tt:element name="Nome" type="tt:string"/>
8             <tt:element name="Input" type="tt:string"/>
9             <tt:element name="Output" type="tt:string"/>
10          </tt:sequence>
11        </tt:complexType>
12      </tt:element>
13    </tt:sequence>
14  </tt:complexType>
15 </tt:element>

```

---

Para dar uma ideia mais geral sobre ambos os *xml schema* criados, em vez de se apresentar aqui ambos os ficheiros integralmente, expôr-se-á os diagramas que o programa *oxygen* constrói e coloca ao nosso dispor, pois decidiu-se que torna o entendimento do schema muito mais simples.

Desta forma, de seguida apresentam-se os diagramas para o enunciado e para a tentativa:

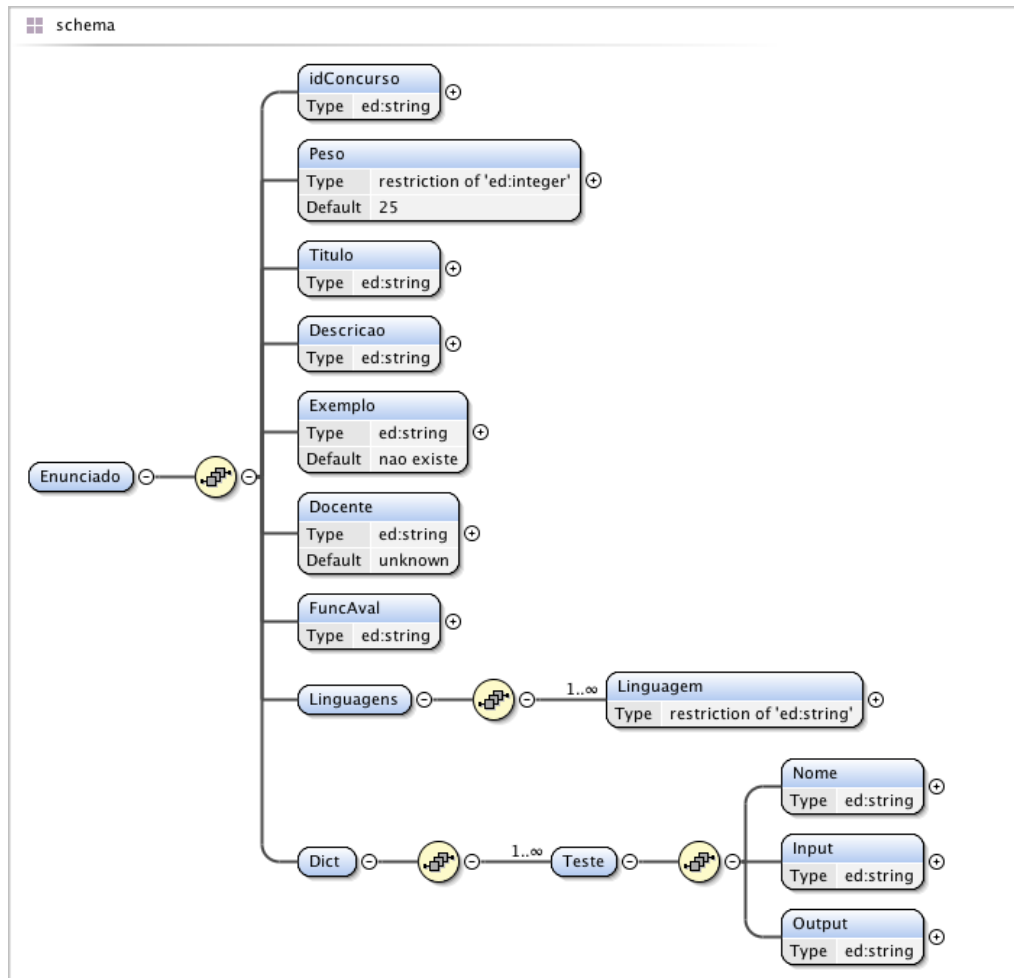


Figura 2.4: diagrama do schema para o enunciado

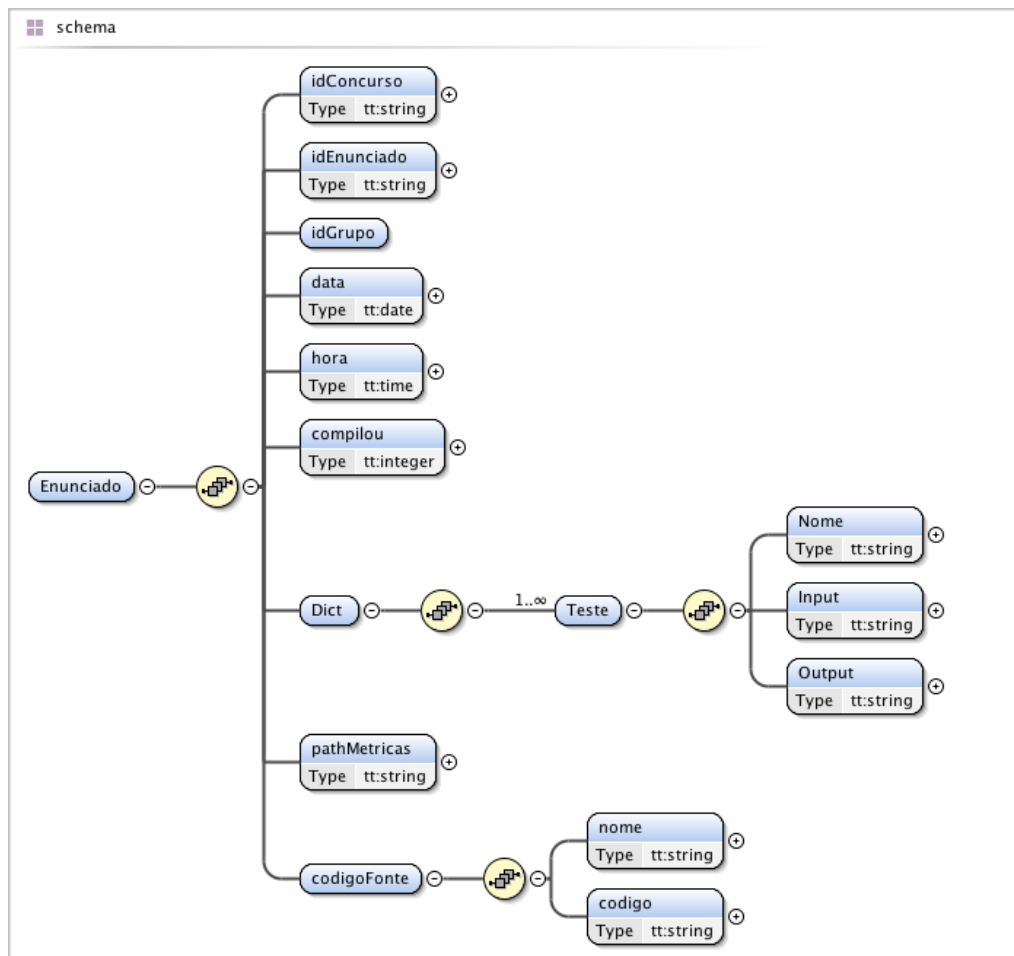


Figura 2.5: diagrama do schema para a tentativa

Parte II

Milestone II

# 3

## Implementação

### Conteúdo

<b>3.1 Criação de grupos, docentes e concorrentes</b>	<b>19</b>
<b>3.2 Linguagens de programação</b>	<b>19</b>
<b>3.3 Compilação</b>	<b>20</b>
<b>3.4 Execução</b>	<b>20</b>
<b>3.5 Guardar resultados</b>	<b>20</b>

Neste capítulo vamos expôr alguns pormenores relacionados com a implementação do problema proposto. Será explicada a maneira que encontramos para que seja cumprida a arquitectura que escolhemos e modelamos inicialmente.

### 3.1 Criação de grupos, docentes e concorrentes

O sistema permite que qualquer utilizador não registado se registre como grupo, e associe a si, um ou mais concorrentes. Este login é utilizado por todo o grupo, para participar nos mais variados concursos.

As contas de docente só podem ser criados pelo administrador. Para simplificar o trabalho do administrador, o docente pode criar uma conta de grupo, à qual mais tarde será concedida privilégios de docente.

### 3.2 Linguagens de programação

O nosso sistema é multilingue, ou seja, é possível submeter código fonte em várias linguagens de programação diferentes, desde que a linguagem tenha sido correctamente configurada por um docente. Cada linguagem é caracterizada por uma série de campos, os quais serão explicados de seguida:

- string de compilação: string que será executada quando se pretender compilar determinado código fonte. Esta string tem a particularidade de no lugar em que é suposto conter o nome do ficheiro a compilar, contém `#{file}`.  
Desta forma a string de compilação torna-se genérica, e independente do nome do ficheiro a compilar. exemplo: `gcc -O2 -Wall #{file}`
- string simples de execução: string utilizada para executar quando o código fonte foi compilado pela string de compilação.

exemplo 1:

- string de compilação: `gcc -O2 -Wall #{file}`
- string de execução respectiva: `./a.out`

exemplo 2:

- string de compilação: `gcc -O2 -Wall #{file} -o exec`
- string de execução respectiva: `./exec`

- string complexa de execução: a necessidade de uma segunda string de execução surgiu quando tentamos preparar o sistema para receber makefiles (inicialmente apenas para C). Nestes casos, o nome do executável gerado pela compilação não é conhecido à partida. Desta forma é necessário analisar o makefile, e só depois executar, tendo em conta a informação que retiramos do makefile.

Assim, e para a linguagem C, a string complexa de execução seria:

- `./#{file}`

em que `#{file}` representa o nome do executável.

### 3.3 Compilação

Estando as linguagens de programação correctamente configuradas, a compilação torna-se bastante simples. Quando uma tentativa é submetida no sistema, começamos por verificar se foi submetida apenas um ficheiro de código, ou um ficheiro comprimido.

Caso seja apenas um ficheiro, o sistema tenta compilar o código submetido, com a string de compilação da linguagem de programação em causa.

No caso de se tratar de um ficheiro comprimido, depois de o descomprimir, o sistema verifica se existe um makefile entre os ficheiros extraídos. Caso se verifique, é corrido o comando *make*, e tenta retirar o nome do executável gerado, de modo a poder ser usado na execução.

### 3.4 Execução

No fim da compilação, o sistema vai executar o programa uma vez para cada input. O processo de execução no caso de a compilação ter sido feita à custa do makefile, é feita usando a string complexa de execução (sendo o nome do executável aquele que foi retirado do makefile). Se tal não tiver acontecido, é usada a string simples.

A execução pode ser abortada se ultrapassar o tempo máximo de execução, que é definido aquando da criação do enunciado em questão.

### 3.5 Guardar resultados

Para cada input do enunciado em questão, o programa é executado uma vez. O seu output é comparado com o output esperado e é guardada uma entrada na base de dados com a percentagem de testes nos quais o programa teve sucesso.

No caso de o código não compilar, ou da execução do programa demorar mais tempo do que o máximo previsto pelo docente quando criou o enunciado, estas informações são também guardadas na base de dados.

Além de se guardarem todas as tentativas, a melhor é também guardada numa tabela à parte, para que o melhor resultado para cada enunciado seja de fácil acesso.



# 4

## Métricas

### Conteúdo

4.1	Análise Estática . . . . .	21
4.2	Análise Dinâmica . . . . .	22
4.3	Métricas de qualidade de <i>software</i> . . . . .	23
4.3.1	Patterns . . . . .	23

Existem diversas métricas, com objectivos diferentes, para analisar um projecto de *software*. Essas métricas podem ser vistas como diferentes 'lentes' com as quais olhamos para um software. Neste capítulo, pretende-se mostrar a investigação que foi realizada relativamente a este tema, começando primeiro por definir alguns conceitos e depois dividir as métricas por categorias. Cada categoria será estruturada da mesma maneira, indicada mais à frente.

Assim, encarou-se a análise a um software como sendo uma área que se divide em dois ramos, a **Análise Estática** e a **Análise Dinâmica**.

### 4.1 Análise Estática

A Análise Estática é olhar para um programa sob o ponto de vista do seu código ou ficheiro já compilado, e retirar conclusões sobre as suas características, sem nunca recorrer à sua execução ou análise de resultados da execução. Ainda relativamente à Análise Estática, temos essencialmente duas maneiras de olhar para o software. Podemos ver este tendo em conta a qualidade do ficheiro objecto produzido ( o código máquina que ira correr, p. ex: em java seria o *bytecode*) ou então tendo única e exclusivamente como objecto de observação os ficheiros de texto correspondentes ao código que compõe o programa. De notar que a Análise Estática é sempre relativa ao código do programa, ou seja, até mesmo uma análise que tenha em vista a qualidade do ficheiro objecto vai ser feita sobre o código do programa.

Assim sendo, no que diz respeito à qualidade do ficheiro objecto produzido, temos:

**Syntax checking** é um programa ou parte de um programa que tenta atestar a correcção da linguagem escrita.

**Type checking** é o processo de verificacao dos tipos de dados num software que visa garantir a restrição no que diz respeito aos tipos, implicando assim maior qualidade do software produzido

e menos probabilidade de acontecerem erros aquando da execução. Cada vez mais as linguagens recentes apresentam este tipo de sistemas, o que levam a que muitos dos erros ocorram em tempo de compilação, ou seja: completamente ainda a tempo de serem corrigidos pelos programadores, por exemplo: `Haskell`, `C++0x`, `JAVA6`. Estas linguagens apresentam um sistema de tipos forte o que garante este processo.

**Decompilation** é o processo de pegar num ficheiro objecto e tentar inferir ou descobrir o seu código fonte que o originou. Designa-se assim porque é o inverso do processo de compilação. Com a ajuda deste tipo de análise consegue-se obter, entre outras coisas, os algoritmos alto nível do código máquina em questão.

No que diz respeito à análise da qualidade do código como produto final temos as seguintes metodologias:

**Code metrics** é uma vasta área que se dedica a análise do código em si para tirar conclusões acerca da sua qualidade, estabilidade e manutenção.

**Style checking** funciona como uma análise para verificar determinadas regras que à partida se acreditam como boas na produção de código. Estas regras podem ser relativas a indentação, existência de ficheiros `README` e de documentação.

**Verification reverse engineering**, o método que serve para verificar se a implementação de um determinado sistema cumpre a sua especificação.

O objectivo deste trabalho é puramente analisar estaticamente um programa, relativamente às métricas de código e eventualmente relativamente ao estilo também. Mesmo assim este tema tão vasto deixou-nos com motivação para conhecer o que é este mundo da análise de software.

## 4.2 Análise Dinâmica

Outros tipos de análises existentes são as chamadas análises dinâmicas, estas pegam numa peça de *software* e não tendo em conta, nem se preocupando com o código que a constitui, executam simplesmente o programa e analisam exaustivamente sob vários prismas o seu comportamento. De seguida vamos dissertar sobre alguns destes métodos e práticas que existem para analisar a execução de um programa.

**Log analysis** é o método que consiste em pesquisar (automaticamente ou manualmente) os ficheiros de log produzidos por um determinado *software*, para perceber o que este está a fazer. Este tipo de análise muitas das vezes é feita a programas muito complexos e extensos que comunicam com o mundo real (rede, stdin, mundo IO). Um exemplo de quem faz este tipo de análise são os administradores de sistemas.

**Testing** é investigar o comportamento de um software através de uma bateria de testes que podem ter em consideração um determinado uso num caso que pode ser real. Geralmente, este tipo de análise simula os casos extremos a que o *software* pode ir, porque se acredita empiricamente que ao ter sucesso em situações extremas, há-de ter sucesso nos restantes casos. O que se pretende obter com este tipo de análise é o aumento na confiança de que o programa está a fazer o que é suposto, por parte de quem fabrica o produto.

**Debugging** é um método que ajuda as pessoas a terem conhecimento do que determinado *software* está a fazer. Este método geralmente é usado ainda numa fase inicial do produto, quando está a ser desenvolvido pelos programadores. É um bom método de detectar defeitos, falhas ou pequenos *bugs* no *software*.

**Instrumentation** é o método de monitorizar e medir o nível de performance de um determinado produto.

**Profiling** é a investigação sobre o comportamento de um programa aquando a sua execução, usando para isso informações do género recursos computacionais. Este tipo de análise é útil para por exemplo efectuar gestão de memória.

**Benchmarking** é o processo de comparar o processo do utilizador com os processos conhecidos de outros, de modo a obter conhecimento sobre as melhores práticas efectuadas na indústria.

## 4.3 Métricas de qualidade de *software*

A presença de testes num determinado programa de *software*, leva a que esse artefacto ganhe pontos no que diz respeito à análise estática sob o ponto de vista da qualidade, porque, como dissemos anteriormente, a presença de testes num projecto de *software* leva a que tenhamos mais confiança neste. Existem algumas fórmulas que nos dão alguns indicadores numéricos sobre o nível desta confiança. De seguida são apresentadas algumas sobre a cobertura de testes.

$$\text{Line Coverage} = \frac{\text{Nr of test lines}}{\text{nr of tested lines}}$$

$$\text{Decision coverage} = \frac{\text{Nr of test methods}}{\text{Sum of McCabe complexity}}$$

$$\text{Test granularity} = \frac{\text{Nr of test lines}}{\text{nr of tests}}$$

$$\text{Test efficiency} = \frac{\text{Decision coverage}}{\text{line coverage}}$$

Podemos sempre aumentar a nossa precisão na análise se considerarmos apenas as linhas que contêm código e não as linhas em branco ou linhas com apenas um carácter, como por exemplo as aberturas de blocos em **C**, **JAVA**. Ainda podemos também considerar não apenas o numero total de linhas mas também o número de métodos/funções testados.

De notar que as formulas atrás descritas podem ser modificadas para obter outros tipos de métricas, não só referentes a testes, como por exemplo:

$$\text{Code granularity} = \frac{\text{Nr of lines}}{\text{Nr of (methods/functions)}}$$

Como já referimos anteriormente, a análise que se pretende, por agora, é essencialmente estática. Assim sendo, segue-se uma lista de técnicas no que diz respeito à análise estática, estando estruturada da seguinte maneira: duas secções, uma para **Patterns** e outra para **Métricas de qualidade**

### 4.3.1 Patterns

# 5

## Aplicação Web

...

# 6

## Interface pelo terminal

### Conteúdo

<b>6.1</b>	<b>Perl</b> . . . . .	<b>25</b>
<b>6.2</b>	<b>Menus e exemplos</b> . . . . .	<b>25</b>

Tendo em vista a facilidade, para alguns utilizadores, em manusear um sistema por um terminal, decidiu-se criar uma interface para a aplicação desenvolvida. Esta interface, ainda em fase de desenvolvimento, vai permitir, essencialmente, trabalhar com a base de dados do sistema. Os objectivos passam por consultar listas de determinadas entidades, desde **enunciados** a utilizadores do sistema. De realçar que este modo de comunicação com o sistema apenas é utilizado pelos **administradores**

### 6.1 Perl

Como linguagem de desenvolvimento para esta interface, decidiu-se usar **Perl** devido à rapidez de implementação (visto que a criação de uma interface pelo terminal não constituía um dos principais objectivos) e a vasta diversificação de módulos existentes para auxílio ao desenvolvimento. Desses módulos, destaca-se o uso do módulo **DBIx::Class**, um módulo de comunicação a base de dados (apresentado durante as aulas de **EL::PLN**), que basicamente representa em classes cada tabela existente na base de dados, transformando também simples **queries** em métodos sobre as tabelas.

Tem-se em vista também a utilização de um módulo que use a biblioteca do sistema **Readline** e Falta: **readline**, **hash md5** para ser compatível com **ruby** ...

### 6.2 Menus e exemplos

Na fase actual desta interface, o utilizador desta interface terá pela frente um menu principal (Figura 6.1).

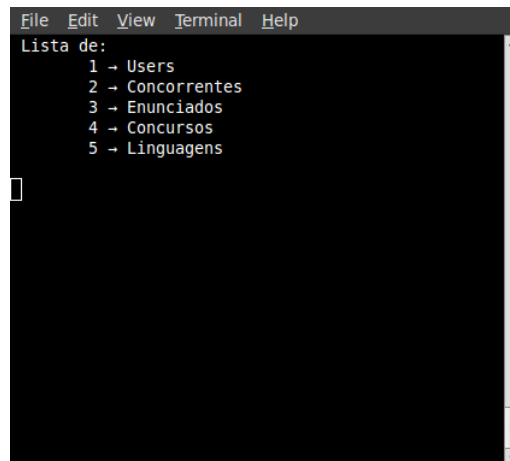


Figura 6.1: Menu Principal

Cada escolha desse menu representa as principais acções que se podem efectuar com uma base de dados:

- A listagem de elementos
- A procura de certos elementos e posterior actualização dos mesmos
- A inserção de novos elementos

Assim, caso o utilizador escolha a primeira opção, será levado para um novo menu contendo as 5 principais entidades deste sistema: Os **Users**; os **Enunciados**; as **Concorrentes**; os **Concursos**; e as **Linguagens** disponíveis para responder em cada concurso. A título de exemplo, caso o utilizador quisesse saber as linguagens disponíveis, a informação seria apresentada da seguinte forma(Figura 6.2):

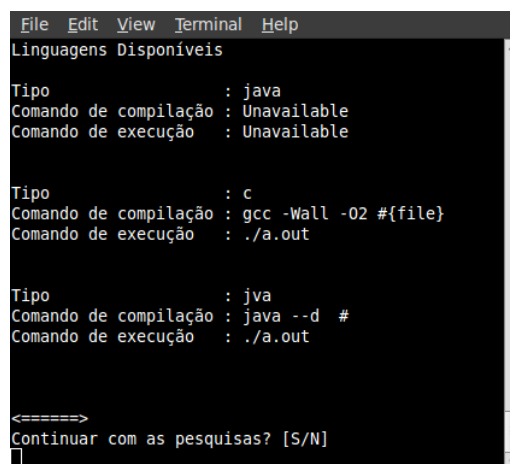
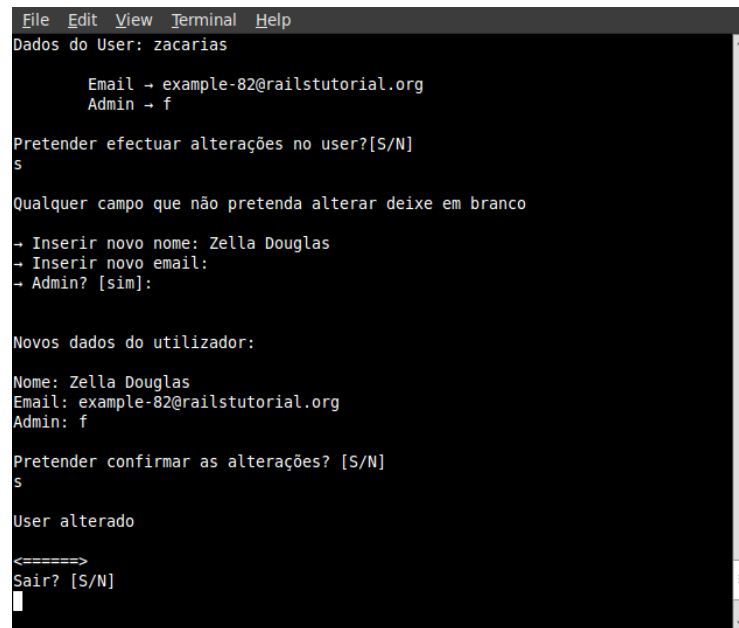


Figura 6.2: Linguagens disponíveis

O utilizador também poderia procurar por um único elemento. Como se pode ver na Figura 6.3, dando um nome de utilizador, seria retornada a informação sobre esse sujeito. Caso pretendesse, o utilizador poderia posteriormente proceder à alteração do mesmo.



```
File Edit View Terminal Help
Dados do User: zacarias

      Email → example-82@railstutorial.org
      Admin → f

Pretender efectuar alterações no user?[S/N]
s

Qualquer campo que não pretenda alterar deixe em branco

→ Inserir novo nome: Zella Douglas
→ Inserir novo email:
→ Admin? [sim]:

Novos dados do utilizador:
Nome: Zella Douglas
Email: example-82@railstutorial.org
Admin: f

Pretender confirmar as alterações? [S/N]
s

User alterado

<=====>
Sair? [S/N]
█
```

Figura 6.3: Procura e alteração do user Zella Douglas

# 7

## Scripts de avaliação



# 8

Front-end

# 9

## Conclusão e Trabalho Futuro

Ao longo de toda primeira fase modelamos os vários aspectos do nosso sistema. Toda a modelação do sistema realizada, é importante, devido à visão mais alargada que nos deu do problema e da sua resolução. Logo será muito importante também para que na fase de implementação tudo corra da melhor maneira e não seja necessário repensar grandes componentes do sistema.

A adicionar à modelação, foram já dados uns primeiros passos no que toca à importação de enunciados e tentativas em formato xml.

Definimos o conteúdo de cada xml, e criamos um schema para cada um deles, de modo a poder ser posteriormente usado na validação de dados.

Pensamos que com o trabalho realizado nesta fase, o projecto está bem encaminhado, e passaremos à implementação do sistema logo que possível.