Universidade do Minho
Departamento de Informática

Engenharia de Linguagens

Projecto Integrado

Software para Análise e Avaliação de Programas

Grupo 2:

José Pedro Silva
Mário Ulisses Costa
Pedro Faria

Braga, 11 de Dezembro de 2010

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Este relatório descreve 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, como de 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 para nós 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 que já conheciamos e aprendemos durante a nossa formação académica ou à parte desta. Um dos nossos grandes objectivos pessoais é consolidar e desenvolver ainda mais o uso de tecnologias variadas, para arranjar uma solução elegante, funcional e que cumpra os requesitos do sistema descrito. Assim, este projecto é mais do que mais um projecto de mestrado, passando a ser encarado como um desafio ao conhecimento e a por em prática conhecimentos adquiridos.

1.2.1 Ferramentas

As ferramentas/tecnologias que pensamos ir precisar são as seguinte:

- DB2
- Perl
- Ruby on Rails
- Haskell

Iremos usar 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 uteis 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 Contextualização

1.3.1 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 a 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.

1.3.2 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.

1.3.3 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

Estrutura do Relatório...

Capítulo 2

Modelação do Problema

Contextualização do cap 3

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 dispóniveis 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.

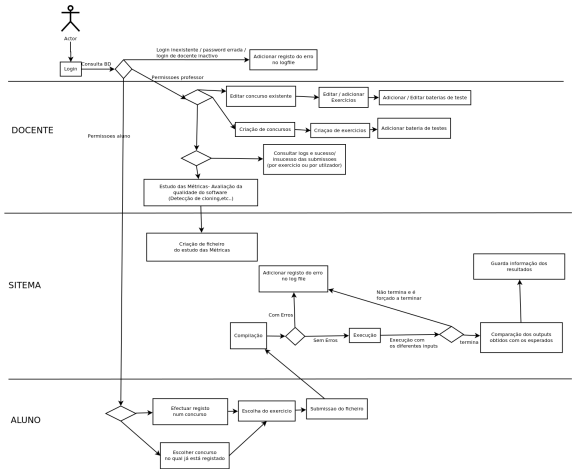

Figura 2.1: Arquitectura do sistema

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. Se o login não foi efectuado com sucesso, é adicionado no log file uma entrada com a descrição do erro. No caso de o login ser efectuado com sucesso, consoante as permissões do utilizador em questão, tem diferentes opções ao seu dispôr.

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 concurso, 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 compararar 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 nos oferecem.

Assim sendo, temos descritos os contratos da seguinte forma:

$$\begin{gathered}\{P\}\\ C\\ \{R\}\end{gathered}$$

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 teem de ser elementos booleanos e devem-se referir à assinatura da função. Assim sendo estamos a definir 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 que difinimos 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

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

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

Decidimos, 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. Achamos que especificar isso não iria trazer nada de interessante ao que pretendemos mostrar aqui. Assim, neste modelo formal apenas se vê de forma clara, os contratos que queremos que o nosso sitema tenha.

Começamos então por definir o contrato da função *login* que permite a um determinado utilizador entrar no sistema. Este 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 provacidade sobre os dados criticos do utilizador, como a password, decidimos 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.

$\{existsInDatabase(u)\}$
$login :: u \sim Username \times Hash \to SessionID \to Error + SessionID$
$\{\}$

Aprenta-se de seguida o modelo de dados formal que o sistema usa. Consideramos 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 exercicios.

$data\ Dict\ a\ b = (a \times b)^*$
$data\ Exercicio = Exercicio\ Enunciado\ (Dict\ Input\ Output)$
$data\ Contest = Contest\ Nome\ Tipo\ Exercicio^*$

Para criar um novo concurso, temos de assegurar que o utilizador que requisita este serviço é um professor, visto não nos interessar que alunos criem concursos. Temos ainda de assegurar que o concurso que se vai criar tem no minimo um exercicio.

$\{existeSession(s) \wedge isProf(s) \wedge (notEmpty \circ getExercice)\ c\}$
$createContest :: s \sim SessionID \to c \sim Contest \to 1$
$\{(notEmpty \circ getDict)\ c\}$

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

$\{existeSession(s) \wedge isProf(s) \wedge (not \circ exist)(Exercicioed)\}$
$createExercice :: s \sim SessionID \to e \sim Enunciado \to d \sim (Dictab) \to 1$
$\{exerciceCreated(Exercicioed)\}$

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

$\{existeSession(s) \wedge isProf(s) \wedge contestIsClosed(c)\}$
$consultarLogsContest :: s \sim SessionID \to c \sim Contest \to LogsContest$
$\{\}$

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

$\{existSession(s) \wedge contestNotFull(c)\}$
$registerOnContest :: s \sim SessionID \to c \sim Contest \to Credenciais$
$\{\}$

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

$\{existeSession(s) \wedge exerciceExist(e)\}$
$submitExercicio :: s \sim SessionID \to e \sim Exercicio \to res \sim Resolucao \to rep \sim Report$
$\{rep = geraReportseres\}$

Modelação de acções de selecção

Temos acções do nosso sistema que envolvem a selecção de items nos forms ou então métodos que geram efeitos secundários, assim decidimos explicar aqui esse conjunto de contractos. Estes métodos são interessantes de modelar porque, assim fica mais claro ver os parametros que recebemos para os concretizar.

Temos então a acção de escolha de um exercicio numa panoplia de exercicios disponiveis no concurso que o utilizador está actualmente inscrito e a participar.
Relembramos que o uso do tipo 1 significa o tipo unitário, ou seja, queremos denotar que do ponto de vista formal esta operação não devolve nada, apenas altera o sistema. Sistema esse que no inicio explicamos que por motivos de complexidade não tinha interesse expecificar formalmente.

$\{existeSession(s) \wedge exerciceExist(e)\}$
$escolheExercicio :: s \sim SessionID \rightarrow e \sim Exercicio \rightarrow 1$
$\{\}$

Temos ainda a escolha do concurso para um grupo que está já registado.

$\{existSession(s) \wedge existContest(c) \wedge userRegistadoNoContest(s, c)\}$
$escolheConcursoJaRegistado :: s \sim SessionID \rightarrow c \sim Contest \rightarrow 1$
$\{\}$

Mostramos de seguida o contracto da função que gera um relatório ao concorrente.

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

A titulo de exemplo, da potencialidade deste tipo de modelação, servimo-nos agora da linguagem do Haskell para especificar a definição desta operação em maior detalhe. Temos então as seguintes definições:

$$\begin{aligned} geraReportBugCompile &:: Exercicio \rightarrow Error \rightarrow Report \\ geraReportBugCompare &:: Exercicio \rightarrow Errado \rightarrow Report \\ geraReportNoBug &:: Exercicio \rightarrow Resolucao \rightarrow Report \end{aligned}$$

$$execute :: Program \rightarrow Exercicio \rightarrow ResolucaoProposta$$

Temos ainda que *ResolucaoProposta* é a resolução submetida pelos concorrentes e aprenstenta o seguinte tipo:
$dataResolucaoProposta = DictInputOutput$

Temos 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.

$\{\}$
$compile :: Resolucao \rightarrow EitherErrorProgram$
$\{\}$

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.

$\{length(Exercicio) == length(ResolucaoProposta)\}$
$compare :: Exercicio \rightarrow ResolucaoProposta \rightarrow Certo + Errado$
$\{\}$

```
geraReport :: Exercicio -> Resolucao -> Report
geraReport e res = do
        case compile(res) of
                (Left error) -> geraReportBugCompile error res
                (Right p) ->
                        let resProps = execute p e
                        in case (compare e resProps) of
                                (Left certo) -> geraReportNoBug e res
                                (Right errado) -> geraReportBugCompare errado res
```

Temos ainda o contrato da função que gera o relatório final baseando-se na participação de uma equipa num determinado concurso.

$\{existSession(s) \wedge existContest(c) \wedge\}$
$geraFinalReport :: s \sim SessionID \rightarrow c \sim Contest \rightarrow DictExercicioResolucao \rightarrow Report$
$\{\}$

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 .

Achamos 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 vamos explicar em que consistem os concursos, enunciados e tentativas.
Um concurso, resumidamente, é um agregado de enunciados. Tem outras propriedades tais como um título, data de inicio 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 inicio à prova), e por fim, regras de classificação.

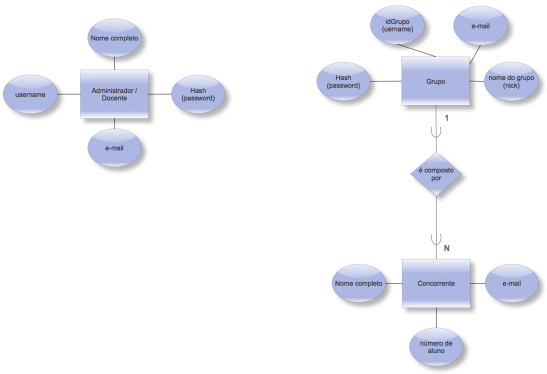

Figura 2.2: Modelo de dados - Grupo e Docente/Admin

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 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.

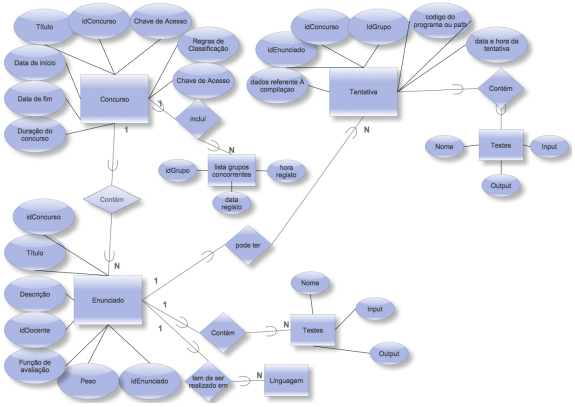

Figura 2.3: Modelo de dados - Concurso, tentativa e enunciado

2.4 Importação de dados

Uma das funcionalidades requeridas ao nosso sistema 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.

```
<?xml version="1.0" encoding="UTF-8"?>

<Enunciado xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="enunciado.xsd">
    <idConcurso> 1 </idConcurso>
    <Peso>20</Peso>
    <Titulo> Exercicio 1 </Titulo>
    <Descricao> Some os numeros que lhe sao passados como argumento, e
        apresente o resultado. </Descricao>
    <Exemplo>Input: 1 1 1 1 1   Output: 5</Exemplo>
    <Docente> PRH </Docente>
    <FuncAval>Diff</FuncAval>
    <Linguagens>
        <Linguagem>C</Linguagem>
    </Linguagens>
    <Dict>
        <Teste>
            <Nome>Lista vazia</Nome>
            <Input></Input>
            <Output>0</Output>
        </Teste>
        <Teste>
            <Nome>Lista c/ 1 elem</Nome>
            <Input>1</Input>
            <Output>1</Output>
        </Teste>
        <Teste>
            <Nome>Lista c/ varios elem</Nome>
            <Input> 2 3 4 5 </Input>
            <Output>14</Output>
        </Teste>
    </Dict>

</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 que o código fonte não seja processado com o restante xml que o contém.

```
<?xml version="1.0" encoding="UTF-8"?>

<Enunciado xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="tentativa.xsd">
    <idConcurso>1</idConcurso>
    <idEnunciado>1</idEnunciado>
    <idGrupo>36</idGrupo>

    <data>2010-12-08</data>
    <hora>16:33:00</hora>

    <compilou>1</compilou>

```

```
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 definimos 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 apresentaremos alguns exemplos e explicações.

No xsd referente ao *enunciado* encontramos 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">
```

```
    <ed:simpleType>
        <ed:restriction base="ed:string">
            <ed:enumeration value="C"/>
            <ed:enumeration value="Java"/>
            <ed:enumeration value="Haskell"/>
        </ed:restriction>
    </ed:simpleType>
</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 definirmos que possam existir mais de que uma tag *Teste* dentro de *Dict*, adicionamos o atributo *maxOccurs*, na entidade *Teste*, com o valor *"unbounded"*. O valor mínimo não é necessário definir, porque é um por default.

```
<tt:element name="Dict">
    <tt:complexType>
        <tt:sequence>
            <tt:element name="Teste" maxOccurs="unbounded">
                <tt:complexType>
                    <tt:sequence>
                        <tt:element name="Nome" type="tt:string"/>
                        <tt:element name="Input" type="tt:string"/>
                        <tt:element name="Output" type="tt:string"/>
                    </tt:sequence>
                </tt:complexType>
            </tt:element>
        </tt:sequence>
    </tt:complexType>
</tt:element>
```

Para dar uma ideia mais geral sobre ambos os *xml schema* criados, em vez de apresentarmos aqui ambos os ficheiros integralmente, vamos antes expor os diagramas que o programa *oxygen* constrói e coloca ao nosso dispor, pois pensamos que torna o entendimento do schema muito mais simples.

Desta forma aqui ficam os diagramas para o enunciado e para a tentativa:

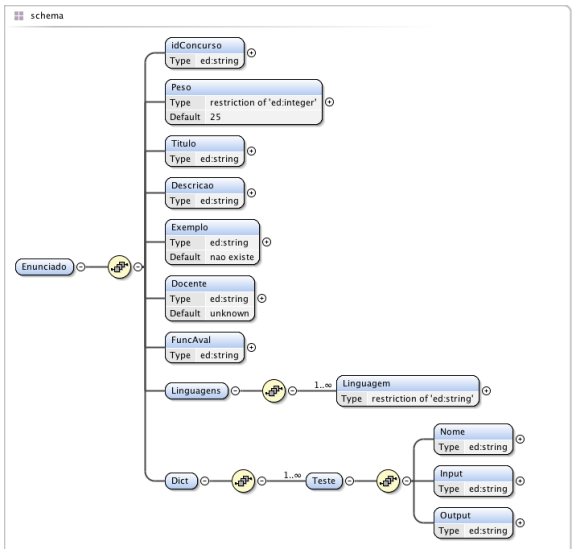

Figura 2.4: diagrama do schema para o enunciado

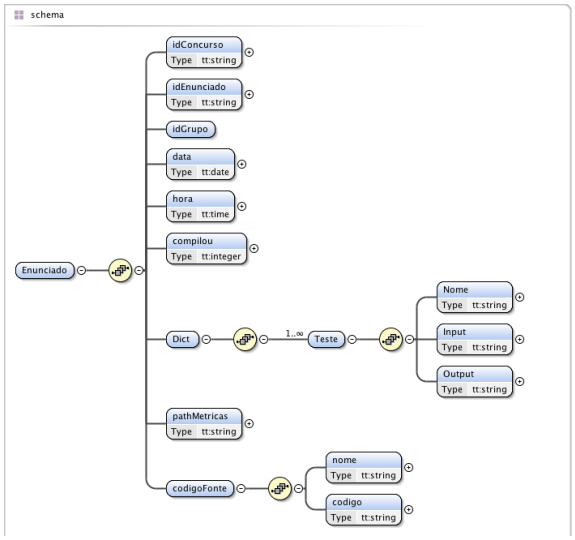

Figura 2.5: diagrama do schema para a tentativa

Capítulo 3

Conclusão e Trabalho Futuro

conclusão: curti buess isto da modelação :D quero fazer isto todos os dias 4ever

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 possivel.