Projecto Integrado

Universidade do Minho

Mestrado de Informática, Engenharia de Linguagens

Sérgio Areias pgº13381 Hugo Areias pgº11060

8 de Julho de 2009

Conteúdo

1 Introdução

Este projecto aparece no seguimento da Unidade Curricular de Engenharia de Linguagens do Mestrado de Informática, com o intuito de fazer uma aplicação abrangente de todos os conhecimentos que recebemos durante o ano lectivo e contribuir para a nossa capacidade de gerir um projecto em grupo.

Este foi implementado praticamente na sua totalidade com o recurso a duas ferramentas, sendo estas o *perl*[1] e o *ANTLR*[2]. Também tivemos que aplicar alguns conhecimentos de *SQL*[3] (no nosso caso *SQLite*[4]) e de *UML*[5], o que dá para ter uma noção do quanto abrangente era este projecto integrado.

O DRAW (Desenvolvimento Rápido de Aplicações *WEB*) é uma ferramenta criada com o intuito de facilitar o desenvolvimento de aplicações *WEB*, tal como o seu nome indica, de uma forma rápida e prática. O DRAW suporta algumas ferramentas e linguagens e é possível que venha a suportar ainda mais no futuro. Através da ferramenta DREQL é possível atingir níveis de transparência importantes que facilitam as funções do utilizador quando existe a necessidade de manipular e aceder a motores de base de dados, isto porque as rotinas de acesso à base de dados são automaticamente geradas com base nos diagramas de classe fornecidos pelo utilizador. Esses diagramas de classe são convertidos para o esquema relacional e a partir desse esquema são criadas as rotinas com as devidas operações e validações.

Esta ferramenta também disponibiliza um variado e precioso suporte a *CSS* para fornecer ao utilizador um elevado nível de controlo e liberdade quando o que se pretende é personalizar o resultado final da aplicação. Este suporte é feito através de classes *CSS*.

Objectos também não foram esquecidos e encontram-se implementados na última versão do DRAW, na qual é possível declarar objectos, invocar métodos e armazenar o estado final do objecto através do auxílio de base de dados dedicadas.

O DRAW é uma ferramenta recente, mas já com algumas potencialidades, espera-se que nas próximas versões o número de objectos e motores de base de dados suportados aumente e que algumas limitações sejam ultrapassadas.

[1] *http://www.perl.org/*
[2] *http://www.antlr.org/*
[3] *http://en.wikipedia.org/wiki/SQL*
[4] *http://www.sqlite.org/*
[5] *http://www.uml.org/*

2 Descrição do Problema

2.1 Breve Introdução ao Problema

Este projecto foi pensado para ser dividido em três grandes camadas, o *Draw*, o *dreql* e o *middleware*. Três camadas que abrangem a matéria estudada durante o ano lectivo.

O *dreql* é suposto ser uma linguagem que permite a descrição da estrutura de dados de uma aplicação. Esta linguagem tinha alguns requisitos mínimos mas, de resto foi dada a liberdade a cada grupo de a desenvolver da forma que mais lhe convinha desde que se mantivesse dentro dos limites do tema. Também é suposto fazer a conversão do modelo de classes para o modelo relacional de forma a ser possível criarmos o código para a base de dados, sendo a tecnologia desta também algo que cada grupo poderia escolher. Além disto é suposto o *dreql* ter código associado para gerar as funções em *perl* que implementam as funções do *CRUD*.

O *middleware* foi pensado para ser a camada que "cola"o *dreql* com o *Draw* de forma a no fim termos as três camadas a funcionar em conjunto. Temos total liberdade na implementação desta camada desde que resolva o que é pedido. Na aula o nosso grupo propôs a criação de um módulo em *perl* para resolver o problema mas, esta é só uma das soluções possíveis.

Finalmente o *Draw* que é a ferramenta a desenvolver para permitir o desenvolvimento automatizado de aplicações (não é totalmente automatizado mas vamos tentar que seja o máximo possível). O objectivo desta ferramenta é encaixar no grupo de ferramentas que se baseia num autómato reactivo e que segue a filosofia MVC (Modelo, Vista, Controlador).

2.2 Objectivos a Cumprir

Os requisitos do *dreql*, de uma forma mais objectiva e precisa, são os seguintes:

- "A linguagem deve permitir definir os Tipos Atómicos (Primitivos, Renomeados, Enumerados)";
- "Tipos Estruturados (Set, Seq, Tree, Tuple e Map)";
- "Relações implícita ou explicitamente";
- "Alguns atributos dos campos devem ser especificados (editável, nulo, visível, comprimento máximo, mascarado, chave, auto-increment)";
- "Gerador de Código associado à dreqL para realizar as funções de CRUD".

Quanto ao *Draw* são os seguintes:

- "Implementação de uma linguagem básica para a representação de grafos (por exemplo, inspirada na do graphviz), e uma ferramenta que gere um conjunto de CGI capazes de navegar sobre este grafo";
- "Especificação de um conjunto base de tipos (STR, INT, TEXT, FLOAT, DATE, PASSWORD, EMAIL, URL). Adição sobre a linguagem da etapa anterior de parâmetros

em cada aresta. Cada estado (CGI) deve mostrar um formulário por acção (aresta) existente. Por exemplo, se do vértice A partem duas arestas, uma com os campos Nome: STR, Morada: TEXT, e uma outra com os campos Nome: STR, Pass: PASSWORD, a CGI referente ao vértice A deverá apresentar dois formulários, um para cada uma das possíveis acções. Depois de escolhida uma acção, o estado de destino dessa aresta deverá imprimir os valores preenchidos no formulário anterior";

- "Adição de funções de processamento nas arestas do grafo. Estas funções estarão presentes num outro documento Perl (por exemplo num módulo Perl) e serão invocadas sempre que necessário. Assim, cada transição entre estados deixa de conter apenas os parâmetros, mas sim uma assinatura de uma função: nome da função, parâmetros necessários, e o tipo de retorno da função. Quando um utilizador utilizar um formulário, os dados serão usados para invocar a função, e o resultado final será mostrado. Por exemplo: soma(a: INT, b:INT): INT";

- "Não há garantias que uma função invocada durante a transição entre estados não dê erro. Ou seja, uma função de inserir que, teoricamente, não devolve nada (apenas altera o estado na base de dados) pode dar um erro. Como esta função, qualquer outra pode dar erro. Assim, implicitamente o DRAW deve considerar que todas as funções retornam um "maybe": o tipo especificado se não der erro, ou undef no caso de ter ocorrido um erro. Em caso de erro, a variável especial do Perl $@ será usada para guardar a mensagem de erro. O nodo de destino deverá verificar o retorno da função invocada. Se o valor for undef, a mensagem de erro é mostrada. Caso contrário, é mostrado o resultado de invocar a função";

- "Uma função não retorna necessariamente um tipo escalar. Por exemplo, uma pesquisa numa base de dados retorna uma lista de registos. Torna-se, pois, imprescindível adicionar tipos estruturados à linguagem DRAW:"

  - "Tuplo: uma sequência de tipos (básicos, para já!). Por exemplo: ProcuraRegisto(Nome: STR):(Nome: STR, Idade: INT, Morada: STR)";
  - "Lista: uma sequência homogénea de tipos (básicos ou tuplos). Por exemplo: Alunos():(Nome: STR, Idade: INT)*".

- "Permitir que em cada transição, para além da invocação de uma função, seja possível propagar variáveis entre estados. Deste modo será possível, por exemplo, que o username de um utilizador introduzido na página de login possa ser propagado por todas as restantes páginas. Note-se que as variáveis propagadas só são mostradas se fizerem parte de um formulário. Os formulários deverão ser preenchidos caso existam definidas variáveis com os nomes dos campos respectivos";

- "Permitir que as funções de transição retornem um tipo especial, chamado 'VARS' que corresponda a uma tabela associativa, que mapeie nomes de variáveis em valores. Isto permitirá, por exemplo, que depois de um formulário em que se peça a chave de uma tabela, seja possível mostrar o formulário para editar o registo da base de dados, preenchendo automaticamente o formulário com os respectivos valores".

3 Processo de Desenvolvimento

3.1 Dreql

Como já foi abordado acima, esta camada está estruturada em várias subcamadas, uma delas é o processador de uma linguagem, que faz a representação textual de um diagrama de classes (no nosso caso em UML), e conversor de um modelo de classes para um modelo relacional de forma ao resultado ser redireccionado para a segunda subcamada, que gera o código *SQL* para a criação das tabelas e também as funções de manipulação dos dados (estas em *perl*).

3.1.1 Camada das Classes

O primeiro passo do desenvolvimento, nesta fase, foi um processador para possível análise léxica (*ANTLR grammar*). Este processador foi desenvolvido tendo em conta todos os requisitos pedidos pelos professores durante o decorrer do ano lectivo.

A linguagem que desenvolvemos admite somente três grupos de comandos. Comandos para descrever as classes e o seu conteúdo, comandos para especificar as relações entres as várias classes e comandos para os tipos renomeados. Dentro do primeiro grupo de comandos temos várias opções para descrever o tipo de classe ou até propriedades que ela possa ter. Como podemos ver abaixo, permitimos a possibilidade de o utilizador descrever classes abstractas, classes que herdam conteúdo de outras classes ou as classes simples[6].

```
dclass  : ('class' | 'Class') table '(' fields ')'
          -> ^(CLASS table fields)
          | ('abstract' | 'Abstract') table '(' fields ')'
          -> ^(ABSTRACT table fields)
          | ('class' | 'Class') table ('extends' | 'Extends') table '(' fields ')'
          -> ^(EXTENDS table table fields)
        ;
```

Para especificar as relações entre as classes, só o podemos fazer de uma forma podendo unicamente variar as opções que o comando recebe. As opções incluem o nome das tabelas e a cardinalidade da relação que cada uma tem. Abaixo podemos ver todos os valores, que podemos usar nas relações, definidos na gramática e também um exemplo de uma relação.

```
renom   :       ('renom' | 'Renom') TYPE '=' TYPE
                -> ^(RENOM TYPE TYPE)
        ;

reltype : (a='1' | b='0..1' | c='1..*' | d='0..*' | e='*' | f=NUM | NUM '..' NUM)
                -> {a!=null}? ^(RELTYPE ONE)
                -> {b!=null}? ^(RELTYPE ZEROTOONE)
                -> {c!=null}? ^(RELTYPE ONETON)
                -> {d!=null}? ^(RELTYPE ZEROTON)
                -> {e!=null}? ^(RELTYPE NTOM)
                -> {f!=null}? ^(RELTYPE NUM)
```

[6] entende-se por simples todos os outros tipos de classes sem características especiais

```
                -> ^(RELTYPE NUM NUM)
        ;

Relation ( Registo, Recepcao: 1, *);
```

Como podemos ver as duas tabelas, Registo e Recepcao, têm definida uma relação entre elas em que a cardinalidade é de 1 para muitos. Para gerar o código *SQL*, subentende-se na relação que a tabela à esquerda é a que cede a chave primária para estrangeira na tabela à direita. Se a relação for *n* para *n* então as duas cedem a chave primária e uma nova tabela tem que ser criada, mas mais à frente neste relatório voltaremos a abordar este assunto com mais detalhe.

Por último, as renomeações funcionam de uma forma muito semelhante ao *typedef*[7] em *C*[8], e serve unicamente para facilitar a legibilidade do código.

```
renom    :      ('renom' | 'Renom') TYPE '=' TYPE
                -> ^(RENOM TYPE TYPE)
        ;
```

Para podermos definir as classes foi necessário tomar várias decisões, isto porque tinhamos que decidir que opções iríamos disponibilizar ao utilizador para enriquecer o seu modelo. Claro que quanto mais opções estivessem à disponibilidade do utilizador, mais rica teria que ser a nossa gramática.

Na definição dos campos de uma tabela, decidimos permitir várias formas de o fazer de forma à linguagem ser bastante flexível. Temos a possibilidade de ter simplesmente o nome do campo, ou então ter também o tipo de dados e/ou opções específicas sobre o mesmo. No caso em que temos só o nome do campo, usamos um tipo de dado predefenido facilitando imenso a escrita do modelo. Por exemplo, imaginemos um modelo em que a maior parte dos campos das classes são texto simples, isto implicaria adicionar o tipo texto[9] à definição de todos os campos. Esta opção na gramática permite então ao utilizador omitir o tipo texto e deixar que o nosso processador lhe atribua o tipo de dados predefinido.

```
field    :      fieldname ':' datatype '[' optional ']'
                -> ^(FIELD fieldname datatype optional)
                | fieldname ':'  '[' optional ']'
                -> ^(FIELD fieldname optional)
                | fieldname ':' datatype
                -> ^(FIELD fieldname datatype)
                | fieldname
                -> ^(FIELD fieldname)
        ;
```

Quanto aos tipos de dados somos bastante flexíveis, ja que só diferenciamos os tipos especiais, como *set* ou *enum* (podemos ver abaixo), tudo o resto é aceite como uma *string* e só é tratado posteriormente.

```
datatype        :       TYPE '(' NUM ')'
                -> ^(DATATYPE TYPE NUM)
```

[7] *http://en.wikipedia.org/wiki/Typedef*
[8] *http://www.open-std.org/jtc1/sc22/wg14/*
[9] texto é fictício e utilizado só como exemplo

```
                | TYPE
                -> ^(DATATYPE TYPE)
                | ('enum' | 'Enum') '(' enumopt ')'
                -> ^(DATATYPE ENUM enumopt)
                | set
                -> ^(DATATYPE set)
        ;
```

O tipo de dados *set* é tratado de forma especial já que tem variações na sua forma. Mais uma vez decidimos que o tipo de dados fosse omitido quando esse é do tipo de dados predefinido. Mais à frente vamos abordar as implicações que este tipo tem na conversão para o modelo relacional.

```
set     :       ('set' | 'Set') TYPE '(' NUM ')'
                -> ^(SETTYPE TYPE NUM)
                | ('set' | 'Set') TYPE
                -> ^(SETTYPE TYPE)
                | ('set' | 'Set')
                -> ^(SETTYPE SET)
        ;
```

As opções específicas que falamos em cima no momento de definir os campos, são as que estão descritas abaixo no pequeno extracto da gramática. Nenhuma destas opções foi escolhida especialmente por nós, já que estavam nos requisitos de avaliação da gramática[10].

```
option  :
   (a='editable' | b='visible' | c='optional' | d='masked' | e='key' | 'autoinc')
                -> {a!=null}? ^(OPTION EDITABLE)
                -> {b!=null}? ^(OPTION VISIBLE)
                -> {c!=null}? ^(OPTION OPTIONAL)
                -> {d!=null}? ^(OPTION MASKED)
                -> {e!=null}? ^(OPTION KEY)
                -> ^(OPTION AUTOINC)
        ;
```

O segundo passo resume-se à utilização das acções semânticas de forma a criar as várias estruturas que vão permitir gerarmos o modelo relacional e as funções de manipulação de dados. Não vamos aprofundar as acções semânticas, pelo menos não todas, já que não apresentam nada de novo.

Para guardar tudo o que é processado, criamos três estruturas que servem para armazenar as classes, as relações e as renomeações no código. Em baixo vamos mostrar a disposição do conteúdo das várias estruturas e explicar o que cada símbolo significa.

```
Renomeacoes {
            String => String,
            String => String,
            String => String,
                    ...
}
```

[10] A gramática na sua totalidade pode ser visualizada no anexo A.

Nas renomeações temos, do lado esquerdo, cada termo associado a um tipo de dados (do lado direito) conhecido pelo processador.

```
Relacoes {
      tabela 1 => {
                     {
                        tabela 2,
                        tipo relacao 1,
                        tipo relacao 2
                     },
                     {
                        tabela 3,
                        tipo relacao 1,
                        tipo relacao 3
                     },
                    ...
                  },
                 ...
}
```

Nas relações temos para cada tabela, todas as tabelas que com têm uma relação com a mesma e o tipo de relação que existe entre ambas. Todas as tabelas que se encontrem na entrada de uma tabela, têm que ceder as suas chaves primárias como chaves estrangeiras para a mesma.

Se tivermos em conta a representação acima, então temos que a *tabela 1* tem uma relação com a *tabela 2* e outra com a *tabela 3*. O tipo de relação entre elas sabe-se através do *tipo relação 1, 2 e 3* e também sabemos então que *tabela 2* e *tabela 3* vão ceder as chaves primárias como chaves estrangeiras da *tabela 1*.

```
Classes {
        classe => {
                    {
                       campos {
                                {
                                  campo,
                                  tipo,
                                  tamanho,
                                  opcoes {
                                            opcao 1,
                                            opcao 2,
                                              ...
                                         }
                                },
                               ...
                               },
                       chaves {
                                  chave 1,
                                  chave 2,
                                    ...
                                },
                       abstract,
                       extended
                    },
                  },
                 ...
}
```

A estrutura das classes já armazena uma quantidade de informação maior, tendo para cada classe a lista de todos os seus campos e chaves, uma opção para verificar se a classe é abstracta e outra para verificar se a classe herda informação de outra tabela. Cada campo tem associado o seu identificador, o tipo de dados, o tamanho e as opções para ter em conta no código *SQL*.

Ter estas estruturas ajudava imenso mas, para mais tarde gerar o código para a construção da base de dados precisavamos de algo mais, algo já com toda a informação necessária para evitar cálculos auxiliares. Para isso decidimos criar uma nova estrutura em que a informação era disposta como uma representação de um modelo relacional ao invés do modelo de classes (isto substitui as três estruturas que vimos acima).

Abaixo podemos ver a representação da nova estrutura, antes de começarmos a explicar como foram convertidos os valores das outras estruturas para esta.

```
Tabelas {
        tabela => {
                    {
                        campos {
                                  {
                                   campo,
                                   tipo,
                                   tamanho,
                                   opcoes {
                                             opcao 1,
                                             opcao 2,
                                               ...
                                           }
                                  },
                                 ...
                                },
                        chaves primarias {
                                           chave 1,
                                           chave 2,
                                             ...
                                          },
                        chaves estrangeiras {
                                              (chave 1,ref tabela),
                                              (chave 2,ref tabela),
                                                 ...
                                             },
                    },
                  },
                 ...
}
```

Para cada tabela temos então, todos os seus campos com as respectivas características, temos as chaves primárias e também as chaves estrangeiras com as referências para a tabela onde são chave primária.

Mas como fazer então a conversão? Sabemos que temos vários tipos de situações especiais que fazem com que o algoritmo não seja regular, por exemplo, se tivermos uma relação n para n implica que temos que criar uma nova tabela que partilha informação com

as tabelas presentes na relação. Todas estas situações tiveram que ser estudadas para não sermos surpreendidos na fase de testes.

Para começar, temos que percorrer todas as classes de forma a criar uma tabela para cada uma destas classes, excepto para as classes abstractas. Para cada classe vamos gerar o conteúdo das tabelas que resultam da mesma, este problema foi dividido em vários sub problemas e já vamos perceber porquê.

Primeiro começamos por gerar a informação dos campos da tabela que, como podemos ver abaixo, num primeiro passo copia os campos da classe para a tabela e num segundo passo verifica se a classe herda informação de outra e, se for o caso, carrega a nova classe e copia os seus campos para a tabela. Destes campos só são copiados aqueles que não tem o tipo de dados complexo *set* que já a seguir iremos estudar a solução para os mesmos.

```
for(Fields field : fields) {
   tablefields = fieldCode(field,tablefields);
}
if(cont.getExtends() != null) {
   String abstable = cont.getExtends();
   if(classes.containsKey(abstable)) {
      fields = classes.get(abstable).getFields();
      for(Fields field : fields) {
         tablefields = fieldCode(field,tablefields);
      }
   }
   else System.out.println("ERROR: Abstract class " + abstable
                              + " is not defined!!!\n");
}
tbcontent.setFields(tablefields);
```

Se algum dos campos da tabela tem o tipo de dados *set* então temos que criar uma nova tabela sendo o nome a combinação dos nomes envolvidos separados por um sublinhado[11]. Esta tabela vai ter como chaves, as chaves primárias da tabela a que pertence o campo em questão e o próprio campo. Como só temos os nomes das chaves temos que procurar os dados sobre a mesma de forma a podermos criar um *create* para esta tabela posteriormente. Nas chaves estrangeiras acontece o mesmo à excepção de guardarmos também a referência para a tabela.

```
if(data.matches("set")) {
   String newtable =  create + "_" + field.getName();
   newfields = fieldCodeSet(field,newfields);
   newprimkeys.add(field.getName());
   newprimkeys.addAll(keys);
   for(String key : keys) {
      Fields keyfield = searchField(fields,key);
      newfields = fieldCode(keyfield,newfields);
      newfgnkeys.add(new ObjPair(key,create));
   }
   tbcontent.setFields(newfields);
   tbcontent.setPrimKeys(newprimkeys);
   tbcontent.setFgnKeys(newfgnkeys);
   tablescont.put(newtable,tbcontent);
}
```

[11] sublinhado é o mesmo que o *underscore*

Depois de estes dois passos estarem concluídos só nos falta então gerar a informação para as chaves, tanto primárias como estrangeiras, mas para obter essa informação é necessário verificar se a tabela tem algum tipo de relação com outras tabelas. Se existe alguma relação e é do tipo muitos para muitos então é necessária a criação de uma nova tabela.

```
if((from.matches("ZeroToN") || from.matches("OneToN") || from.matches("N"))
   && (to.matches("ZeroToN") || to.matches("OneToN") || to.matches("N"))) {
    tablescont = createNewTable(classes,create,tablename,tablescont);
}
```

Essa nova tabela tem como nome a combinação dos nomes das duas tabelas. As chaves primárias é directo já que nas classes já as temos marcadas por isso é so copiar de ambas as tabelas, já nas chaves estrangeiras temos que percorrer as chaves uma a uma para guardar a referência para a tabela. Os campos da tabela são gerados através dos campos de ambas as tabelas, excepto os campos que contêm tipos de dados complexos que não são copiados.

```
String newtable = table + "_" + tablename;
newprimkeys.addAll(keys);
newprimkeys.addAll(keys2);
for(String key : keys) {
    Fields field = searchField(cont.getFields(), key);
    newfields = fieldCode(field,newfields);
    newfgnkeys.add(new ObjPair<String,String>(key,table));
}
for(String key : keys2) {
    Fields field = searchField(cont2.getFields(), key);
    newfields = fieldCode(field,newfields);
    newfgnkeys.add(new ObjPair<String,String>(key,tablename));
}
TableContent tbcontent = new TableContent();
tbcontent.setFields(newfields);
tbcontent.setPrimKeys(newprimkeys);
tbcontent.setFgnKeys(newfgnkeys);
tablescont.put(newtable,tbcontent);
```

Para as relações que não são de muitos para muitos basta carregar todas as chaves da tabela à direita na relação e guardá-las como chaves estrangeiras na tabela[12]. Ao guardar estas é também necessário guardar toda a informação do campo respectivo para poder gerar o código do *create*.

```
ArrayList<String> relkeys = classes.get(tablename).getKeys();
ArrayList<Fields> relfields = classes.get(tablename).getFields();

for(String key : relkeys) {
    Fields relfield = searchField(relfields,key);
    tbcontent.setFields(fieldCode(relfield,tbcontent.getFields()));
    if(keys.size() == 0) {
        primkeys.add(key);
    }
    fgnkeys.add(new ObjPair<String,String>(key,tablename));
}
```

[12]na linguagem é a tabela da esquerda que cede as chaves por isso na estrutura decidimos fazer ao contrário que assim ficamos com uma lista de todas as tabelas que cedem chaves a uma tabela em particular

Com isto temos uma conversão de um modelo de classes para um modelo relacional, e por fim só falta armazenar tudo na nova estrutura, sendo agora possível criar o gerador de código *SQL*.

```
tbcontent.setFgnKeys(fgnkeys);
tbcontent.setPrimKeys(primkeys);
tablescont.put(create,tbcontent);
```

3.1.2 CRUD

Esta segunda, e última camada do *dreql*, está dividida em dois problemas. O primeiro que gera o código *SQL* necessário para criar a base de dados, e o segundo que gera as funções para manipulação dos dados na base de dados, funções estas que são escritas em *perl*. Estes dois problemas são independentes um do outro já que não precisamos do resultado de um para poder implementar o outro.

No primeiro problema, para todas as tabelas temos que gerar um *create*, com todos os campos excluíndo os do tipo *set*, como podemos ver abaixo, e atribuir-lhes as opções correctas mas em *SQLite*. Podemos verificar que se o campo é uma chave, lhe atribuimos a opção *NOT NULL* de forma ao campo não ser opcional.

```
if(!data.matches("set")) {
    str.append("\t" + field.getName() + " " + field.getDatatype());
    if(null != field.getSize()) {
        str.append("(" + field.getSize() + ")");
    }
    if(null != field.getOptions()) {
        ArrayList<String> options = field.getOptions();
        for(String opt : options) {
            if(opt.matches("Editable"))
                str.append(" READONLY");
            else if(opt.matches("Visible"))
                str.append(" VISIBLE");
            else if(opt.matches("Optional"))
                str.append(" NULL");
            else if(opt.matches("Masked"))
                str.append(" MASKED");
            else if(opt.matches("AutoInc"))
                str.append(" AUTOINCREMENT");
            else if(opt.matches("Key"))
                str.append(" NOT NULL");
        }
        str.append(",\n");
    }
    else {
        str.append(" NULL,\n");
    }
}
```

Depois do código dos campos gerados, basta só acrescentar o código das chaves, tanto primárias como estrangeiras. Como podemos ver com a nova estrutura que criamos (a que faz a representação de um modelo relacional) criar o código para a base de dados é directo, sem termos que fazer praticamente nenhum cálculo auxiliar.

```
for(String table : tablescont.keySet()) {
   TableContent cont = tablescont.get(table);
   str.append("create table " + table + " (\n");
   for(Fields field : cont.getFields()) {
      str.append(fieldSQLCode(field));
   }
   str.append("\tprimary key (");
   for(String key : cont.getPrimKeys()) {
      str.append(key + ", ");
   }
   str = new StringBuffer(str.substring(0,str.length()-2));
   str.append("),\n");
   for(ObjPair<String,String> obj : cont.getFgnKeys()) {
      str.append("\tforeign key (" + obj.getFirst() + ") references "
                 + obj.getSecond() + "(" + obj.getFirst() + "),\n");
   }
   str = new StringBuffer(str.substring(0,str.length()-2));
   str.append("\n\t);\n");
}
```

No segundo problema podiamos tomar algumas opções na implementação, por exemplo, nós decidimos gerar funções de manipulação de dados para cada tabela ao invés de criarmos funções genéricas. Não podemos dizer qual destas hipóteses é a mais acertada já que cada uma apresenta as suas vantagens consoante a outra, mas para o problema que é não nos parece necessário tanta preocupação com os recursos a utilizar.

Para podermos aceder às funções do lado do *Draw* decidimos criar um módulo em *perl*, este é gerado através do *dreql*. A ideia do módulo na altura pareceu-nos a coisa mais acertada a fazer já que são duas tecnologias diferentes, mas possivelmente isto pode vir a mudar com a introdução do processamento de objectos em *Java* por parte do *Draw*.

Sendo assim, para ser possível criar o módulo simplesmente temos que imprimir para um documento, além das funções, os cabeçalhos das mesmas e as estruturas a usar. Decidimos usar as estruturas no topo como globais de forma a evitar ter que as definir sempre em cada função.

Um exemplo do resultado é o que podemos ver abaixo.

```
use DBI;
use Tie::IxHash;
use Switch;
use base 'Exporter';

our @EXPORT = ('insert_Controlo', 'delete_Controlo', 'update_Controlo',
               'select_Controlo', 'insert_Emissao', 'delete_Emissao',
               'update_Emissao', 'select_Emissao');

our $VERSION = '0.01';

our $db;
tie %{$db}, 'Tie::IxHash';
```

Para isto ser possível foi necessário dissecar toda a informação contida na nossa estrutura de forma a ser possível às funções do módulo serem capazes de fazer tudo de uma forma autónoma.

Primeiro foi necessário gerar os cabeçalhos das funções utilizando os nomes de cada

tabela como podemos ver abaixo. Para cada tabela temos quatro funções para manipulação de dados da base de dados, sendo elas o *insert* para inserir novos dados, o *update* para actualizar dados contidos na base de dados, o *delete* para apagar dados e por último o *select* para podermos ter acesso aos dados.

```
for(String table : tablescont.keySet()) {
   str.append("'insert_" + table + "', 'delete_" + table
      + "', 'update_" + table + "', 'select_" + table + "', ");
}
```

Depois de já termos os cabeçalhos, geramos as estruturas que vão ser utilizadas pelo módulo. Geramos estruturas para os campos, chaves primárias e chaves secundárias de cada tabela, de forma a ficarmos com um mapeamento de todas as tabelas e toda a informação sobre esta.

```
str.append("\n);\n" +((char) 36)+ "db->{" + tablename + "}->{'primkeys'} = {\n");
for(String pkey : table.getPrimKeys()) {
   str.append("\t" + pkey + " => undef,\n");
}
str.append("};\n" + ((char) 36) + "db->{" + tablename + "}->{'fgnkeys'} = {\n");
for(ObjPair<String,String> fgnkey : table.getFgnKeys()) {
   str.append("\t" + fgnkey.getFirst() + " => '" + fgnkey.getSecond() + "',\n");
}
```

Excerto do código que gera as estruturas para as chaves

Depois de termos todas as estruturas só necessitamos de imprimir todas as funções, como é óbvio não vamos incluir aqui o código porque são funções pouco legíveis e muito grandes, mas abaixo podemos ver um excerto de uma função para inserir numa tabela.

```
str.append("\n# Preloaded methods go here.\n\n=head2 insert_"
   + tablename + "\n\nINSERT\n\n=cut\n\nsub insert_" + tablename
   + " {\n\tmy @fields = @_;");
str.append("\n\tmy @tfield = keys " + ((char)37) + "{" + ((char)36)
   + "db->{" + tablename + "}->{'fields'}};\n\tmy " + ((char)36)
   + "diff = scalar(@tfield) - scalar(@fields);");
str.append("\n\n\tif ( " + ((char)36) + "diff < 0 ) { " + ((char)36)
   + "@ = \"Invalid number of fields submitted!\"; return undef }\n\tmy "
   + ((char)36) + "i = 0;\n\tmy @newfld;");
str.append("\n\tmy " + ((char)36) + "dbh = DBI->connect(\"dbi:SQLite:
   dbname=[DATABASE_PATH]\",\"[DB_USERNAME]\",\"[DB_PASSWORD]\",");
str.append("{ RaiseError => 1, AutoCommit => 0, unicode => 1 }) or die "
   + ((char)36) + "DBI::errstr;");
str.append("\n\tforeach ( @tfield ) {\n\t\tmy " + ((char)36)
   + "miss=0;\n\t\tswitch ( " + ((char)36) + "db->{" + tablename
   + "}->{'fields'}->{uc(" + ((char)36) + "_)}->{'type'} ) {");
```

3.2 DRAW

3.2.1 Manual de Utilização

Estrutura de um Projecto DRAW

O DRAW é uma ferramenta de auxílio à criação rápida e eficiente de aplicações *WEB* que se encontra na versão *2.2* do seu desenvolvimento. A ferramenta é auxiliada por outras ferramentas, tal como o DREQL, do qual obtém as rotinas para um fácil acesso a motores de base de dados, neste momento, esta versão apenas suporta *SQLite* mas estão a ser feitos os esforços para no futuro serem suportados, possivelmente, *MySQL* e *PostgreSQL*.

Um projecto em DRAW é dividido maioritariamente em quatro grupos, o bloco das *flags* para sinalização, o grupo das variáveis, onde são declaradas todas as variáveis que serão utilizadas no decorrer do projecto, o bloco de funções, onde se definem as rotinas que serão utilizadas nas várias transições e por fim o grafo, o bloco mais complexo, onde se apresenta uma representação textual do mapa da aplicação, da qual será extraída toda a informação acerca dos nodos e de suas transições. No início de cada projecto é sempre especificado um género de título de contexto obrigatório que é utilizado com o intuito de intitular o módulo de acesso à base de dados. Dos quatro grupos anteriormente referidos, apenas dois são de teor obrigatório, o bloco de *flags* e o grafo, visto que uma aplicação não precisa necessariamente de envolver variáveis ou funções, apesar de ser muito provável qualquer aplicação envolver a totalidade dos grupos.

Nos seguintes tópicos abordaremos como realizar um projecto com o auxílio do DRAW e visualizaremos todas as opções disponíveis para um maior controlo do resultado final.

Tipos de Dados

São vários os tipos de dados disponíveis, direccionados principalmente para a criação de formulários *WEB*. Para além dos tipos mais usuais como *String*, *Integer* e *Float*, podem também ser declaradas variáveis do tipo *Date* para representação de datas ou *Phone* para números de telefone. A lista não fica por aqui, sendo um pouco mais extensiva e abrangente tal como é apresentada abaixo.

- **String** - Utilizado no caso de variáveis de texto reduzido
- **Integer** - Utilizado em caso de inteiros
- **Float** - Abrange todo o tipo de valores com vírgula flutuante
- **Date** - Datas no formato *'DD/MM/YYYY'*
- **Phone** - Números de telefone no formato *'+xxx xxx xxx xxx'*
- **Text** - Ajustado para descrições ou para números elevados de caracteres

- **Password** - Aconselhado no uso de *passwords*. Efectua *masking* dos dados
- **URL** - Para uso de *URLs* que respeitem o seguinte formato: *'http::/...'*. Todas as variáveis deste tipo são apresentadas na forma de um *link*, quando vistas no *browser*
- **Email** - Permite referenciar contas de e-mail. Automaticamente é criado um *link*, tal como no tipo *URL*
- **Code** - Permite uma fácil introdução de código de linguagens
- **Img** - Este tipo de dados é assimilado na forma de *URL* mas é apresentado na forma de imagem quando visualizado no *browser*

Existem mais alguns tipos de dados estruturados no DRAW (*List*, *Hash*, etc.), mas apenas podem ser utilizados como retorno de funções.

Definir *Flags*

O bloco de *flags* de sinalização é obrigatório, mas apesar disso, apenas a *flag DRAW_PATH* é obrigatória quando se pretende criar um projecto. Esta serve para declarar o local onde o projecto irá ser instalado.

Todas as *flags* são precedidas do identificador *DRAW* para garantir que pertencem ao bloco de *flags*. As *flags* disponíveis são apresentadas de seguida:

- **DRAW_CSS** - Permite definir um ficheiro *CSS* para personalizar as *CGIs* geradas pelo DRAW
- **DRAW_AUTHOR** - Permite definir o nome do autor. O nome é apresentado na parte inferior das *CGIs*
- **DRAW_EMAIL** - Permite definir o e-mail do autor, para possível contacto. Tal como o nome, também é apresentado na parte inferior das *CGIs*
- **DRAW_WEB** - Permite definir a página *WEB* do autor. Partilha as mesmas características do e-mail
- **DRAW_BACK** - Permite substituir o nome dos *links* para *back*, quando é feita uma transição entre dois nodos e existe um *link* de regresso para o nodo de partida
- **DRAW_PATH** - *Flag* obrigatória que representa o local onde o projecto irá ser instalado juntamente com qualquer outro ficheiro gerado após a instalação
- **DRAW_DB_PATH** - Utilizada apenas no caso de pretender o acesso a uma base de dados. A *flag* indica o local onde a base de dados será criada e mantida
- **DRAW_DB_USER** - Utilizada no caso da base de dados necessitar de autenticação para ser acedida
- **DRAW_DB_PASS** - Partilha as mesmas características da *flag DRAW_DB_USER*

Definir uma *flag* no DRAW faz-se da seguinte forma:

FLAG = Valor;

Do lado esquerdo da equação escolhe-se a *flag* a definir e do lado direito declara-se o valor que se pretende que a *flag* assuma. Deve ter particular atenção ao finalizar a definição da *flag* acrescentar o ponto-e-vírgula (;).

Exemplo:

```
DRAW_BACK = 0;
DRAW_AUTHOR = Hugo Miguel;
DRAW_WEB = www.google.com;
DRAW_EMAIL = hadesrulez@gmail.com;
DRAW_CSS = /mycss.css;
DRAW_PATH = /Applications/MAMP/cgi-bin/Projecto/WikiRepositorium;
```

Neste exemplo começamos por desactivar a substituição automática dos *links* de retorno, definimos a informação acerca do autor, a sua página pessoal e o seu contacto e-mail. Todas as *CGIs* serão afectadas pelo *CSS* contido no ficheiro *mycss.css* e o projecto será instalado no local "*/Applications/MAMP/cgi-bin/Projecto/WikiRepositorium*". Este processo não é efectuado propriamente nesta ordem, dado que a ordem não é determinante, nem preservada quando o projecto é executado pelo DRAW.

Declarar Variáveis

A declaração de variáveis funciona quase do mesmo modo que no bloco das *flags*. Todas as declarações devem ser iniciadas com o tipo da variável e finalizadas com um ponto-e-vírgula.

Tipo Nome;

Existem mais formas mais avançadas de declarar variáveis que serão discutidas posteriormente.

Exemplo:

```
String Alias;
URL Wikipage;
EmaIL MyEmail;
img image;
```

Variáveis Especiais

As variáveis podem assumir qualquer tipo de dados permitido pelo DRAW, excepto tipos estruturados (*List*, *Hash*, etc.), isto é, todos os tipos previamente apresentados são susceptíveis de serem utilizados aquando a definição de uma variável. Para além desses tipos, existem também os tipos especiais ou de selecção. Esses tipos permitem ao utilizador uma manipulação mais abrangente sobre as *CGIs* finais. A versão *2.2* do DRAW suporta três

tipos diferentes de selecção.

- **Select** - A variável, quando requisitada para um formulário, apresenta-se na forma de uma caixa de escolha (*select box do HTML*), podendo assumir um valor dos vários propostos
- **Checkbox** - Semelhante ao tipo *Select*, excepto a capacidade de assumir mais do que um valor
- **Radio** - Semelhante ao tipo *Select*

A declaração de uma variável de selecção é feita da mesma forma que qualquer outra variável já vista até este ponto. O tipo de dados pode ser um dos três termos reservados, *Select*, *Checkbox* ou *Radio*.

Exemplo:

```
Select Field;
CHECKBOX Condition;
radio cs;
```

Nota: Qualquer termo reservado pelo DRAW não é *case sensitive*, isto é, um termo em maiúsculas tem exactamente o mesmo significado que um em minúsculas e vice-versa.

Inicializar Variáveis

Em vários projectos surgem situações em que necessitamos de uma variável previamente inicializada com um valor, mas que não tenha propriamente de transitar por todas as *CGIs* como um campo escondido (*hidden*). O DRAW possibilita a utilização de tal prática, permitindo inicializar uma variável na altura em que é definida, da seguinte forma:

Tipo Nome = Valor;

Assim podemos atribuir um valor a uma variável e utilizá-la sempre que assim o desejarmos, sem ser necessário o utilizador da aplicação interagir com essa mesma variável. Este tipo de variáveis usufruem das mesmas propriedades que qualquer outra variável, é apenas diferenciada pelo facto de começar previamente inicializada com um valor definido pelo utilizador do DRAW. Estas variáveis também podem ser passadas como parâmetros para funções sem serem submetidas pelo utilizador da aplicação *WEB* criada, devido ao facto de, o DRAW verificar automaticamente se essa variável se encontra inicializada.

Exemplo:

```
String Alias = exemplo;
Email MyEmail = hadesrulez@gmail.com;
radio cs = 1;
```

Declaração de Objectos

A principal evolução da versão anterior do DRAW para a nova versão foi, a possibilidade de declarar e efectuar operações sobre objectos. Visitaremos as possíveis operações mais à frente neste manual, focando-nos precisamente na correcta declaração de objectos.

Linguagem Módulo Nome;

Como se encontra apresentado em cima, para definir um objecto são necessárias informações sobre as propriedades do mesmo. Em primeiro lugar define-se a linguagem na qual o objecto se encontra representado. Apesar do DRAW neste momento apenas suportar objectos *Perl*, é estimado que a próxima versão do DRAW já venha a ter suporte para objectos *JAVA*. Após a linguagem, é definido o módulo ou classe da qual o objecto é criado. Por último, atribui-se um nome ao objecto para depois poder ser invocado nas transições entre os nodos.

Exemplo:

```
Perl CGI MyCgi;
Perl String MyStr;
Perl LWP::UserAgent MyObj;
```

Declarar Funções

O bloco de funções ou rotinas não é obrigatório, como já tinha sido referido anteriormente, mas não é por isso que deixa de ser um dos blocos cruciais na realização de um projecto, tanto pelo auxílio que dá à aplicação como pela liberdade que transmite ao utilizador do DRAW quando constrói a sua aplicação. As grandes vantagens da existência deste bloco são o facto de todas as funções ou rotinas se encontrarem agrupadas por módulos e de não ser necessário definir variáveis como parâmetros de saída ou entrada das mesmas, como iremos observar nos próximos tópicos.

Especificar Protótipo

Especificar protótipos de funções ou rotinas no DRAW é relativamente fácil, devido ao facto, de poder ser especificado quase da mesma forma da qual se encontra no próprio módulo. Uma função para ser definida correctamente necessita de um valor de retorno, um nome e dos parâmetros de entrada, neste caso apenas os tipos de dados de entrada, facilitando substancialmente a escrita dos protótipos, visto não precisar de incluir variáveis nesta etapa.

Retorno Nome (Parâmetros);

Como retorno de uma função podemos ter qualquer tipo de dados apresentado até este ponto, excepto objectos e tipos de selecção, mais os tipos estruturados que veremos de seguida. Tanto o nome da função como os parâmetros têm que estar de acordo com o protótipo do módulo.

Nota: Também são suportadas funções sem tipo de retorno, através do termo *Void*.

Exemplo:

```
string removeEntry(string);
img randomImg(int,float);
email convertToEmail(url,string);
string modifyEntry(string,string,url,text,string,int);
void showAll();
```

Tipos de Retorno

Todos os tipos de dados utilizados como parâmetros de entrada podem também ser utilizados como parâmetro de retorno, apesar do contrário não se verificar, dado que, tipos de dados estruturados não são suportados como parâmetros de entrada.

- **Tuplos** - Associa um valor a cada campo
- **List** - Retorna uma lista de valores do mesmo tipo
- **Lista de Tuplos** - Associa um valor a cada campo e agrega os tuplos todos numa lista
- **Hash** - Devolve uma *hash*, com chaves associadas a valores
- **Vars** - Semelhante à *Hash*, com a simples diferença de preencher os formulários automaticamente através dos valores contidos na *hash*. As chaves referem-se aos campos e os respectivos valores são atribuídos ao campo com o nome da chave

Exemplo:

```
vars searchEntry(string,int);
hash showEntries();
List(string,string,url,text,string,int) search(string,string,string,int);
List getChars(string);
(string,string,int) getUser(string);
```

No caso do retorno ser um tuplo, o termo é omitido e são apenas definidos os tipos dos campos de saída. O mesmo acontece com as listas de tuplos, onde apenas difere na adição do tipo estruturado *List*. Os restantes respeitam o método de declaração usual.

Estruturação por Módulos

Já vimos como se podem declarar funções, mas agora como é que faríamos para identificar o módulo onde essas funções se encontram? O DRAW força a que um módulo seja especificado quando é inserida uma função! Considerando agora o caso em que pretendemos importar várias funções de diferentes módulos, como é que especificávamos vários os módulos? O DRAW suporta blocos de vários módulos, compostos por todas as respectivas funções, garantindo assim uma maior abrangência dos módulos e funções que podem ser importados, que visa também um considerável aumento das potencialidades das aplicações

geradas com o auxílio da ferramenta DRAW.

Exemplo:

```
WikiRepositorium{
        hash showEntries();
        string removeEntry(string);
        vars searchEntry(string,int);
}
Storable{
        hash file_magic(string);
        hash retrieve(string);
}
```

Deste modo podemos declarar todos os módulos que pretendemos que as *CGIs* importem de forma organizada, permitindo uma fácil manutenção e mantendo aceitável a legibilidade do projecto.

Nota: As *CGIs* apenas importam os módulos que necessitam, isto é, se uma determinada *CGI* nunca executar nenhuma função, o DRAW garante que essa *CGI* não importa nenhum dos módulos declarados pelo utilizador.

Construção do Grafo

O mapa da aplicação ou grafo representa a especificação da aplicação numa forma textual. Através dessa especificação, e com o auxílio dos blocos definidos anteriormente, é possível ao DRAW gerar as várias *CGIs* e associá-las às respectivas transições. O grafo é constituído por transições e nodos, e por nodos subentenda-se *CGIs*, podendo os nodos ser de dois tipos, origem ou destino.

Transição entre Nodos

Entende-se por transição entre nodos, ligações unidireccionais entre duas *CGIs*, isto é, através de uma determinada *CGI* podemos aceder a outras *CGIs* através de ligações definidas pelo utilizador, que consistem em travessia directa, com submissão de formulários e/ou com execução de funções ou métodos. Para definir uma transição entre dois nodos, no DRAW, procede-se da forma seguidamente apresentada.

Nodo Origem —> Nodo Destino

Neste caso a transição seria feita de forma directa e seria representada na forma de uma *link*, devido ao facto de, não ser requisitada uma submissão de um formulário da *CGI* de origem para a de destino. Atenção que caso fosse executada uma função ou um método durante a transição, nestas mesmas condições, a representação iria ser exactamente a mesma, dado que, continuava a não ser requisitada uma submissão de um formulário entre as *CGIs*. Nos seguintes tópicos abordaremos os restantes casos de transições.

Exemplo:

```
MainPage -> MainPage # Apenas visivel em caso de erro!
MainPage -> AddPage
```

Criar Formulários

Formulários permitem a submissão de dados entre várias *CGIs*, dados esses que podem ser utilizados como parâmetros de funções ou métodos. Deve-se proceder do seguinte modo para criar um formulário:

$$Nodo\ Origem \text{ —> } Nodo\ Destino \Rightarrow Campo1;\ Campo2$$

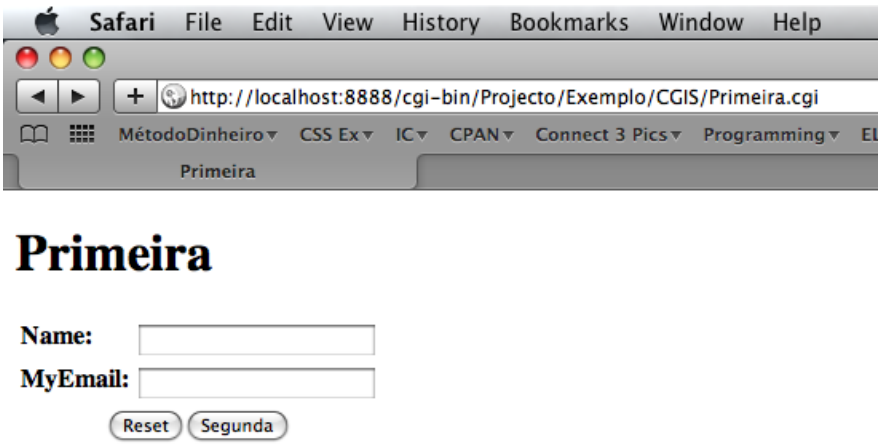
A ter em atenção que o último campo não é finalizado com um ponto-e-vírgula.

Exemplo:

```
String Name;
Email MyEmail;

Primeira -> Segunda => Name; MyEmail
```

Neste exemplo exigimos que seja submetido um formulário com dois campos, do nodo "Primeira" para o nodo "Segunda", um do tipo *String* e outro do tipo *Email*. O resultado final seria o seguinte:



Para além de transição directa, este é o tipo de transição mais simples que o DRAW suporta. Veremos tipos mais estruturados adiante.

Propriedades dos Campos

É usual, nos tempos de hoje, nem todos os campos de um formulário serem necessários para este ser submetido com sucesso. Através de um dígito de sinalização o DRAW consegue

interpretar se o campo é ou não de teor obrigatório. Também a partir de um dígito de sinalização, é possível definir itens transitivos entre *CGIs*, isto é, um campo transitivo submetido na *CGI* de origem será automaticamente preenchido na totalidade dos formulários da *CGI* de destino.

Exemplo:

```
Primeira -> Segunda =>
        Name:0:0;
        Nick:1;
        MyEmail:0:1
```

Neste exemplo observamos três variáveis ou campos, seguidos dos respectivos dígitos de sinalização. A variável *Name* não é transitiva nem de preenchimento obrigatório. A variável *Nick* é transitiva e por último a variável *MyEmail* é apenas de preenchimento obrigatório. O primeiro dígito representa sempre a transitividade da variável e o segundo a obrigatoriedade.

Criar Campos de Selecção

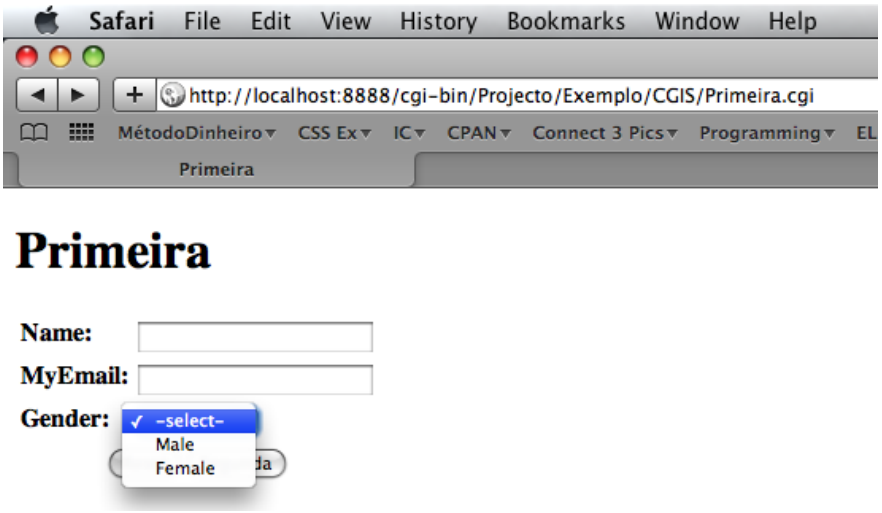
Na secção de declaração de variáveis demonstramos que era possível declarar variáveis do tipo de selecção, agora vamos mostrar como se adicionam a um formulário.

Nodo Origem —> Nodo Destino ⇒ Campo1; Campo2; Campo3[Opções]

Exemplo:

```
String Name;
Email MyEmail;
Select Gender;

Primeira -> Segunda => Name; MyEmail; Gender[Male,Female]
```



Um campo de selecção deve incluir todos os valores que pode assumir numa lista ladeada por parêntesis rectos ([]). O exemplo acima demonstra uma caixa de selecção do género da

pessoa, masculino ou feminino, de escolha obrigatória.

Associar Funções a Transições

Associar funções a transições funciona de maneira semelhante à sua declaração. Os tipos de dados de retorno são omitidos e todos os outros tipos são substituídos por uma variável do mesmo tipo.

Exemplo:

```
WikiRepositorium;

text Description;
url Wikipage;
select Field;
select Conditions;
string Alias = test;
string Title;
string Keyword;
radio cs = 1;
string Theme;
int Relevance;

WikiRepositorium{
        vars searchEntry(string,int);
        List(string,string,url,text,string,int) removeAll(string,string,string,int);
}

Primeira -> Segunda =>
        Field[Alias,Title,Wikipage,Theme,Relevance];
        Conditions[StartWith,EndWith,Contains,Equal,NotEqual];
        cs[0,1];
        Keyword;
        (Alias,Title,Wikipage,Description,Theme,Relevance)
                                removeAll(Field,Keyword,Conditions,cs);
        searchEntry(Alias,cs)
```

Com base na observação deste exemplo, podemos abordar quatro aspectos importantes.

Variáveis de Entrada e Retorno

Todas os tipos de dados são substituídos por uma variável do mesmo tipo e os tipos de retorno são omitidos, como já tinha sido referido anteriormente, mas dê particular atenção à invocação da função *removeAll*, ela encontra-se precedida por um tuplo que contem algumas variáveis declaradas previamente no bloco de variáveis. Isto acontece porque o retorno dessa função é uma lista de tuplos, sendo por isso necessário definir as variáveis às quais os valores de retorno dizem respeito. O mesmo acontece no caso do tipo de retorno ser um único tuplo.

Consistência dos Dados

O DRAW quando interpreta a disposição do grafo, procura validar ao máximo a con-

sistência dos dados inseridos pelo utilizador. A invocação de funções não é excepção, por isso internamente, o DRAW efectua uma exaustiva comparação dos tipos de dados do protótipo da função aos tipos de dados das variáveis que são definidas como parâmetros, tanto de entrada como saída.

Exemplo:

```
...
String Alias;

WikiRepositorium{
        vars searchEntry(string,int);
        List(int,string,url,text,string,int) removeAll(string,string,string,int);
}

Primeira -> Segunda =>
        ...
        (Alias,Title,Wikipage,Description,Theme,Relevance)
                                removeAll(Field,Keyword,Conditions,cs);
        ...
```

No exemplo acima, ocorreria um erro ao compilar o projecto, visto que a variável *Alias*, do tipo *String*, é utilizada como parâmetro de saída na função *removeAll*, na posição de um parâmetro do tipo *Int*, de acordo com o protótipo da função. Existem algumas casos especiais que são excepção a esta regra como veremos mais à frente. O erro gerado seria o seguinte:

"*Type mismatch! Variable: Alias, Function: removeAll, From Primeira to Segunda!*"

Parâmetros Previamente Inicializados

Variáveis previamente inicializadas no bloco de variáveis são também aceites como parâmetro de entrada de funções, isto porque o DRAW encarrega-se de verificar se essa variável se encontra inicializada no caso de não ter sido submetida e não ser de preenchimento obrigatório.

Exemplo:

```
string Alias = test;
radio cs;

WikiRepositorium{
        vars searchEntry(string,int);
}

Primeira -> Segunda =>
        cs[0,1];
        searchEntry(Alias,cs)
```

Neste caso, a função *searchEntry* iria receber como parâmetros a palavra "test" e o valor submetido no campo *cs*, isto porque, como podemos observar, a variável *Alias* não é utilizada no formulário do nodo *Primeira* para o nodo *Segunda*, logo assume o valor anteriormente

atribuído pelo utilizador.

Campos Seleccionáveis

Por último, podemos observar no exemplo anterior que a variável *cs* é do tipo *Radio*, mas a função *searchEntry* recebe como parâmetros uma *String* e um *Int*, isto não era suposto dar erro? Não, porque, como referimos acima, existem casos de excepção, mais propriamente os tipos de dados de selecção. Os tipos de selecção podem assumir qualquer outro tipo logo são ignorados na verificação efectuada pelo DRAW, isto é, podem assumir qualquer tipo de dados de entrada sem que o DRAW lance um erro para o saída.

Invocar Objectos

Para além de funções, também temos a possibilidade de invocar métodos no decorrer das travessias entre nodos. Para tal é necessário existir pelo menos um objecto declarado no bloco de variáveis. Para definirmos um método, primeiro é crucial identificar o objecto ao qual pertence, isso é possível das seguintes formas:

- *Objecto*{*Método*}
- *Objecto*{*Método(Parâmetros)*}
- *Objecto*{*Módulo —> Método*}
- *Objecto*{*Módulo —> Método(Parâmetros)*}

Os métodos não precisam de ser previamente declarados no bloco de funções dado que na definição do objecto já é induzido o respectivo módulo.

Inicializar Objectos

Um objecto antes de ser considerado apto para ser utilizado deve ser previamente criado e inicializado, em caso disso, evitando assim erros em tempo de execução.

Exemplo:

```
string Alias = test;
radio cs;
Perl CGI MyObj;

WikiRepositorium{
        vars searchEntry(string,int);
}

Primeira -> Segunda =>
        cs[0,1];
        MyObj{CGI -> new};
        searchEntry(Alias,cs)
```

No exemplo acima apresentado, declaramos um objecto denominado de *MyObj*, pertencente ao módulo *CGI* de *Perl*. Na travessia do nodo *Primeira* para o nodo *Segunda* é definido e invocado o método *new* relativo ao mesmo objecto, isto vai permitir que o objecto seja criado durante essa transição e que possa vir a ser utilizado posteriormente sem qualquer tipo de problemas.

Nota: O estado dos diversos objectos é armazenado numa base de dados criada unicamente com o intuito de armazenar todos os objectos utilizados numa determinada aplicação.

Invocar Métodos

No exemplo seguinte observamos algumas declarações de métodos que deverão executar durante as travessias pelo grafo representado. São criados dois objectos, um *CGI* e outro *Data::Dumper*, ambos partilham *Perl* como linguagem. Transitando do nodo *Primeira* para o *Segunda*, os objectos são criados e os seus estados são armazenados numa base de dados auxiliar. Se for feita a travessia contrária, serão invocados os métodos *h3* e *Dumper* dos objectos *MyObj* e *ShObj* respectivamente. Os métodos são executados da mesma forma que as funções, excepto que o resultado final do objecto é sempre armazenado numa base de dados para posterior acesso.

Exemplo:

```
string Alias = test;
radio cs;
Perl CGI MyObj;
Perl Data::Dumper ShObj;

Primeira -> Segunda =>
        MyObj{CGI -> new};
        ShObj{Data::Dumper -> new}
Segunda -> Primeira =>
        cs[0,1];
        MyObj{h3(Alias,cs)};
        ShObj{Dumper(Alias)}
```

CSS

O DRAW de raiz não aplica qualquer tipo de alteração à disposição das *CGIs*, nem ao *layout* final das páginas, mas suporta a inserção de *CSS* e presta uma pequena assistência em caso de se desejar alterar os elementos *html* existentes nas *CGIs*. Veremos como nos seguintes tópicos.

Munir um Projecto de *CSS*

Como já foi referido na secção das *flags*, um projecto pode ser munido de *CSS* através da *flag DRAW_CSS*.

DRAW_CSS = /mycss.css;

Classes Auxiliares

Aos vários elementos *html* gerados pelo DRAW, é associada uma classe para facilitar sua manipulação através de *CSS*, isto permite ao utilizador um maior controlo sobre o resultado final da sua aplicação e tanto mais, a possibilidade de personalização das páginas que compõem a aplicação.

- *forms* - Esta classe abrange todos os formulários criados para uma determinada *CGI*
- *link* - Esta classe abrange todos as âncoras criadas para uma determinada *CGI*
- *info* - Esta classe abrange a informação fornecida pelas *flags* *DRAW_AUTHOR*, *DRAW_WEB* e *DRAW_EMAIL*
- *colist* - Esta classe abrange o resultado da impressão do conteúdo de uma lista de tuplos
- *comp* - Esta classe abrange o resultado da impressão do conteúdo de um tuplo
- *hash* - Esta classe abrange o resultado da impressão do conteúdo de uma *hash*
- *misc* - Esta classe abrange o resultado da impressão do conteúdo de uma variável
- *list* - Esta classe abrange o resultado da impressão do conteúdo de uma lista

Personalizar Formulários

Para personalizar formulários é necessário aceder aos elementos gerados pelo DRAW. Manipular a totalidade do formulário é possível acedendo ao elemento com o *id* desejado. Por exemplo, assumindo-se que existe uma *CGI* *MainPage*, para se aceder ao formulário de submissão dessa *CGI* deve-se seleccionar o elemento com o *id=*"*MainPage.cgi_body*".

id="*NomeDaCGI.cgi_body*"

Se o que se pretende é manipular um campo de cada vez, então deve-se seleccionar o elemento com

id="*NomeDaCGI.cgi_NomeDoCampo_field*"

Para além destes exemplos também é possível manipular as tabelas, linhas, células, etiquetas, etc.

Base de Dados

Acessibilidade

Só é possível a existência de base de dados nos casos em que a *flag* DRAW_DB_PATH se encontra definida, visto que esta *flag* indica o local onde a base de dados será criada, permitindo assim que um módulo com as rotinas de acesso seja instalado, para posterior uso por parte da aplicação.

DRAW_DB_PATH = /Applications/MAMP/cgi-bin/Projecto/WikiRepositorium/Data/mydb.db;

Importar o Módulo

Normalmente o módulo seria importado no bloco de funções, mas não aconselhamos essa prática devido ao facto de ser preferível criar um novo módulo que o importe, isto porque, garante ao utilizador uma maior liberdade e adaptação aquando a criação de uma determinada aplicação. Deste modo o utilizador pode adaptar e personalizar as mensagens de erro, o *output* da aplicação, etc. O módulo gerado pelo DRAW recebe o nome do título do projecto, originando assim um conflito no caso da criação de dois projectos com o mesmo nome.

Nota: O módulo é compilado e instalado automaticamente pelo DRAW.

Manipulação dos Dados

Importar as funções de acesso à base de dados no DRAW é feito do mesmo modo que qualquer outra função. Numa fase inicial, depois do módulo estar devidamente instalado, importam-se todas as funções que se pretende utilizar no bloco de funções, identificando o(s) módulo(s) respectivo(s). Após as funções se encontrarem devidamente declaradas, basta definir as funções na representação do grafo tal como foi apresentado anteriormente.

Exemplo:

```
BDACCESS;

Int Id;
String Name;
Int Age;
Email MyEmail;

BDACCESS{
        vars Select_Entry(int);
        void Insert_Entry(int,string,int,email);
}

Main -> Insert =>
        Id;
        Name;
        Age;
        MyEmail;
        Insert_Entry(Id,Name,Age,MyEmail)
Main -> Select =>
        Id;
        Select_Entry(Id)
Insert -> Main
Select -> Main
```

Controlo de Erros

Existem vários erros possíveis de ocorrer quando se efectua um acesso à base de dados,

mas na maioria das vezes não haverá problema visto que o DRAW efectua a maioria das validações necessárias para que as operações funcionem correctamente, em caso de sucesso ou insucesso do pedido. Erros de inserção de dados, duplicação, valores inconsistentes, campos obrigatórios e remoção consistente dos dados são todos verificados e validados pelo DRAW.

3.3 Resultados Obtidos

Depois de estudada a estrutura da aplicação, resta-nos agora ver como esta se comporta na prática. Como foi visto acima nós podemos definir classes e relações com o uso da nossa linguagem, então como primeiro exemplo vamos criar duas classes com uma relação simples entre elas.

```
Class Emissao (
        Cod_Emissao:int(3) [key],
        Cod_Destinatario:int(3),
        Assunto: Nomes,
        DataEmissao:date);

Class Entidade (
        Cod_Entidade:int(3) [key],
        Nome,
        Morada: [optional],
        Telefone: [optional],
        Email: [optional]);

Relation ( Entidade, Emissao: 1, * );
Renom Nomes = varchar;
```

Depois de correrrmos o *dreql* com este *input* temos o seguinte resultado (na realidade são vários mas para já vamos só ver este):

```
create table Emissao (
        Cod_Emissao int(3) NOT NULL,
        Cod_Destinatario int(3) NULL,
        Assunto varchar NULL,
        DataEmissao date NULL,
        Cod_Entidade int(3) NOT NULL,
        primary key (Cod_Emissao),
        foreign key (Cod_Entidade) references Entidade(Cod_Entidade)
        );

create table Entidade (
        Cod_Entidade int(3) NOT NULL,
        Nome varchar(255),
        Morada varchar(255) NOT NULL,
        Telefone varchar(255) NOT NULL,
        Email varchar(255) NOT NULL,
        primary key (Cod_Entidade)
        );
```

Acima podemos ver então a forma como são criadas as chaves com base nas relações que cada classe tem, neste exemplo ficamos com uma chave estrangeira na tabela 'Emissao' que faz referência à tabela 'Entidade'. Também podemos visualizar a forma como cada campo é definido dependendo das suas opções, por exemplo, o campo 'Morada' tem simplesmente a opção para o campo não ser opcional, no resultado *SQL* foi-lhe então atribuído a opção *NOT NULL* e o tipo de dados *varchar* com um tamanho predefenido, como já tinhamos explicado anteriormente.

Agora vamos ver o que acontece caso tenhamos duas classes mas uma delas abstracta e outra que herda o conteúdo desta. Abaixo podemos ver as classes e o resultado depois da execução.

```
Abstract Grande (
        Coisas,
        Coisas_,
        Coisas__);

Class Controlo Extends Grande (
        Responsavel,
        Aprovacao:boolean,
        Cod_Controlo:int(3) [key]);

create table Controlo (
        Responsavel varchar(255) NULL,
        Aprovacao boolean,
        Cod_Controlo int(3) NOT NULL,
        Coisas varchar(255) NULL,
        Coisas_ varchar(255) NULL,
        Coisas__ varchar(255) NULL,
        primary key (Cod_Controlo),
        );
```

Como podemos visualizar, a tabela 'Controlo' herdou todos os campos da classe abstracta que por sua vez não aparece no modelo relacional, como é de esperar.

No próximo teste vamos acrescentar um tipo de dados composto, neste caso o *set*, e ver como se comporta.

```
Class Registo (
        Cod_Registo:int(3) [key],
        TipoRegisto: Set,
        DataRegisto:date);

create table Registo (
        Cod_Registo int(3) NOT NULL,
        DataRegisto date NULL,
        primary key (Cod_Registo)
        );

create table Registo_TipoRegisto (
        TipoRegisto varchar(255) NULL,
        Cod_Registo int(3) NOT NULL,
        primary key (TipoRegisto, Cod_Registo),
        foreign key (Cod_Registo) references Registo(Cod_Registo)
        );
```

Este tipo de dados composto dá origem a uma nova tabela que mantêm uma relação com a tabela onde o campo respectivo estava definido. A nova tabela recebe como chave estrangeira a chave primária da tabela original e como chave primária o campo que tinha *set* como tipo de dados.

Mas estes resultados não são os únicos que o *dreql* retorna, além destes são também geradas as funções que integram o *CRUD* sendo estas diferentes para cada tabela. Abaixo mostramos o exemplo da função de remoção para a tabela 'Registo', como é óbvio não vamos mostrar todas já que são funções com dimensões bastante razoáveis.

```
sub delete_Registo {
   my $condition = shift;
   my $dbh = DBI->connect("dbi:SQLite:dbname=[DATABASE_PATH]"
      ,"[DB_USERNAME]","[DB_PASSWORD]",
      { RaiseError => 1, AutoCommit => 0, unicode => 1 })
      or die $DBI::errstr;

   foreach ( keys %{$db->{Registo}->{'primkeys'}} ) {
      foreach my $tb ( keys %{$db} ) {
         if ( exists $db->{$tb}->{'fgnkeys'}->{$_} ) {
            my $notnull = 0;
            foreach my $opt ( @{$db->{$tb}->{'fields'}->{$_}->{'options'}} ) {
               if ( $opt =~ /\boptional\b/i ) { $notnull = 1; last }
               else { next }
            }
            unless ( 1 == $notnull ) {
               my $sql;
               if ( 'undef' eq $condition ) { $sql = "SELECT $_ FROM Registo" }
               else { $sql = "SELECT $_ FROM Registo WHERE $condition" }
               my $sth = $dbh->prepare($sql) or $@ = $dbh->errstr;
               $sth->execute or $@ = $dbh->errstr;
               my $rv = $sth->fetchall_hashref($_);
               foreach my $key ( keys %{$rv} ) {
                  my $sql = "UPDATE $tb SET $_=? WHERE $_=$key";
                  my $sth = $dbh->prepare($sql) or $@ = $dbh->errstr;
                  $sth->execute(undef) or $@ = $dbh->errstr;
               }
               $dbh->commit
            }
         }
      }
   }
   my $sql;
   if ( 'undef' eq $condition ) { $sql = "DELETE FROM Registo" }
   else { $sql = "DELETE FROM Registo WHERE $condition" }
   my $sth = $dbh->prepare($sql) or $@ = $dbh->errstr;
   my $res=$sth->execute or $@ = $dbh->errstr;
   $dbh->commit;
   $dbh->disconnect or warn $dbh->errstr;
   return $res;
}
```

Nos anexos B e C temos a totalidade do ficheiro de entrada e o resultado obtido, respectivamente, depois de terminado o processamento.

4 Conclusão

Durante este relatório vimos as várias camadas que dividem a aplicação, mostramos como a linguagem que desenvolvemos faz a representação de um modelo de classes e como este é posteriormente convertido para um modelo relacional (*dreql*) para podermos criar a base de dados. Vimos também como o *dreql* gera o código para as funções que são incorporadas no *CRUD* de forma a mais tarde serem utilizadas pelo *Draw*.

Estudamos em detalhe a estrutura interna do *Draw* e nisto vimos como era aplicada a filosofia dos autómatos reactivos neste. Vimos várias propriedades do *Draw* e a importância que estas têm no seu funcionamento e apresentamos várias alternativas para as mesmas.

Conseguimos concluir tudo o que nos propusemos apresentar dentro da data estipulada, que podemos dizer já que foi mais que justa, já que tivemos acesso ao enunciado praticamente no início do ano lectivo e os professores foram bastante flexíveis nas datas (no nosso entender até demais). Se voltassemos a fazer este projecto, haveriam coisas que alteraríamos como por exemplo a forma como as relações estão definidas na nossa linguagem já que podiamos ter embebido esta informação no momento em que fazemos a descrição da classe. Na parte do *Draw* voltaríamos a fazer tudo sem nenhuma alteração, já que refizemos esta camada duas vezes.

Como trabalho futuro, tendo em conta o *dreql*, estamos a pensar incluir novos tipos de dados compostos e novas opções, também estamos a pensar fazer algumas alterações à nossa linguagem. Isto se não fizermos a conversão total do código para *perl* que, para já, é a nossa primeira opção.

Quanto ao *Draw*, para já estamos a pensar criar uma *makefile* em *perl* para o nosso projecto, mas depois estamos a pensar fazer alterações sérias à estrutura. Vamos tentar fazer com que seja possível importar as próprias classes me *Java* utilizando o módulo *Java::Import* do *perl*. Também vamos tentar fazer com que a aplicação suporte diferentes motores para bases de dados (para já só suporta *SQLite*) para que seja possível o utilizador utilizar as tecnologias em que se sente mais à vontade.

Quanto ao grafo que define a aplicação, estamos a pensar convertê-lo para uma representação em *XML* criar um *XML schema*[13] para estes. Este é uma alteração que não foi pensada com o objectivo de algum ganho em eficiência mas sim pela legibilidade e manutenção.

Também estamos a pensar fazer umas melhorias no campo da segurança, incluíndo *md5*[14] na codificação das *passwords* dos utilizadores de forma a tornar a tornar esta aplicação o mais perto do real possível. Não queremos com este projecto simplesmente uma nota porque nós estamos interessados em não o deixar morrer e trabalhar nele de forma a cumprirmos todos estes objectivos que propusemos aqui.

[13] *http://en.wikipedia.org/wiki/XML_schema*
[14] *http://en.wikipedia.org/wiki/MD5*

A Gramática

```
dreql    :        commands
                  -> ^(DREQL commands)
         ;

commands          :        (command ';')+
                  -> ^(COMMANDS command+)
         ;

command  :        (a=dclass | b=relation | c=renom)
                  -> {a!=null}? ^(COMMAND dclass)
                  -> {b!=null}? ^(COMMAND relation)
                  -> ^(COMMAND renom)
         ;

dclass   : ('class' | 'Class') table '(' fields ')'
           -> ^(CLASS table fields)
           | ('abstract' | 'Abstract') table '(' fields ')'
           -> ^(ABSTRACT table fields)
           | ('class' | 'Class') table ('extends' | 'Extends') table '(' fields ')'
           -> ^(EXTENDS table table fields)
         ;

relation          :
   ('relation' | 'Relation') '(' table ',' table ':' reltype ',' reltype ')'
                  -> ^(RELATION table table reltype reltype)
         ;

renom     :       ('renom' | 'Renom') TYPE '=' TYPE
                  -> ^(RENOM TYPE TYPE)
         ;

table     :       TYPE
                  -> ^(TABLE TYPE)
         ;

fields    :       field (',' field)*
                  -> ^(FIELDS field+)
         ;

field     :       fieldname ':' datatype '[' optional ']'
                  -> ^(FIELD fieldname datatype optional)
                  | fieldname ':'  '[' optional ']'
                  -> ^(FIELD fieldname optional)
                  | fieldname ':' datatype
                  -> ^(FIELD fieldname datatype)
                  | fieldname
                  -> ^(FIELD fieldname)
         ;

fieldname         :        TYPE
                  -> ^(FIELDNAME TYPE)
         ;


optional          :        option (',' option)*
                  -> ^(OPTIONAL option+)
```

```
        ;
datatype        :       TYPE '(' NUM ')'
                -> ^(DATATYPE TYPE NUM)
                | TYPE
                -> ^(DATATYPE TYPE)
                | ('enum' | 'Enum') '(' enumopt ')'
                -> ^(DATATYPE ENUM enumopt)
                | set
                -> ^(DATATYPE set)
        ;

set     :       ('set' | 'Set') TYPE '(' NUM ')'
                -> ^(SETTYPE TYPE NUM)
                | ('set' | 'Set') TYPE
                -> ^(SETTYPE TYPE)
                | ('set' | 'Set')
                -> ^(SETTYPE SET)
        ;

enumopt :       args (',' args)*
                -> ^(ENUMOPT args+)
        ;

args    :       VAL
                -> ^(ARGS VAL)
        ;

option  :
   (a='editable' | b='visible' | c='optional' | d='masked' | e='key' | 'autoinc')
                -> {a!=null}? ^(OPTION EDITABLE)
                -> {b!=null}? ^(OPTION VISIBLE)
                -> {c!=null}? ^(OPTION OPTIONAL)
                -> {d!=null}? ^(OPTION MASKED)
                -> {e!=null}? ^(OPTION KEY)
                -> ^(OPTION AUTOINC)
        ;

reltype :
   (a='1' | b='0..1' | c='1..*' | d='0..*' | e='*' | f=NUM | NUM '..' NUM)
                -> {a!=null}? ^(RELTYPE ONE)
                -> {b!=null}? ^(RELTYPE ZEROTOONE)
                -> {c!=null}? ^(RELTYPE ONETON)
                -> {d!=null}? ^(RELTYPE ZEROTON)
                -> {e!=null}? ^(RELTYPE NTOM)
                -> {f!=null}? ^(RELTYPE NUM)
                -> ^(RELTYPE NUM NUM)
        ;

NUM     :       ('0'..'9')+
        ;

TYPE    :       ('A'..'Z' | 'a'..'z' | '_')+
        ;

VAL     :       ('\'' (options{greedy=false;}:.)*  '\'')
        ;

NS      :       (' ' | '\t' | '\n' | 'r') { skip(); }
```

```
        ;

COMMENT :       '/*' (options {greedy=false; }:.)* '*/' { skip(); }
        ;

LINE_COMMENT        :       '//' ~('\n'|'\r')* '\r'? '\n' { skip(); }
        ;
```

B Modelo de Classes

```
Abstract Grande (
        Coisas,
        Coisas_,
        Coisas__);
Class Registo (
        Cod_Registo:int(3) [key],
        TipoRegisto: Set,
        DataRegisto:date);
Class Controlo Extends Grande (
        Responsavel,
        Aprovacao:boolean,
        Cod_Controlo:int(3) [key] );
Class Recepcao (
        Cod_Recepcao:int(3) [key],
        Cod_Remetente:int(3) ,
        Assunto: Nomes,
        DataRecepcao:date);
Class Emissao (
        Cod_Emissao:int(3) [key],
        Cod_Destinatario:int(3),
        Assunto: Nomes,
        DataEmissao:date);
Class Entidade (
        Cod_Entidade:int(3) [key],
        Nome,
        Morada: [optional],
        Telefone: [optional],
        Email: [optional]);
Relation ( Registo, Emissao: 1, * );
Relation ( Registo, Controlo: 1, 1 );
Relation ( Registo, Recepcao: 1, *);
Relation ( Entidade, Emissao: 1, * );
Relation ( Entidade, Recepcao: 1, * );
Renom Nomes = varchar;
```

C Modelo Relacional

```
create table Registo_TipoRegisto (
        TipoRegisto varchar(255) NULL,
        Cod_Registo int(3) NOT NULL,
        primary key (TipoRegisto, Cod_Registo),
        foreign key (Cod_Registo) references Registo(Cod_Registo)
        );
create table Controlo (
        Responsavel varchar(255) NULL,
        Aprovacao boolean,
        Cod_Controlo int(3) NOT NULL,
        Coisas varchar(255) NULL,
        Coisas_ varchar(255) NULL,
        Coisas__ varchar(255) NULL,
        Cod_Registo int(3) NOT NULL,
        primary key (Cod_Controlo),
        foreign key (Cod_Registo) references Registo(Cod_Registo)
        );
create table Emissao (
        Cod_Emissao int(3) NOT NULL,
        Cod_Destinatario int(3) NULL,
        Assunto varchar NULL,
        DataEmissao date NULL,
        Cod_Registo int(3) NOT NULL,
        Cod_Entidade int(3) NOT NULL,
        primary key (Cod_Emissao),
        foreign key (Cod_Registo) references Registo(Cod_Registo),
        foreign key (Cod_Entidade) references Entidade(Cod_Entidade)
        );
create table Recepcao (
        Cod_Recepcao int(3) NOT NULL,
        Cod_Remetente int(3) NULL,
        Assunto varchar NULL,
        DataRecepcao date NULL,
        Cod_Registo int(3) NOT NULL,
        Cod_Entidade int(3) NOT NULL,
        primary key (Cod_Recepcao),
        foreign key (Cod_Registo) references Registo(Cod_Registo),
        foreign key (Cod_Entidade) references Entidade(Cod_Entidade)
        );
create table Registo (
        Cod_Registo int(3) NOT NULL,
        DataRegisto date NULL,
        primary key (Cod_Registo)
        );
create table Entidade (
        Cod_Entidade int(3) NOT NULL,
        Nome varchar(255),
        Morada varchar(255) NOT NULL,
        Telefone varchar(255) NOT NULL,
        Email varchar(255) NOT NULL,
        primary key (Cod_Entidade)
        );
```

Glossário

ANTLR ANother Tool for Language Recognition, 2, 5

UML Unified Modeling Language, 2, 5

XML Extensible Markup Language, 31