DjDSL: Tcl and NX for Building DSLs

Stefan Sobernig

*hommage à **D**isc Jockey **S**uper **L**eiwand*

Leiwand?

- Pronounced as [ˈlaɪ̯vant]
- Means: Awesome, cool, excellent
- Sometimes emphasised as "urleiwand" or "voi leiwand" (totally awesome)!

See metropole.at (https://metropole.at/word-of-the-week-leiwand/), last access: 28.06.2022

Overview

- DSL, language-development systems ("workbenches")
- DjDSL
- A gentle primer
- Outlook

Domain-specific Language (DSL)

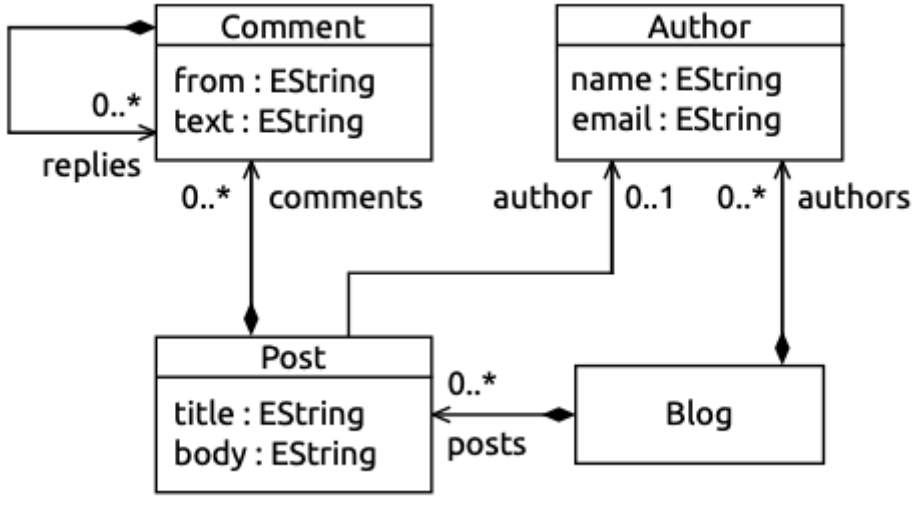
- A DSL is a little language, focused on a particular aspect of a software system or of an application domain.
- A DSL does not allow for building a complete software system, but you often use multiple DSLs in a system mainly written in a general-purpose language.
- Tcl has a number of DSLs (per command):
    - `expr {foobar()}`
    - free-form `clock scan`: `clock scan "next last thiss now tomorrow ago"`
    - `regexp`/`regsub`

See also Fowler (2010): Domain-specific Software Languages

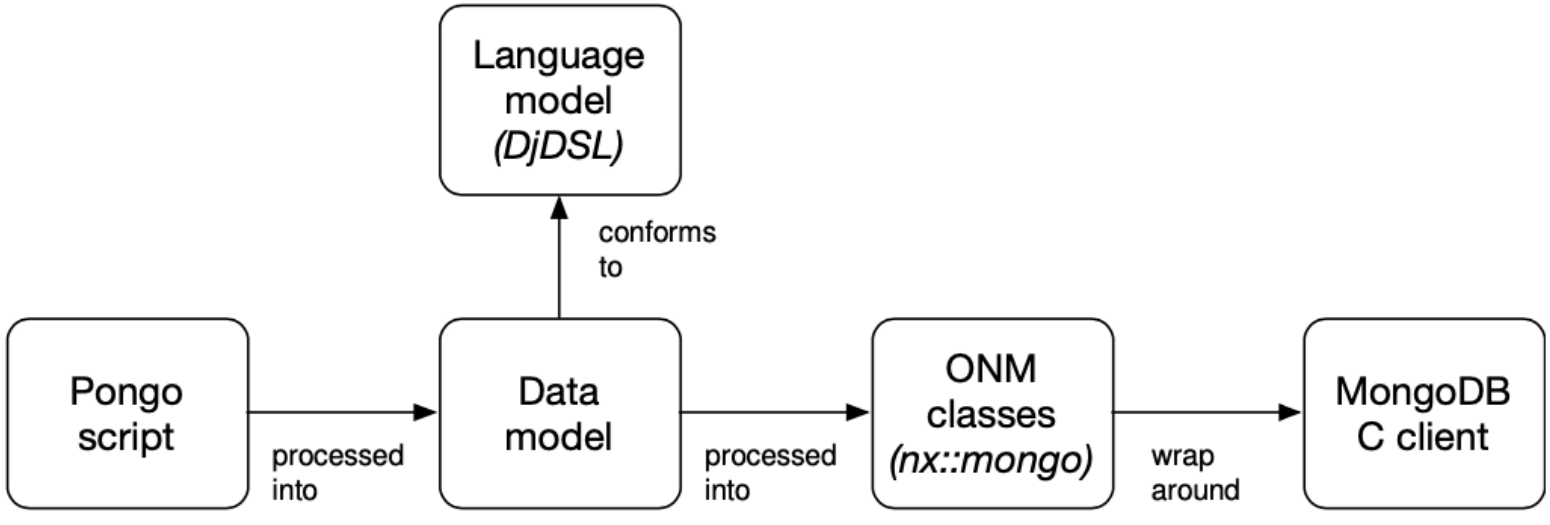
Pongo (Emfatic) DSL (1) : Script

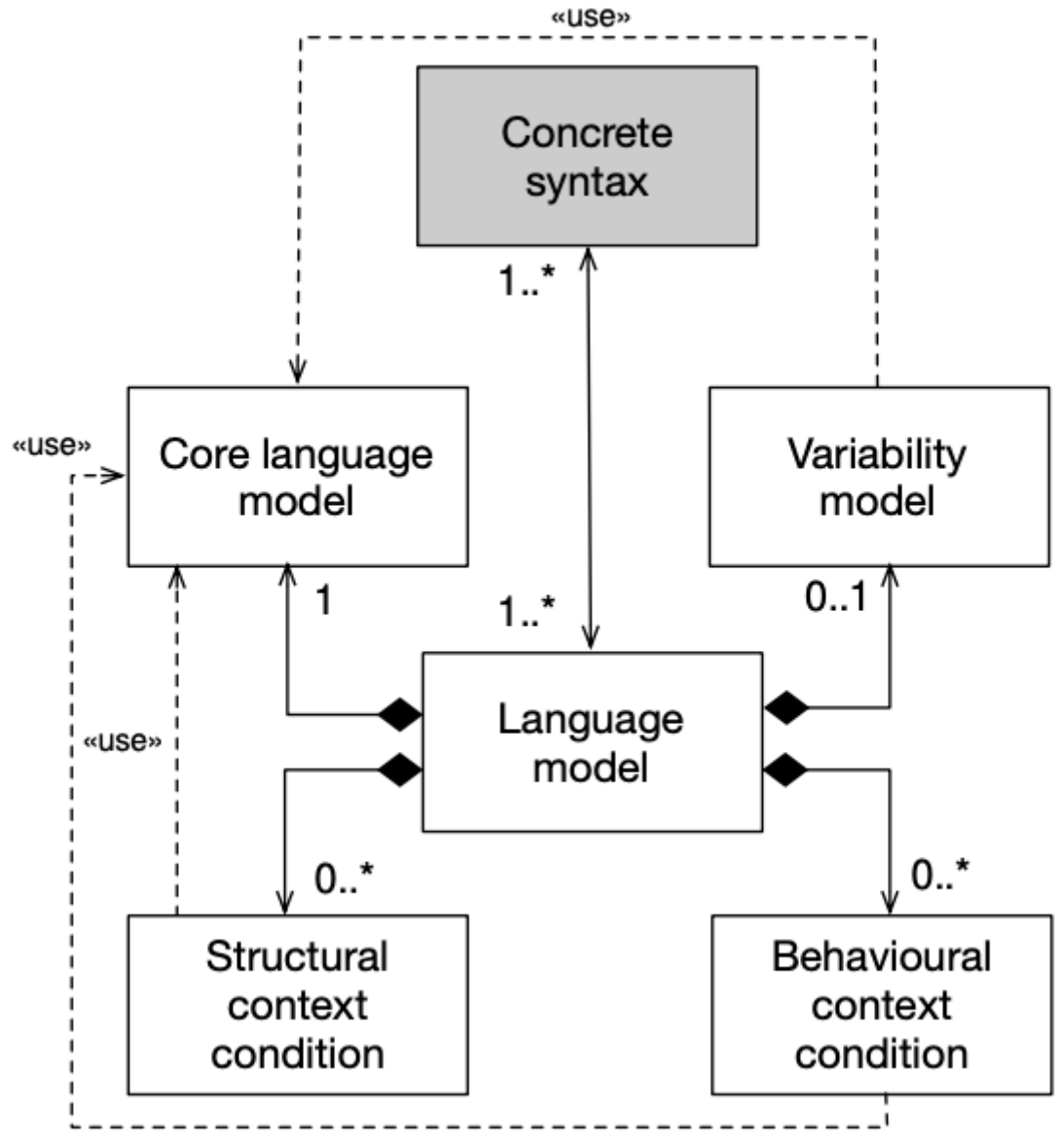
```
{java}
    @db class Blog {
      val Post posts;
      val Author authors;
    }
    class Post {
      attr String title;
      attr String body;
      ref Author author;
    }
    class Author {
      attr String name;
      attr String email;
    }
```

Data model


Comment
from : EString
text : EString
0..*
replies
Author
name : EString
email : EString
0..*
comments
author
0..1
0..*
authors
Post
title : EString
body : EString
0..*
posts
Blog

Pongo (Emfatic) DSL (2): Object-NoSQL-Mapper




«use»
Concrete syntax
1..*
«use»
Core language model
Variability model
1
1..*
0..1
«use»
Language model
0..*
0..*
Structural context condition
Behavioural context condition

DjDSL as a DSL development system written in Tcl/ NX that is

- ... **language-based** (language development from within a host/ implementation language);
- ... **compositional** (DSL features are implemented as first-class language/ code units);

DjDSL supports

- ... developing internal, external, and mixed DSL (syntaxes) based on common assets;
- ... (all) DSL composition styles: extension, unification, extension composition, and self-extension;

State of practise 👽

- Language workbenches for **single**-DSL development
- Ex.: Eclipse Xtext (ANTLR) on top of Eclipse EMF/ Java
- More recent:
    - Frontend: Exclipse Theia (browser-based IDEs)
    - Backend: Language-server protocol, LSP, servers (originating from Visual Studio; standalone, or provided by Eclipse Xtext)

State of research 

- Helvetia + PetitParser (language-based: Pharo Smalltalk)
- Neverlang + AiDE (language-based: Java, plus tooling)
- FeatureHouse + Spoofax (tool-based)
- Fusion + Gromp (tool-based)

In [2]:

```
## What we want?
## In the spirit of TIP 131
## "Read My Mind and Do What I Mean"

set script {
  @db class Blog {
    val Post posts;
    val Author authors;
  }

  class Post {
    attr String title;
    attr String body;
    ref Author author; b
  }

  # ...
}
### !!! MAGICAL BOX !!!
rmmadwim $script
### !!! !!! !!! !!! !!!

Post create ::p1 -title "My first blog post"
Blog create ::myBlog -posts ::p1
::myBlog save; # voilà, persists to MongoDB!
```

```
invalid command name "rmmadwim"
```

DjDSL provides a touch of `rmmadwim` DSL development:

1. a systematic manner to define a Pongo language model (class, val, attr);
2. one or several concrete syntaxes on top of that language model:
    - **internal**: piggybacking onto Tcl's syntax
    - **external**: parsing expression grammars (PEGs)
3. add behavioural features to the structural language model (e.g., Datastore backends: MongoDB connector etc.)

In [3]:

```
# 1) Language-model definition
package req djdsl::lm
namespace import ::djdsl::lm::*

Asset create Base {
  LanguageModel create Model {
      Classifier create Element
      Classifier create NamedElement -superclasses Element {
        :property -accessor public name:required,alnum
      }
      # class
      Classifier create Class -superclasses NamedElement
      # attr
      Classifier create Attribute -superclasses NamedElement
      # attr/ ref
      Classifier create Reference -superclasses NamedElement
  }; # Model
}; # Base

puts [info commands ::Base::Model::*]
set aPongoDataModelElement [::Base::Model::Class new -name "Blog"]
```

```
::Base::Model::Attribute ::Base::Model::slot ::Base::Model::Class ::Base::Mode
l::Element ::Base::Model::Reference ::Base::Model::NamedElement
```

Out[3]:

```
::nsf::__#2
```

Compositions produce a *resulting* language model, to be instantiated by a generated parser

In [5]:
```
Composition create ShallowPongo \
  -binds ::Base \
  -base ::Base::Model
```

Out[5]:
```
::ShallowPongo
```

In [6]:

```
# 2) Define an external concrete-syntax (using an EBNF-like grammar notation)

set aFirstGrammar {
  # 2a) low-level details (keywords, delimiter characters, ...)
  void:  REF       <- WS 'ref' WS ;
  void:  ATTR      <- WS 'attr' WS ;
  void:  VAL       <- WS 'val' WS ;
  void:  CLASS     <- WS 'class' WS ;
  void:  DB        <- WS '@db' WS ;
  void:  OBRACKET  <- WS '{' WS ;
  void:  CBRACKET  <- WS '}' WS;
  void:  SCOLON    <- WS ';' WS;
  void:  WS        <- (COMMENT / <space>)*;
  void:  COMMENT   <- '//' (!EOL .)* EOL ;
  void:  EOL       <- '\n' / '\r' ;
};
```

Out[6]:

In [7]:

```
append aFirstGrammar {
  # 2b) high-level details
  P           <- `Model` ClsStmt+;
  ClsStmt     <- `Class` root:(`true` DB / `false` !DB) CLASS name:ID
                    OBRACKET StmtList CBRACKET;
  StmtList    <- (Stmt SCOLON)*;
  Stmt        <- attributes:AttrStmt / references:RefStmt;
  RefStmt     <- `Reference` containment:(`false` REF / `true` VAL)
                    referenceType:(`$root elements $0` ID) WS name:ID;
  AttrStmt    <- `Attribute` ATTR attributeType:(`$root datatypes $0` ID)
                   WS name:ID;
  ID          <- <alnum>+;
};
```

Out[7]:

The complete grammar

In [8]:
```
set aFirstGrammar
```

Out[8]:
```
# 2a) low-level details (keywords, delimiter characters, ...)
void:  REF      <- WS 'ref' WS ;
void:  ATTR     <- WS 'attr' WS ;
void:  VAL      <- WS 'val' WS ;
void:  CLASS    <- WS 'class' WS ;
void:  DB       <- WS '@db' WS ;
void:  OBRACKET <- WS '{' WS ;
void:  CBRACKET <- WS '}' WS;
void:  SCOLON   <- WS ';' WS;
void:  WS       <- (COMMENT / <space>)*;
void:  COMMENT  <- '//' (!EOL .)* EOL ;
void:  EOL      <- '\n' / '\r' ;

# 2b) high-level details
P          <- `Model` ClsStmt+;
ClsStmt    <- `Class` root:(`true` DB / `false` !DB) CLASS name:ID
                OBRACKET StmtList CBRACKET;
StmtList   <- (Stmt SCOLON)*;
Stmt       <- attributes:AttrStmt / references:RefStmt;
RefStmt    <- `Reference` containment:(`false` REF / `true` VAL)
                referenceType:(`$root elements $0` ID) WS name:ID;
AttrStmt   <- `Attribute` ATTR attributeType:(`$root datatypes $0` ID)
                WS name:ID;
ID         <- <alnum>+;
```

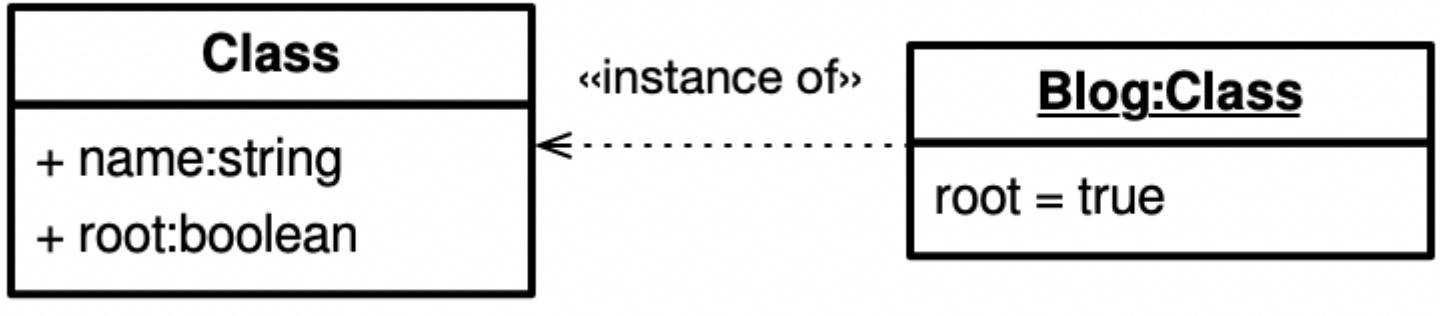
The Pongo statement ...

```
@db class Blog
```

... becomes processed via ...

```
        ClsStmt     <- `Class` root:(`true` DB / `false` !DB) CLASS name:ID OBRACKE
T StmtList CBRACKET;
void:   DB          <- WS '@db' WS ;
```

... into:



In [9]:

```
# Provide for the necessary dependencies:
package req djdsl::opeg
namespace import ::djdsl::opeg::*

# 2c) Box grammar into a Tcl command/ an NX class:

Grammar create PongoGrm -start P $aFirstGrammar

# 2d) Instantiate grammar to produce another Tcl command (NX object) acting as a p
arser:
PongoGrm new;
```

Out[9]:

Some "rmmadwim", finally! 🙂

In [13]:

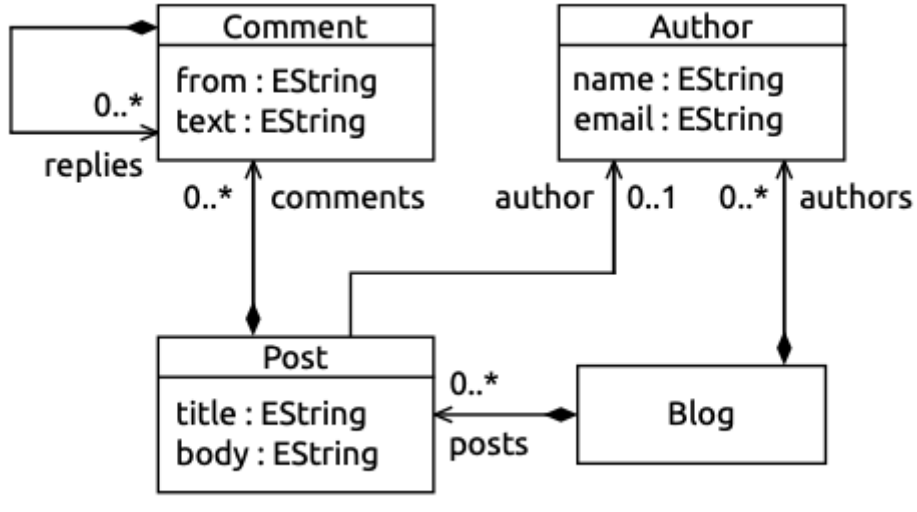
```
set lmf [LanguageModelFactory new -lm ::ShallowPongo::Model]
set pongoParser [PongoGrm new -factory $lmf]

set blogModel [$pongoParser parse {
    @db class Blog {
      val Post posts;
      val Author authors;
    }
    class Post {
      attr String title;
      attr String body;
      ref Author author;
    }
    class Author {
      attr String name;
      attr String email;
    }
}];
```

Out[13]:

In [14]:

```
puts [$blogModel info class]
foreach c [$pongoDataModel info children] {
  puts "[$c info class] => [$c name get]"
}
```

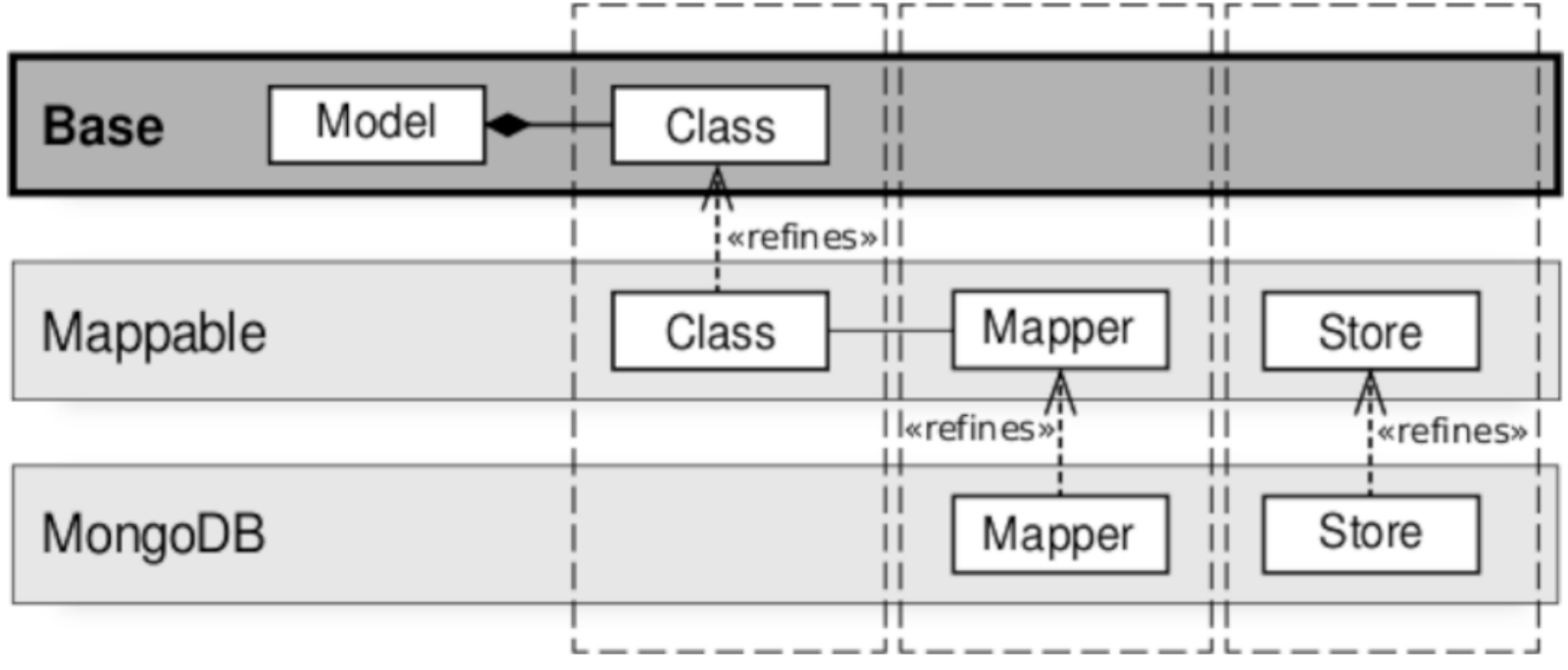
```
::ShallowPongo::Model
::ShallowPongo::Model::Attribute => name
::ShallowPongo::Model::Attribute => title
::ShallowPongo::Model::Attribute => email
::ShallowPongo::Model::Attribute => body
::ShallowPongo::Model::Reference => posts
::ShallowPongo::Model::Class => Author
::ShallowPongo::Model::Reference => author
::ShallowPongo::Model::Reference => authors
::ShallowPongo::Model::Class => Post
::ShallowPongo::Model::Class => Blog
```



Still, some "rmmadwim" missing ! 😞

In [15]:
```
set blog [${blogModel}::Blog new]
```

```
invalid command name "::ShallowPongo::__#0::Blog"
```


Base
Model
Class
«refines»
Mappable
Class
Mapper
Store
«refines»
«refines»
MongoDB
Mapper
Store

Collaborations allow one to add structural/ behavioural increments 🧚

In [16]:
```
Asset create Backends {

    Collaboration create Mappable {
      Classifier create Store
      Classifier create Mapper
      Role create Element
      Role create Class
    }; # Mappable

    Collaboration create MongoDB {
      Role create Mapper
      Role create Store;
    }; # MongoDB

}; # Backends
```

Out[16]: ::Backends

Compositions produce a *resulting* language model, to be instantiated by a generated parser

In [18]:
```
Composition create MongoDBPongo \
  -binds {Backends Base} \
  -base ::Base::Model \
  -features {
    ::Backends::Mappable
    ::Backends::MongoDB
}
```

Out[18]:
```
::MongoDBPongo
```

Let's recreate the Pongo parser, using MongoDBPongo (rather than ShallowPongo)

In [19]:
```
set lmf [LanguageModelFactory new \
               -lm ::MongoDBPongo::Model]
set pongoParser [PongoGrm new -factory $lmf]

set blogModel [$pongoParser parse {
    @db class Blog {
      val Post posts;
      val Author authors;
    }
    class Post {
      attr String title;
      attr String body;
      ref Author author;
    }
    class Author {
      attr String name;
      attr String email;
    }
  }]
```

Out[19]:
```
::MongoDBPongo::__#d
```

In [20]:

```
set store [$blogModel new store]
set blog [${store}::Blog new]
set post [${store}::Post new \
  -title "A post" -body "Some text"]
$blog posts set $post;
$store save $blog
```

So, there is a ✨ touch ✨ of `rmmadwim` for DSL development :

1. a systematic manner to define a language model via `LanguageModel`, and `Collaboration`, `Composition` ✅
2. one or several concrete syntaxes on top of that language model via OPEG grammars ✅
3. add behavioural features to the structural language model (e.g., Datastore backends: MongoDB connector etc.); `Collaboration` ✅

There is a complete Pongo/ MongoDB tutorial available at GitHub (https://github.com/mrcalvin/djdsl/blob/master/tutorials/pongo.adoc).

Outlook & wishlist 🗻

1. OPEGs based on CPARAM 🚀 (currently: TclOO Parser)
2. Incremental PEG parsing for `tcllib`'s PT
3. Incremental O(bject)PEGs
4. `tcljupyter`: Some missing pieces
5. Tooling improvements (DjDSL):
    - Grammar-editing services
    - Generator for Jupyter kernels and notebook extensions (Jupyter as IDE)
    - Language services

Kudos 👏 to Tcl community members

- Gustaf Neumann for NSF/ NX (https://next-scripting.org/)
- Andreas "aku" Kulpries for tcllib's parser tools (pt) (https://core.tcl-lang.org/tcllib/doc/trunk/embedded/md/tcllib/files/apps/pt.md)
- Marc Janssen for tcljupyter (https://github.com/mpcjanssen/tcljupyter)
- Stuart "stu" Cassoff for spotoconf (https://chiselapp.com/user/stwo/repository/spotoconf/)
- Roy Keene for KitCreator (http://kitcreator.rkeene.org/fossil/)

Recommended ☀️ *summer* 🌞 read!

or, the Tcl way, give it a try: https://github.com/mrcalvin/djdsl/ (https://github.com/mrcalvin/djdsl/)