Technology modeling with MegaL in software development

Ralf Lämmel
Software Languages Team
University of Koblenz-Landau

MegaL is a technology modeling language developed by the Software Languages Team.
Acknowledgement: Jean-Marie Favre designed this (such a) slide.

We have a problem!
Too many technologies.
Too little time.
Software models vs. technology models

• **Software** models
  • Structure and behavior of the software system

• **Technology** models
  • *Entities* related to technology usage
  • *Relationships* between those entities
For comparison —
Different kinds of software models

- Data models (to be implemented in a database)
- Structural models (to be implemented in software)
  - Class diagrams (to model state and relationships)
  - Package diagrams (to group classes)
- Behavioral models (to be implemented in software)
  - Sequence diagrams (to define specific „scenarios“)
  - Activity diagrams (to define general „workflows“)
  - State diagrams (to define object states and transitions)
A **structural** model for a metro’s turnstile
(A software model — not a technology model)
A **behavioral** model for a metro’s turnstile
(A software model — not a technology model)
Examples of technology models

- Compilation with Java’s javac compiler
- Population of a mySQL database
- XML data binding with Java’s JAXB technology

We also use the name "megamodel" sometimes for "technology models". We use the MegaL modeling language for technology models. MegaL is being developed by the Software Languages Team.
Compilation with Java’s javac compiler


Given a source file (a Java program), the compiler produces bytecode (a .class file). The bytecode could be executed directly by the JVM (Java Virtual Machine), but our technology model is not going to cover the execution aspect.
Compilation with Java’s javac compiler

$ ls HelloWorld.*
HelloWorld.java
$ more HelloWorld.java
public class HelloWorld {

    public static void main(String[] args) {
        System.out.println("Hello, World");
    }
}

$ javac HelloWorld.java
$ ls HelloWorld.*
HelloWorld.class
HelloWorld.java
Compilation with Java’s javac compiler

- Java : Language
- JavaByteCode : Language
- Compiler < Technology
- javac : Compiler
- aJavaProgram : File — e.g., HelloWorld.java
- aJavaByteCodeProgram : File — e.g., HelloWorld.class
- aJavaProgram \(\text{elementOf}\) Java
- aJavaByteCodeProgram \(\text{elementOf}\) JavaByteCode
- compilation : Java \(\rightarrow\) JavaByteCode
- javac defines compilation
- compilation(aJavaProgram) \(\rightarrow\) aJavaByteCodeProgram

Let’s look at the language elements of MegaL in some detail.
Compilation with Java’s javac compiler

- **Java**: Language
  - "Java" is an entity of type "Language".
- **JavaByteCode**: Language
- **Compiler**: Technology
- **javac**: Compiler
  - "javac" is an entity of type "Compiler".
- **aJavaProgram**: File — e.g., HelloWorld.java
- **aJavaByteCodeProgram**: File — e.g., HelloWorld.class
- **aJavaProgram elementOf** Java
- **aJavaByteCodeProgram elementOf** JavaByteCode
- **compilation**: Java → JavaByteCode
  - **javac** defines **compilation**
  - **compilation**(aJavaProgram) ↦ aJavaByteCodeProgram
Compilation with Java’s javac compiler

- Java : Language
- JavaByteCode : Language
- Compiler < Technology
- javac : Compiler
- aJavaProgram : File — e.g., HelloWorld.java
- aJavaByteCodeProgram : File — e.g., HelloWorld.class
- aJavaProgram elementOf Java
- aJavaByteCodeProgram elementOf JavaByteCode
- compilation : Java → JavaByteCode
- javac defines compilation
- compilation(aJavaProgram) ↦ aJavaByteCodeProgram
Compilation with Java’s javac compiler

- Java : Language
- JavaByteCode : Language
- Compiler < Technology
- javac : Compiler
- aJavaProgram : File — e.g., HelloWorld.java
- aJavaByteCodeProgram : File — e.g., HelloWorld.class
  - aJavaProgram elementOf Java
  - aJavaByteCodeProgram elementOf JavaByteCode
- compilation : Java → JavaByteCode
  - javac defines compilation
  - compilation(aJavaProgram) ↦ aJavaByteCodeProgram
Given a MySQL database, the MySQL workbench is used to populate the database. To this end, the database schema (some CREATE TABLE statements) are executed as a script, followed by another script with sample data (some INSERT statements).
https://www.mysql.com/products/workbench/
DROP TABLE IF EXISTS employee;
DROP TABLE IF EXISTS department;

# Departments

CREATE TABLE IF NOT EXISTS department (  
id INTEGER AUTO_INCREMENT PRIMARY KEY,  
name VARCHAR(100) NOT NULL,  
did INTEGER,  
FOREIGN KEY (did) REFERENCES department(id) ON DELETE CASCADE ON UPDATE CASCADE)

# Employees

CREATE TABLE IF NOT EXISTS employee (  
id INTEGER AUTO_INCREMENT PRIMARY KEY,  
name VARCHAR(50) NOT NULL,  
adress VARCHAR(50) NOT NULL,  
salary DOUBLE NOT NULL,  
manager BOOL NOT NULL,  
did INTEGER NOT NULL,  
FOREIGN KEY (did) REFERENCES department(id) ON DELETE CASCADE ON UPDATE CASCADE)
# Departments

INSERT INTO department (name) VALUES ("Research"); -- deptId = 1
INSERT INTO department (name) VALUES ("Development"); -- deptId = 2
INSERT INTO department (name,did) VALUES ("Dev1",2); -- deptId = 2
INSERT INTO department (name,did) VALUES ("Dev1.1",3); -- deptId = 2

# Employees

INSERT INTO employee (name, address, salary, manager, did) 
SELECT "Craig", "Redmond", 123456, true, 1 
UNION ALL 
SELECT "Ray", "Redmond", 234567, true, 2 
UNION ALL 
SELECT "Klaus", "Boston", 23456, true, 3 
UNION ALL 
SELECT "Karl", "Riga", 2345, true, 4 
UNION ALL 
SELECT "Erik", "Utrecht", 12345, false, 1 
UNION ALL 
SELECT "Ralf", "Koblenz", 1234, false, 1 
UNION ALL 
SELECT "Joe", "Wifi City", 2344, false, 4;
Population of a MySQL database

- Data definition language < Language
- Data manipulation language < Language
- SQL : Language
- SQL DDL : Data definition language
- SQL DML : Data manipulation language
- SQL DDL subsetOf SQL
- SQL DML subsetOf SQL
- RDBMS < Technology
- MySQL : RDBMS
- IDE < Technology
- MySQL Workbench : IDE
- someCreateStatements : File
- someInsertStatements : File
- someCreateStatements elementOf SQL DDL
- someInsertStatements elementOf SQL DML
- database : Artifact
- execution : SQL → Action
- MySQL Workbench defines execution
- creation : Action
- insertion : Action
- execution(someCreateStatements) ↦ creation
- execution(someInsertStatements) ↦ insertion
- creation precedes insertion
- creation modifies database
- insertion modifies database

Let's look at this bigger model, piece by piece.
Taking apart SQL into DDL and DML

- Data definition language < Language
- Data manipulation language < Language
- SQL : Language
- SQL DDL : Data definition language
- SQL DML : Data manipulation language
- SQL DDL subsetOf SQL
- SQL DML subsetOf SQL
Two types of technology involved

- RDBMS < Technology
- MySQL : RDBMS
- IDE < Technology
- MySQL Workbench : IDE
The involved artifacts

- `someCreateStatements` : File
- `someInsertStatements` : File
- `someCreateStatements` `elementOf` SQL DDL
- `someInsertStatements` `elementOf` SQL DML
- `database` : Artifact

The SQL scripts are files, definitely.

The database could be a file, several files, or even a remote-accessible resource. So we don’t get more specific than saying it’s an „artifact“. 
Script execution

- execution : SQL → Action
- MySQL Workbench defines execution
- creation : Action
- insertion : Action
- execution(someCreateStatements) → creation
- execution(someInsertStatements) → insertion
- creation precedes insertion
- creation modifies database
- insertion modifies database

The workbench can execute SQL scripts.
The meaning of script execution is an action.
DDL script needs to be executed before DML script.
The actions affect (modify) the database.
XML data binding with Java’s JAXB technology

http://en.wikipedia.org/wiki/Java_Architecture_for_XML_Binding

Source: http://docs.oracle.com/cd/E17802_01/webservices/webservices/docs/1.6/tutorial/doc/JAXBWorks2.html
A technology model for JAXB (XML-data binding of the Java platform)

- Platform < Technology
- XML data binding technology < Technology
- Code generator < Technology
- Library < Technology

- Java platform : Platform
- Java : Language
- Java platform dependsOn Java

- JAXB : XML data binding technology
- JAXB partOf Java platform
- xjc : Code generator
- javax.xml.bind : Library
- xjc partOf JAXB
- javax.xml.bind partOf JAXB

Part 1/5: Technology break-down
## A technology model for JAXB (XML-data binding of the Java platform)

- **XML**: Language
- **XSD**: Language
- **XSD** subsetOf **XML**

### JAXB.Java
- **JAXB.Java**: Language
- **JAXB.Java** subsetOf **Java**
- **generation**: XSD → **JAXB.Java**

- **xmlTypes**: File+
- **xmlTypes** elementOf **XSD**
- **javaClasses**: File+
- **javaClasses** elementOf **JAXB.Java**
- **xjc** defines **generation**
- **generation**(xmlTypes) ↦ **javaClasses**

---

**XSD** is the XML schema language; it happens to be an XML language itself.

We introduce the name „JAXB.Java“ to refer to the specific Java subset that is used by JAXB’s code generator.

The relevant artifacts, i.e., XML types as input for code generation and Java classes as output. (The types and classes may be distributed over multiple files — thus the „+“.)
A technology model for JAXB
(XML-data binding of the Java platform)

• Code generation: Concept — in the sense of generating code from other
• Serialization: Concept — i.e., object serialization (aka un-/marshalling)
• Annotation: Concept — in the sense of Java annotations or metadata
• Validation: Concept — Schema-based validation or conformance checking

JAXB uses Code generation — JAXB obviously relies on code generation
JAXB supports Serialization — JAXB can be used for object serialization
xjc implements Code generation — The code generator implements …
javax.xml.bind implements Serialization — The library implements …
javaClasses uses Annotation — Generated classes use JAXB annotations
XSD supports Validation — XSD comes with a form of validation

Part 3/5: Involved concepts
### Part 4/5: Instance-level mapping

The unmarshalling function arises as the meaning of a code fragment that is part of the application that uses JAXB. That code clearly utilizes ("refers to") the JAXB library.

The "language" for run-time object graphs:

- `JVM.ObjectGraphs` : Language
- `anObjectGraph` : Transient
- `anObjectGraph elementOf JVM.ObjectGraphs`

We take an XML document as input and somehow invoke "unmarshalling" to retrieve a run-time object graph.

The type "Transient" conveys that we face a run-time artifact.

- `anXmlDoc` : File
- `anXmlDoc elementOf XML`
- `unmarshalling : XML → JVM.ObjectGraphs`
- `unmarshalling(anXmlDoc) ↦ anObjectGraph`

The unmarshalling function arises as the meaning of a code fragment that is part of the application that uses JAXB.

- `application` : File+
- `application elementOf Java`
- `aMethodCall` : Fragment
- `aMethodCall partOf application`
- `aMethodCall refersTo java.xml.bind`
- `aMethodCall defines unmarshalling`
A technology model for JAXB (XML-data binding of the Java platform)

- anXmlDoc conformsTo xmlTypes
- anObjectGraph conformsTo javaClasses
- xmlTypes correspondsTo javaClasses
- anXmlDoc correspondsTo anObjectGraph

Part 5/5: Conformance and correspondence

Not an arbitrary document — rather one that conforms to the given schema!

Not an arbitrary object graph — rather one whose class is part of the given classes!

There is a correspondence at the type level. For each „complex XSD type“, there is Java class of the „same“ name and equivalent content structure.

The structure of the XML document also maps correspondingly to the structure of the object graph.

© 2014-2015, Ralf Lämmel, rlaemmel@acm.org
Summary of technology modeling

• **Entities** in software development
  • e.g.: Java, Python, J2EE, Django, Testing, Inheritance

• **Entity types** in software development
  • e.g.: Language, Technology, Artifact, Concept

• **Relationships** in software development
  • e.g.
    • HelloWorld.java *elementType* Java
    • Django *uses* Python

• **Relationship types** in software development
  • e.g., „elementType“ or „uses“
Entity types

- **Predefined base types**
  - **Language** — conceptual entities (possibly thought of as sets) for languages
  - **Technology** — conceptual entities for technologies
  - **Artifact** —  „manifested“ / „physical“ entities, e.g., a file
  - **System** — software systems such as information systems or web applications
  - **Function** — mathematical functions on languages or actions
  - **Procedure** — functions with side effects
  - **Action** — units of execution that may have side effects on artifacts
  - **Concept** — programming techniques or other concepts in software development
One could introduce more base types!

- **People** — human beings such as developers, stakeholders, etc.

- **Organization** — enterprises and other kinds organizations

- ...

Before introducing a new base type, make sure that it is not better taken care of as a subtype of a predefined entity type.
Base entity type **Language**

- **Definition:**
  
  - *An artificial language used in software development*

- **Subtypes of** *Language*
  
  - Programming language: *Java, Python, Ruby, …*
  
  - Query language: *XPath, SQL, XQuery, …*
  
  - Transformation language: *XSLT, SQL, ATL, …*
  
  - Modeling language: *UML, SDL, BPMN, …*
More subtype of **Language**

- Hypertext language (HTML)
- Markup language (XML)
- Configuration language
- Annotation language
- Template language
- …
Base entity type Technology

- Definition:
  - A tool (in a very general sense) used in software development

- Subtypes of Technology
  - API and library: JDOM, JQuery, Swing, Tkinter, Twitter API, …
  - Framework: JPA, Hibernate, Spring, Django, …
  - IDE: Visual Studio, Eclipse, NetBeans, …
  - Platform: .NET, Android, J2EE, Java, JRE, …
  - Language processor: javac, python, gcc, …
More subtypes of **Technology**

- Server, e.g., Web server
- Web browser
- Plugins
- Office software
- Operating system
- Package portal, package manager
- App store
- …
Base entity type **System**

- Definition:
  - A deployed, implemented, or designed software system

- Subtype of **System**
  - Information system
  - Web application
  - Web service
  - Mobile app
Base entity type **Artifact**

- **Definition:**
  
  - A „manifested“ / „physical“ entity in a software system

- **Subtypes of Artifact — they all concern „representation“!**
  
  - **File**: files in the common sense of an operating system
  - **Folder**: folders as nested collections of files and folders
  - **Resource**: artifacts addressable / retrievable by URI/URL
  - **Transient**: artifacts arising „temporarily“ by the execution of software
  - **Fragment**: artifacts being part of an artifact
Entity type **Fragment**

• Definition:
  
  • *A part of a software artifact*

• Examples
  
  • Classes in a Python script collecting many classes
  
  • Methods in a class of a Python script
  
  • A CREATE TABLE statement in SQL/DDL script
  
  • A method call in a Python script
def search(l, x):
    return searchInRange(l, x, 0, len(l)-1)

def searchInRange(l, x, min, max):
    if min>max:
        return False
    else:
        middle = min+(max-min)/2
        if x > l[middle]:
            # Search in right half
            return searchInRange(l, x, middle+1, max)
        elif x < l[middle]:
            # Search in left half
            return searchInRange(l, x, min, middle-1)
        else:
            # Found in the middle
            return True
„Conceptual“ subtypes of Artifact

• **Specification** — e.g., the Java Language Specification

• **Standard** — e.g., DVI

• **Request** — e.g., an HTTP REST request

• **Response** — e.g., a response to the said request

• **Model** — in the sense of metamodeling and MDE

• **Metamodel** — in the sense of metamodeling and MDE

• **Grammar** — in the sense of software language engineering

It may happen that an artifact-like entity has two types — a representational one and a conceptual one, e.g.:

```
aGrammar : File, Grammar
```

More generally, an entity may be of multiple types, but the base entity types are disjoint.

© 2014-2015, Ralf Lämmel, rlaemmel@acm.org
Base entity type Function

• Definition:
  • A function defined by an artifact or a technology

• Examples
  • The I/O behavior of a program
  • The I/O behavior of a tool as part of a technology
Base entity type **Procedure**

- **Definition:**
  - An artifact-to-action mapping

- **Examples**
  - The semantics of a Python function
    - when it has side effects
Base entity type **Action**

• **Definition:**
  
  • *A unit of execution possibly affecting artifacts*

• **Examples**
  
  • The meaning of a specific method call in a program
  
  • The meaning of a database update
Base entity type **Concept**

- **Definition:**
  - A concept *from the broad domain of software development*

- **Subtypes of** *Concept*
  - Programming technique: iteration, recursion, etc.
  - Modeling principle: inheritance, composition, etc.
  - Design pattern: Composite, Visitor, etc.
  - Feature: persistence, etc.
  - Protocol: HTTP, etc.
Relationship symbols

- `elementOf` — membership relationship for sets
- `defines` — something executable defining a set or a function
- `↦` — function application
- `subsetOf` — subset relationship
- `precedes` — ordering of actions
- `reads | creates | modifies` — effects of actions
- `partOf` — part-of relationship (composition)
- `dependsOn` — any sort of dependency
- `conformTo` — conformance, e.g., in the sense of schema-based validation
- `correspondsTo` — correspondence, e.g., in the sense of systematic trace links
- `refersTo` — encoded references to entities
- `uses` — usage of concepts or languages
- `implements | supports` — implementation or support of concepts
Relationship types for *composition*

- Artifact `partOf` Artifact — an artifact being part of another artifact
- Artifact `partOf` System — an artifact being part of a system
- Technology `partOf` Technology — a technology being part of another technology
Relationship types for *sets*

- Artifact `elementOf` Language — the language of a file or another artifact

- (Technology | Artifact) `defines` (Language | Function) — sets defined provided by an entity

- Function(Artifact) ↦ Artifact — map an artifact to another artifact

- Language `subsetOf` Language — subset relationship on languages
Relationship types for *actions*

- Procedure(Artifact) ↦ Action — map an artifact to an action
- Action *precedes* Action — constraint on the order of executing two actions
- Action *(reads | creates | modifies)* Artifact — side effect of an action
More relationship types

- Entity `dependsOn` Entity — any sort of dependency

- Artifact `conformTo` Artifact — conformance, e.g., in the sense of schema-based validation

- Artifact `correspondsTo` Artifact — correspondence, e.g., in the sense of systematic trace links

- Artifact `refersTo` Entity — references to an entity encoded in an artifact

- (Technology | Artifact) `uses` | `implements` | `supports` Concept — involved concepts

- (Technology | Artifact) `uses` Language — involved languages
An extended example: A technology model for Django

We use the Polls app as the running example.

Find the code here:

https://github.com/rlaemmel/mysite
Python Web frameworks

- Django
- web2py
- Flask
- Bottle
A low-level view on the Polls app

A database

The model

The view

Python code, HTML, and templates

Templates for views
• db_sqlite3 : File
• mysite : Folder
  • __init__.py : File
  • manage.py : File
• media : Folder
• polls : Folder
  • __init__.py : File
  • admin.py : File
  • models.py : File
  • tests.py : File
  • views.py : File
• settings.py : File
• templates : Folder
  • admin : Folder
  • polls : Folder
    • detail.html : File
    • index.html : File
    • results.html : File
• urls.py : File

We use a example-driven view on Django. We imagine that „any“ Django webapp would have artifacts like the „polls“ app.
• `db_sqlite3` : File $\in$ **SQLITE3-IMAGE** (a language we made up)
• `mysite` : Folder
  • `__init__.py` : File $\in$ **Python**
  • `manage.py` : File $\in$ **Python**
• `media` : Folder
• `polls` : Folder
  • `__init__.py` : File $\in$ **Python**
  • `admin.py` : File $\in$ **Python**
  • `models.py` : File $\in$ **Python**
  • `tests.py` : File $\in$ **Python**
  • `views.py` : File $\in$ **Python**
  • `settings.py` : File $\in$ **Python**
• `templates` : Folder
  • `admin` : Folder
  • `polls` : Folder
    • `detail.html` : File $\in$ **HTML**
    • `index.html` : File $\in$ **HTML**
    • `results.html` : File $\in$ **HTML**
• `urls.py` : File $\in$ **Python**

Languages of artifacts: How useful is that?

Is this HTML, proper?
Starting off the technology model

- Web application < System
- Web application framework < Technology
- Interpreter < Technology
- webapp : Web application
- Django : Web application framework
- Python : Language
- Python interpreter : Interpreter
- webapp uses Python
- webapp uses Django
- Django uses Python

We also assume that all „files“ of the app are entities of the technology model.
Issues

• What is the schema underlying the database image?
• What are the roles of the different python scripts?
• How do code and database relate to each other?
• What technologies are used by the app?
• Aren’t the HTML files using non-HTML constructs?
Issues

• What is the schema underlying the database image?
• What are the roles of the different python scripts?
• How do code and database relate to each other?
• What technologies are used by the app?
• Aren’t the HTML files using non-HTML constructs?
A command to request the DB schema

~ $ pwd
/home/rlaemmel/mysite
~ $ python manage.py sql polls

Response by Django

BEGIN;
CREATE TABLE "polls_poll" (  "id" integer NOT NULL PRIMARY KEY,  "question" varchar(200) NOT NULL,  "pub_date" datetime NOT NULL)
);
CREATE TABLE "polls_choice" (  "id" integer NOT NULL PRIMARY KEY,  "poll_id" integer NOT NULL REFERENCES "polls_poll" ("id"),  "choice" varchar(200) NOT NULL,  "votes" integer NOT NULL)
);
COMMIT;
Command line language for Django administration

- python manage.py sql polls
- python manage.py syncdb

(There exist more such administrative commands.)
Use Django’s CLI for administration to reveal the DB schema

- Django.AdminCLI : Language
- schemaCmd : Transient
- schemaCmd elementOf Django.AdminCLI
- schemaRequ : Action
- schemaResp : Transient
- schemaResp elementOf SQL
- schemaRequ reads db.sqlite3
- schemaRequ creates schemaResp
- CLI based execution : Django.AdminCLI → Action
- Python interpreter defines CLI based execution
- CLI based execution(schemaCmd) ↦ schemaRequ

We assume that there is command language as part of Django.

Command and response are transients here, as we assume that they correspond to program input and output when exercising the CLI.
Issues

• What is the schema underlying the database image?

• **What are the roles of the different python scripts?**

• How do code and database relate to each other?

• What technologies are used by the app?

• Aren’t the HTML files using non-HTML constructs?
Concepts behind the many Python scripts

- **mysite**
  - `__init__.py` implements *Initialization*
  - `manage.py` implements *Administration*
- **polls**
  - `__init__.py` implements *Initialization*
  - `admin.py` implements *View*
  - `models.py` implements *Model*
  - `tests.py` implements *Testing*
  - `views.py` implements *View*
  - `settings.py` implements *Configuration*
  - `urls.py` implements *Router (Routing)*
Concepts

- **Model**: the data / business logics part MVC
- **View**: the user interface part of MVC
- **Router**: a form of controller (part of MVC)
- **Configuration**: configuration of a component or a system
- **Initialization**: initialization of a component or a system
- **Administration**: administration of a system
- **Testing**: test of an artifact or a system
The model

```python
from django.db import models
import datetime

class Poll(models.Model):
    question = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')
    def __unicode__(self):
        return self.question
    def was_published_today(self):
        return self.pub_date.date() == datetime.date.today()
    was_published_today.short_description = 'Published today?'

class Choice(models.Model):
    poll = models.ForeignKey(Poll)
    choice = models.CharField(max_length=200)
    votes = models.IntegerField()
    def __unicode__(self):
        return self.choice
```

Don’t bother about details: these are Python (Django) classes for the business data of the Polls app.
Don’t bother about details, but a typical view loads or saves data, and renders data as HTML via a template.
The view for „admins“ according to Django

```python
class ChoiceInline(admin.TabularInline):
    # Another more spacious option
    # class ChoiceInline(admin.StackedInline):
    model = Choice
    extra = 3

class PollAdmin(admin.ModelAdmin):
    fieldsets = [
        (None, {'fields': ['question']}),
        ('Date information', {'fields': ['pub_date'], 'classes': ['collapse']}),
    ]
    inlines = [ChoiceInline]
    list_display = ('question', 'pub_date', 'was_published_today')
    list_filter = ['pub_date']
    search_fields = ['question']
    date_hierarchy = 'pub_date'

admin.site.register(Poll, PollAdmin)
```

These views are standardized by Django: they allow us to do basic data management for polls and choices.
Routing
(A router maps URLs to views)

```python
from django.conf.urls.defaults import patterns, include, url
from django.contrib import import admin
admin.autodiscover()

urlpatterns = patterns(',
    url(r'^polls/$', 'mysite.polls.views.index'),
    url(r'^polls/(?P<poll_id>[\d]+)/$', 'mysite.polls.views.detail'),
    url(r'^polls/(?P<poll_id>[\d]+)/results/$', 'mysite.polls.views.results'),
    url(r'^polls/(?P<poll_id>[\d]+)/vote/$', 'mysite.polls.views.vote'),
    url(r'^admin/', include(admin.site.urls)),
)
```

Regular expression for URLs with parameters

Python function for a particular view
Issues

• What is the schema underlying the database image?

• What are the roles of the different python scripts?

• **How do code and database relate to each other?**

• What technologies are used by the app?

• Aren’t the HTML files using non-HTML constructs?
Request of database sync via CLI

~ $ pwd
/home/rlaemmel/mysite
~ $ python manage.py syncdb

Creating tables ... 
Creating table auth_permission

... 
Creating table django_admin_log
Creating table polls_poll
Creating table polls_choice

You just installed Django's auth system, which means you don't have any superusers defined. 
Would you like to create one now? (yes/no): yes 
Username (Leave blank to use 'rlaemmel'): rlaemmel 

... 
Superuser created successfully.
Installing custom SQL ...
Installing indexes ... 
No fixtures found.

This is basically just informative text produced by the admin functionality to report on database changes.
Use Django’s CLI for administration to sync model code with database

- Text : Language
- syncCmd : Transient
- syncCmd `elementOf` Django.AdminCLI
- syncRequ : Action
- syncResp : Transient
- syncResp `elementOf` Text
- syncRequ `reads` mysite/polls/models.py
- syncRequ `modifies` db.sqlite3
- syncRequ `creates` syncResp
- CLI based execution(syncCmd) ↦ syncRequ

This is similar to the CLI use for requesting the database schema, but this time the database is also modified and the output is text rather than SQL.
Correspondence between model code and database schema

Relationships:
- syncRequ reads mysite/polls/models.py
- syncRequ reads db.sqlite3
- syncRequ modifies db.sqlite3
- schemaResp corresponds to mysite/polls/models.py

```sql
CREATE TABLE "polls_poll" (  
    ...  
) ;
CREATE TABLE "polls_choice" (  
    ...  
)
```

```python
class Poll(models.Model):
    ...  
class Choice(models.Model):
    ...  
```
Issues

• What is the schema underlying the database image?
• What are the roles of the different python scripts?
• How do code and database relate to each other?
• What technologies are used by the app?
• Aren’t the HTML files using non-HTML constructs?
Referenced python modules

- Runtime < Technology
- Template processor < Technology
- Protocol < Concept
- PythonRuntime : Runtime
- os : Library
- datetime : Library
- Django.db : Library
- Django.test : Library
- Django.template : Template processor
- Django.http : Library
- Database access : Concept
- Testing : Concept
- Template processing : Concept

• HTTP : Protocol
• webapp uses os
• webapp uses datetime
• webapp uses Django.db
• webapp uses Django.test
• webapp uses Django.template
• webapp uses Django.http
• datetime partOf PythonRuntime
• os partOf PythonRuntime
• Django.db supports Database access
• Django.test supports Testing
• Django.http supports HTTP

This naming convention introduces parts.
Issues

• What is the schema underlying the database image?
• What are the roles of the different python scripts?
• How do code and database relate to each other?
• What technologies are used by the app?

• Aren’t the HTML files using non-HTML constructs?
The template for the *index* view

```html
{% if latest_poll_list %}
  <ul>
    {% for poll in latest_poll_list %}
      <li><a href="/polls/{{ poll.id }}">{{ poll.question }}</a></li>
    {% endfor %}
  </ul>
{% else %}
  <p>No polls are available.</p>
{% endif %}
```
The template for the *detail* view

```html
<h1>{{ poll.question }}</h1>

{% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}

<form action="/polls/{{ poll.id }}/vote/" method="post">
{% csrf_token %}
{% for choice in poll.choice_set.all %}
    <input type="radio" name="choice" id="choice{{ forloop.counter }}" value="{{ choice.id }}" />
    <label for="choice{{ forloop.counter }}">{{ choice.choice }}</label><br />
{% endfor %}
<input type="submit" value="Vote" />
</form>
```
The template for the *results* view

```html
<h1>{{ poll.question }}</h1>

<ul>
{% for choice in poll.choice_set.all %}
  <li>{{ choice.choice }} -- {{ choice.votes }} vote{{ choice.votes|pluralize }}</li>
{% endfor %}
</ul>

<a href="/polls/{{ poll.id }}">Vote again?</a>
```
A language for templates

- We designate a language \texttt{Django.Templ} $\supset$ HTML.

- Django.Templ offers extra constructs like this:
  - Python expressions {{ … }} evaluating to HTML
  - Loops over Python data to return HTML

- MegaL declarations:
  - Template language $<$ Language
  - \texttt{Django.Templ} : Template language
  - HTML \texttt{subsetOf} Django.Templ
  - \texttt{mysite/templates/polls/*.html} \texttt{elementOf} Django.Templ

We use this notation as a short cut to refer to many entities in an obvious manner.
The End