Megamodels of software systems

(Model of the linguistic architecture)

Prof. Dr. Ralf Lämmel
University of Koblenz-Landau
Faculty of Computer Science
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. megamodels

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

• **Mega** models
  • Languages, technologies, artifacts in a system
  • 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 megamodel)
A **behavioral** model for a metro’s turnstile
(A software model — not a megamodel)
Examples of megamodels

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

We use the **MegaL language** for megamodeling. 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); execution is not covered by the following megamodel.)
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 — the programming language Java
• JavaByteCode : Language — the bytecode language of Java/JVM
• Compiler < Technology
• aJavaProgram : Artifact — e.g., HelloWorld.java (placeholder)
• aJavaByteCodeProgram : Artifact — e.g., HelloWorld.class (placeholder)
• aJavaProgram ∈ Java
• aJavaByteCodeProgram ∈ JavaByteCode
• compilation : Java → JavaByteCode
• javac implements compilation
• compilation(aJavaProgram) ↦ 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**: Artifact — e.g., HelloWorld.java
- **aJavaByteCodeProgram**: Artifact — e.g., HelloWorld.class
- **aJavaProgram ∈ Java**
- **aJavaByteCodeProgram ∈ JavaByteCode**
- **compilation**: Java → JavaByteCode
  - compilation is a function entity.
- **javac implements compilation**
- **compilation(aJavaProgram) ↦ aJavaByteCodeProgram**
Compilation with Java’s javac compiler

• Java : Language
• JavaByteCode : Language
• Compiler < Technology
• javac : Compiler
• aJavaProgram : Artifact — e.g., HelloWorld.java
• aJavaByteCodeProgram : Artifact — e.g., HelloWorld.class
• aJavaProgram ∈ Java
• aJavaByteCodeProgram ∈ JavaByteCode
• compilation : Java → JavaByteCode
• javac implements compilation
• compilation(aJavaProgram) ↦ aJavaByteCodeProgram
Compilation with Java’s javac compiler

- Java : Language
- JavaByteCode : Language
- Compiler < Technology
- javac : Compiler
- aJavaProgram : Artifact — e.g., HelloWorld.java
- aJavaByteCodeProgram : Artifact — e.g., HelloWorld.class
  - aJavaProgram ∈ Java
  - aJavaByteCodeProgram ∈ JavaByteCode
- compilation : Java → JavaByteCode
- javac implements compilation
- compilation(aJavaProgram) ↦ aJavaByteCodeProgram
The RDBMS shell (a workbench or monitor) is used to populate a relational database. To this end, the database schema (some CREATE TABLE statements) are executed, followed by another script with sample data (some INSERT statements).

We will be using MySQL as the RDBMS. 
https://www.mysql.com/
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,  
address 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

```sql
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 = 3
```

# Employees

```sql
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 ≺ SQL —- all CREATE TABLE st.
- SQL DML ≺ SQL —- all CRUD statements
- RDBMS < Technology
- MySQL : RDBMS
- IDE < Technology
- MySQL Workbench : IDE

- createStmts : Artifact
- insertStmts : Artifact
- createStmts ∈ SQL DDL
- insertStmts ∈ SQL DML
- db₁, db₂, db₃ : Artifact
- DbImage : Language
- db₁, db₂, db₃ ∈ DbImage
- execution : SQL × DbImage → DbImage
- MySQL Workbench implements execution
- execution(createStmts, db₁) ↦ db₂
- execution(insertStmts, db₂) ↦ db₃
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 ⊆ SQL
- SQL DML ⊆ SQL
- RDBMS < Technology
- MySQL : RDBMS
- IDE < Technology
- MySQL Workbench : IDE
- createStmts : Artifact
- insertStmts : Artifact
- createStmts ∈ SQL DDL
- insertStmts ∈ SQL DML
- db₁, db₂, db₃ : Artifact
- DbImage : Language
- db₁, db₂, db₃ ∈ DbImage
- execution : SQL × DbImage → DbImage
- MySQL Workbench implements execution
- execution(createStmts, db₁) ↦ db₂
- execution(insertStmts, db₂) ↦ db₃

Taking apart SQL into DDL and DML
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 ⊆ SQL
- SQL DML ⊆ SQL
- RDBMS < Technology
- MySQL : RDBMS
- IDE < Technology
- MySQL Workbench : IDE
- createStmts : Artifact
- insertStmts : Artifact
- createStmts ∈ SQL DDL
- insertStmts ∈ SQL DML
- db₁, db₂, db₃: Artifact

**Identification of involved technologies**

- DbImage : Language
- db₁, db₂, db₃ ∈ DbImage
- execution : SQL × DbImage → DbImage
- MySQL Workbench implements execution
- execution(createStmts, db₁) ↦ db₂
- execution(insertStmts, db₂) ↦ db₃

© 2016 Ralf Lämmel
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 ⊆ SQL
- SQL DML ⊆ SQL
- RDBMS < Technology
- MySQL : RDBMS
- IDE < Technology
- MySQL Workbench : IDE

Identification of involved artifacts, also subject to a language for the database image

- createStmts : Artifact
- insertStmts : Artifact
- createStmts ∈ SQL DDL
- insertStmts ∈ SQL DML
- db₁, db₂, db₃ : Artifact — state of database
- DbImage : Language — MySQL format
- db₁, db₂, db₃ ∈ DbImage
- execution : SQL × DbImage → DbImage
- MySQL Workbench implements execution
- execution(createStmts, db₁) ↦ db₂
- execution(insertStmts, db₂) ↦ db₃
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 ⊆ SQL
- SQL DML ⊆ SQL
- RDBMS < Technology
- MySQL : RDBMS
- IDE < Technology
- MySQL Workbench : IDE

Script execution with the help of the workbench resulting in changed images

- createStmts : Artifact
- insertStmts : Artifact
- createStmts ∈ SQL DDL
- insertStmts ∈ SQL DML
- db₁, db₂, db₃: Artifact
- DbImage : Language
- db₁, db₂, db₃ ∈ DbImage
- execution : SQL × DbImage → DbImage
- MySQL Workbench implements execution
- execution(createStmts, db₁) ↦ db₂
- execution(insertStmts, db₂) ↦ db₃
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 megamodel for JAXB

**XML-data binding of the Java platform**

<table>
<thead>
<tr>
<th>Category</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform &lt; Technology</td>
<td>Java platform</td>
</tr>
<tr>
<td>XML data binding technology &lt; Technology</td>
<td>JAXB</td>
</tr>
<tr>
<td>Code generator &lt; Technology</td>
<td>bindingCompiler</td>
</tr>
<tr>
<td>Library &lt; Technology — or framework or API</td>
<td>bindingFramework</td>
</tr>
</tbody>
</table>

- **Java platform**: Platform
- **Java**: Language
- **Java platform implements Java**

- **JAXB**: XML data binding technology
- **JAXB partOf Java platform**
- **bindingCompiler**: Code generator
- **bindingFramework**: Library
- **bindingCompiler partOf JAXB**
- **bindingFramework partOf JAXB**

### Part 1/5: Technology break-down

- The types of technologies involved
- The overall Java platform which JAXB is part of
- The relevant parts of JAXB
  - [https://jaxb.java.net/2.2.4/docs/xjc.html](https://jaxb.java.net/2.2.4/docs/xjc.html)

© 2016 Ralf Lämmel
A megamodel for JAXB (XML-data binding of the Java platform)

<table>
<thead>
<tr>
<th>XML : Language — all of XML</th>
</tr>
</thead>
<tbody>
<tr>
<td>XSD : Language — XML schema</td>
</tr>
<tr>
<td>XSD ⊆ XML — schemas are written in XML</td>
</tr>
<tr>
<td>JAXB.Java : Language</td>
</tr>
<tr>
<td>JAXB.Java ⊆ Java</td>
</tr>
<tr>
<td>generation : XSD → JAXB.Java</td>
</tr>
</tbody>
</table>

- xmlTypes : File+
- xmlTypes ∈ XSD
- javaClasses : File+
- javaClasses ∈ JAXB.Java
- bindingCompiler implements 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 „+“.)

Part 2/5: Type-level mapping including data flow
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
    elementFormDefault="qualified" targetNamespace="http://www.company.softlang.org/company.xsd">

    <xs:element name="company">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="name"/>
                <xs:element maxOccurs="unbounded" minOccurs="0" ref="department"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:element name="department">
        <xs:complexType>
            <xs:sequence>
                <xs:element ref="name"/>
                <xs:element name="manager" type="employee"/>
                <xs:element maxOccurs="unbounded" minOccurs="0" ref="department"/>
                <xs:element maxOccurs="unbounded" minOccurs="0" name="employee" type="employee"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

    <xs:complexType name="employee">
        <xs:sequence>
            <xs:element ref="name"/>
            <xs:element ref="address"/>
            <xs:element ref="salary"/>
        </xs:sequence>
    </xs:complexType>

</xs:schema>
Generated Java code for “companies” (simplified)

```java
public class Company {
    protected String name;
    protected List<Department> department;
    public String getName() { return name; }
    public void setName(String value) { this.name = value; }
    public List<Department> getDepartment() {
        if (department == null) {
            department = new ArrayList<Department>();
        }
        return this.department;
    }
}
```
Generated Java code for “companies”
package org.softlang.company.xjc;

import java.util.ArrayList;
import java.util.List;
import javax.xml.bind.annotation.XmlAccessType;
import javax.xml.bind.annotation.XmlAccessorType;
import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
import javax.xml.bind.annotation.XmlType;

/**
 * <p>Java class for anonymous complex type.
 * 
 * <p>The following schema fragment specifies the expected content contained within this class.
 * 
 * <pre>
 * <complexType>
 *   <complexContent>
 *     <restriction base="{http://www.w3.org/2001/XMLSchema}anyType">
 *       <sequence>
 *         <element ref="{http://www.company.softlang.org/company.xsd}name"/>
 *         <element ref="{http://www.company.softlang.org/company.xsd}department" maxOccurs="unbounded" minOccurs="0"/>
 *       </sequence>
 *     </restriction>
 *   </complexContent>
 * </complexType>
 * </pre>
 * 
 * The comment is the "watermark" by the binding compiler.
 * These imports also concern the binding framework.
 */
/**
 * Java class for anonymous complex type.
 *
 * The following schema fragment specifies the expected content contained within this class.
 *
 * &lt;complexType&gt;
 * &lt;complexContent&gt;
 * &lt;restriction base="{http://www.w3.org/2001/XMLSchema}anyType}&gt;
 * &lt;sequence&gt;
 * &lt;element ref="{http://www.company.softlang.org/company.xsd}name"/&gt;
 * &lt;element ref="{http://www.company.softlang.org/company.xsd}department" maxOccurs="unbounded" minOccurs="0"/&gt;
 * &lt;/sequence&gt;
 * &lt;/restriction&gt;
 * &lt;/complexContent&gt;
 * &lt;/complexType&gt;
 * &lt;/pre&gt;
 *
 * @XmlAccessorType(XmlAccessType.FIELD)
 * @XmlType(name = "", propOrder = {
 * "name",
 * "department"
 * })
 * @XmlRootElement(name = "company")
 * public class Company {

 * @XmlElement(required = true)
 * protected String name;
 * protected List&lt;Department&gt; department;

 * Gets the value of the name property.
 *
@XmlElement(required = true)
protected String name;
protected List<Department> department;

/**
 * Gets the value of the name property.
 */
public String getName() {
    return name;
}

/**
 * Sets the value of the name property.
 */
public void setName(String value) {
    this.name = value;
}

These annotations are used for validation and serialization.
@XmlElement(required = true)
protected String name;
protected List<Department> department;

/**
 * Gets the value of the name property.
 */
public String getName() {
    return name;
}

/**
 * Sets the value of the name property.
 */
public void setName(String value) {
    this.name = value;
}

/**
 * Gets the value of the department property.
 */
public String getDepartment() {
    return department.toString();
}

/**
 * Sets the value of the department property.
 */
public void setDepartment(List<Department> value) {
    this.department = value;
}

/*
 * This accessor method returns a reference to the live list,
 * not a snapshot. Therefore any modification you make to the
 * returned list will be present inside the JAXB object.
 * WARNING: The list returned by this method is NOT clone
 */
public void setName(String value) {
    this.name = value;
}

/**
 * Gets the value of the department property.
 *
 * <p>
 * This accessor method returns a reference to the live list,
 * not a snapshot. Therefore any modification you make to the
 * returned list will be present inside the JAXB object.
 * This is why there is not a &lt;CODE&gt;set&lt;/CODE&gt; method for the department property.
 * 
 * <p>
 * For example, to add a new item, do as follows:
 * <pre>
 *    getDepartment().add(newItem);
 * </pre>
 * 
 * <p>
 * Objects of the following type(s) are allowed in the list
 * {@link Department }
 * 
 * <p>
 */
public List<Department> getDepartment() {
    if (department == null) {
        department = new ArrayList<Department>();
    }
    return this.department;
}
A megamodel for JAXB (XML-data binding of the Java platform)

- Annotation: Concept — here: Java annotations used as metadata for Java code
- bindingFramework facilitates Annotation — binding framework exports JAXB annotations
- javaClasses uses Annotation — generated classes use JAXB annotations
- Serialization: Concept — here: object serialization (aka un-/marshalling)
  - bindingFramework facilitates Serialization — binding framework used for serialization
- Code generation: Concept — here: the generation of Java code from XML schemas
  - bindingCompiler facilitates Code generation — binding compiler used for code generation
  - bindingCompiler uses Code generation — binding compiler uses code generation itself
- Validation: Concept — schema-based validation or conformance checking
  - JAXB facilitates Validation — Validation is a byproduct of XML-data binding

Part 3/5: Involved concepts
### A megamodel for JAXB
(XML-data binding of the Java platform)

#### Part 4/5: Instance-level mapping including data flow

- **JVM.ObjectGraphs**: Language
- **anObjectGraph**: Transient
- **anObjectGraph ∈ JVM.ObjectGraphs**

- **anXmlDoc**: File
- **anXmlDoc ∈ XML**
- **unmarshalling**: XML → JVM.ObjectGraphs
- **unmarshalling(anXmlDoc) ↦ anObjectGraph**

- **application**: File+
- **application ∈ Java**
- **aMethodCall**: Fragment
- **aMethodCall partOf application**
- **aMethodCall uses bindingFramework**
- **(aMethodCall refersTo bindingFramework)**
- **aMethodCall defines unmarshalling**

---

The „language“ for run-time object graphs

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

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

The unmarshalling function arises as the meaning of a code fragment that is part of the application that uses JAXB. That code clearly uses and refers to the JAXB library.
Java code of application for de-serializing “companies”

```java
public static Company deserializeCompany(File input) throws JAXBException {
    initializeJaxbContext();
    Unmarshaller unMarshaller = jaxbContext.createUnmarshaller();
    return (Company) unMarshaller.unmarshal(input);
}
```
Java code of application for **serializing** “companies”

```java
public static void serializeCompany(File output, Company c) throws JAXBException, FileNotFoundException, XMLStreamException {
    initializeJaxbContext();
    OutputStream os = new FileOutputStream(output);
    Marshaller marshaller = jaxbContext.createMarshaller();
    XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
    XMLStreamWriter writer = outputFactory.createXMLStreamWriter(os);
    marshaller.marshal(c, writer);
}
```
A megamodel for JAXB
(XML-data binding of the Java platform)

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

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!
XML schema types and Java generated classes are very “similar” in structure — we call this correspondence.
XML document and object graph obtained by deserialization are also very “similar” in structure — this is another instance of correspondence.

Part 5/5: Conformance and correspondence

© 2016 Ralf Lämmel
There is a definition of companies, departments, and employees on both sides. The structures aren’t completely equal, of course.
XML schema-to-generated Java classes correspondence

XML schema

```xml
<xs:element name="company">
  <xs:element name="department">
    <xs:complexType name="employee">
      <xs:sequence>
        <xs:element ref="name"/>
        <xs:element ref="address"/>
        <xs:element ref="salary"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:complexType>
<xs:element name="name" type="xs:string"/>
<xs:element name="address" type="xs:string"/>
<xs:element name="salary" type="xs:double"/>
```

Java

```
Company.java
Department.java
Employee.java

Employee
- address
- name
- salary
  - getAddress() : String
  - getName() : String
  - getSalary() : double
  - setAddress(String) : void
  - setName(String) : void
  - setSalary(double) : void
```

The similarity continues at the level of the individual types with interesting differences. For instance, we face getters and setters in Java.
Summary of megamodeling

- **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 $\in$ Java
    - Django *uses* Python

- **Relationship types** in software development
  - e.g., „$\in$“ or „*uses““

Think of relationships as edges in a graph with entities as the nodes.
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** — a conglomeration of artifacts making up a system
  - **Function** — mathematical functions on languages or actions
  - **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 subtypes of **Language**

- Hypertext language: HTML, ...
- Markup language: XML, ...
- Configuration language
- Annotation language
- Template language
- ...

© 2016 Ralf Lämmel
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 (platform), JRE, …*
  
  • Language processor: *javac, python, gcc, …*
More subtypes of Technology

• Server, e.g., Web server
• Web browser
• Plugin
• Office software
• Operating system
• Package portal
• Package manager
• App store
• ...
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
Base entity type **System**

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

- Subtype of **System**
  - Information system
  - Web application
  - Web service
  - Mobile app
Entity type **Fragment**

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

- **Examples**
  - Methods in a class of a Python script
  - A method call in a Python script
  - A CREATE TABLE statement in a SQL/DDL script
Source code as a nested container

```python
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., software and systems engineering standards
  
- **Request** — e.g., an HTTP REST request
- **Response** — e.g., a response to the said request
- **Model** — in the sense of metamodelling and MDE
- **Metamodel** — in the sense of metamodelling and MDE
- ...

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

```
aRequest: Transient, Request
```

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

- **Definition:**
  - A function on *languages* for domain and range defined by an artifact or implemented by 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 **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

- $\in$ — membership relationship for languages
- defines — something defining a language or a function
- implements — something implementing a language or a function
- $\rightarrow$ — function application (data flow)
- $\subseteq$ — subset relationship on languages
- partOf — part-of relationship (composition)
- uses — usage of languages, technologies, and concepts
- facilitates — facilitation (support for) usage
- refersTo — encoded references to entities
- conformsTo — conformance, e.g., in the sense of schema-based validation
- correspondsTo — correspondence, i.e., “systematic” similarity
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

• Language partOf Technology — a language being part of a technology
Relationship types for *languages*

- Artifact $\in$ Language — the language of an artifact

- Artifact *defines* (Language | Function) — languages or functions defined by artifacts

- (Artifact | Technology) *implements* (Language | Function) — … implemented by technologies

- Function(Artifact) $\mapsto$ Artifact — map an artifact to another artifact ("data flow")

- Language $\subseteq$ Language — subset relationship on languages
More relationship types

• Artifact **conformsTo** Artifact — conformance, e.g., in the sense of schema-based validation

• Artifact **correspondsTo** Artifact — correspondence, i.e., “systematic” similarity

• (Artifact | System) **refersTo** Entity — references to an entity encoded in an artifact

• (Technology | Artifact | System) (**uses** | **facilitates**) (Language | Technology | System | Concept)

• (Technology | Artifact | System) **implements** Concept
Web application development with **Django** — explained by means of a megamodel

We use the Polls app as the running example.

Find the code here:  
[https://github.com/rlaemmel/mysite](https://github.com/rlaemmel/mysite)

A deployment MAY be available here:  
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.

Types of artifacts: How useful is that?
- \text{db\_sqlite3} : \text{File} \in \text{SL3IMG} \text{ (a language we made up)}
- \text{mysite} : \text{Folder}
  - \text{\_init\_.py} : \text{File} \in \text{Python}
  - \text{manage.py} : \text{File} \in \text{Python}
  - \text{media} : \text{Folder}
  - \text{polls} : \text{Folder}
    - \text{\_init\_.py} : \text{File} \in \text{Python}
    - \text{admin.py} : \text{File} \in \text{Python}
    - \text{models.py} : \text{File} \in \text{Python}
    - \text{tests.py} : \text{File} \in \text{Python}
    - \text{views.py} : \text{File} \in \text{Python}
    - \text{settings.py} : \text{File} \in \text{Python}
  - \text{templates} : \text{Folder}
    - \text{admin} : \text{Folder}
    - \text{polls} : \text{Folder}
      - \text{detail.html} : \text{File} \in \text{HTML}
      - \text{index.html} : \text{File} \in \text{HTML}
      - \text{results.html} : \text{File} \in \text{HTML}
      - \text{urls.py} : \text{File} \in \text{Python}
The beginning of a megamodel for Django

• 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 retrieve the DB schema

- SL3IMG: Language
- db.sqlite3 : File
- db.sqlite3 ∈ SL3IMG
- Django.AdminCLI : Language
- getSchema : Transient
- getSchema ∈ Django.AdminCLI
- SQL : Language
- schema : Transient
- schema ∈ SQL
- retrieval : Django.AdminCLI × SL3IMG → SQL
- retrieval(getSchema, db.sqlite3) ↦ schema
- Python interpreter defines retrieval

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

Command and schema 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)*

We leave out *declarations* of the concepts.
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.
The view for „end users“

def index(request):
    latest_poll_list = Poll.objects.all().order_by('-pub_date')[:5]
    t = loader.get_template('polls/index.html')
    c = Context({'latest_poll_list': latest_poll_list,})
    return HttpResponse(t.render(c))

def detail(request, poll_id):
    p = get_object_or_404(Poll, pk=poll_id)
    return render_to_response('polls/detail.html', {'poll': p},
            context_instance=RequestContext(request))

def results(request, poll_id):
    p = get_object_or_404(Poll, pk=poll_id)
    return render_to_response('polls/results.html', {'poll': p})

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)

```
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)),
)
```
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?
How do code (model) and database relate to each other?

schema correspondsTo mysite/polls/models.py
Request of database sync via CLI

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

Response by Django

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

- db.sqlite3_1, db.sqlite3_2 : File
- db.sqlite3_1, db.sqlite3_2 ∈ SL3IMG
- syncdb : Transient
- syncdb ∈ Django.AdminCLI
- syncdb refersTo mysite/polls/models.py
- modification : Django.AdminCLI × SL3IMG → SL3IMG
- modification(syncdb, db.sqlite3_1) ↦ db.sqlite3_2
- Python interpreter defines modification

This is similar to the CLI use for requesting the database schema, but this time the database is modified.
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 facilitates Database access
- Django.test facilitates Testing
- Django.http facilitates 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 `Django.Templ` ⊃ `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`
  • `Django.Templ : Template language`
  • `HTML` ⊂ `Django.Templ`
  • `mysite/templates/polls/*.html` ∈ `Django.Templ`

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