Model handling with EMF

An introduction to the Eclipse Modeling Framework

ATLAS group (INRIA & LINA),
University of Nantes
France
http://www.sciences.univ-nantes.fr/lina/atl/

Context of this work

- The present courseware has been elaborated in the context of the MODELWARE European IST FP6 project (http://www.modelware-ist.org/).
- Co-funded by the European Commission, the MODELWARE project involves 19 partners from 8 European countries. MODELWARE aims to improve software productivity by capitalizing on techniques known as Model-Driven Development (MDD).
- To achieve the goal of large-scale adoption of these MDD techniques, MODELWARE promotes the idea of a collaborative development of courseware dedicated to this domain.
- The MDD courseware provided here with the status of open source software is produced under the EPL 1.0 license.

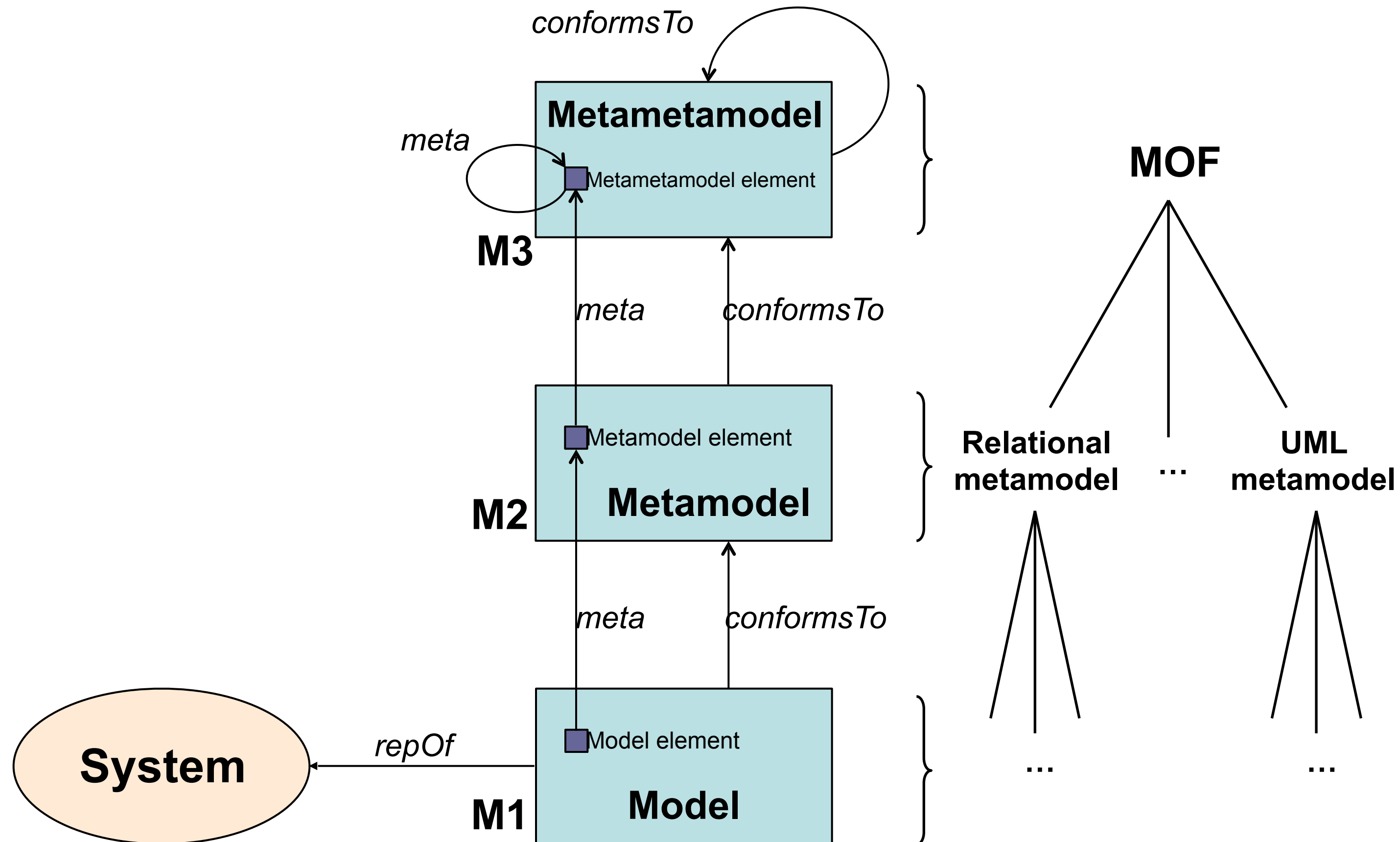
Outline

- Model-Driven Engineering
  - Principles
  - Model-Driven Architecture
  - Operating on models
- The Eclipse platform
- Handling models with EMF

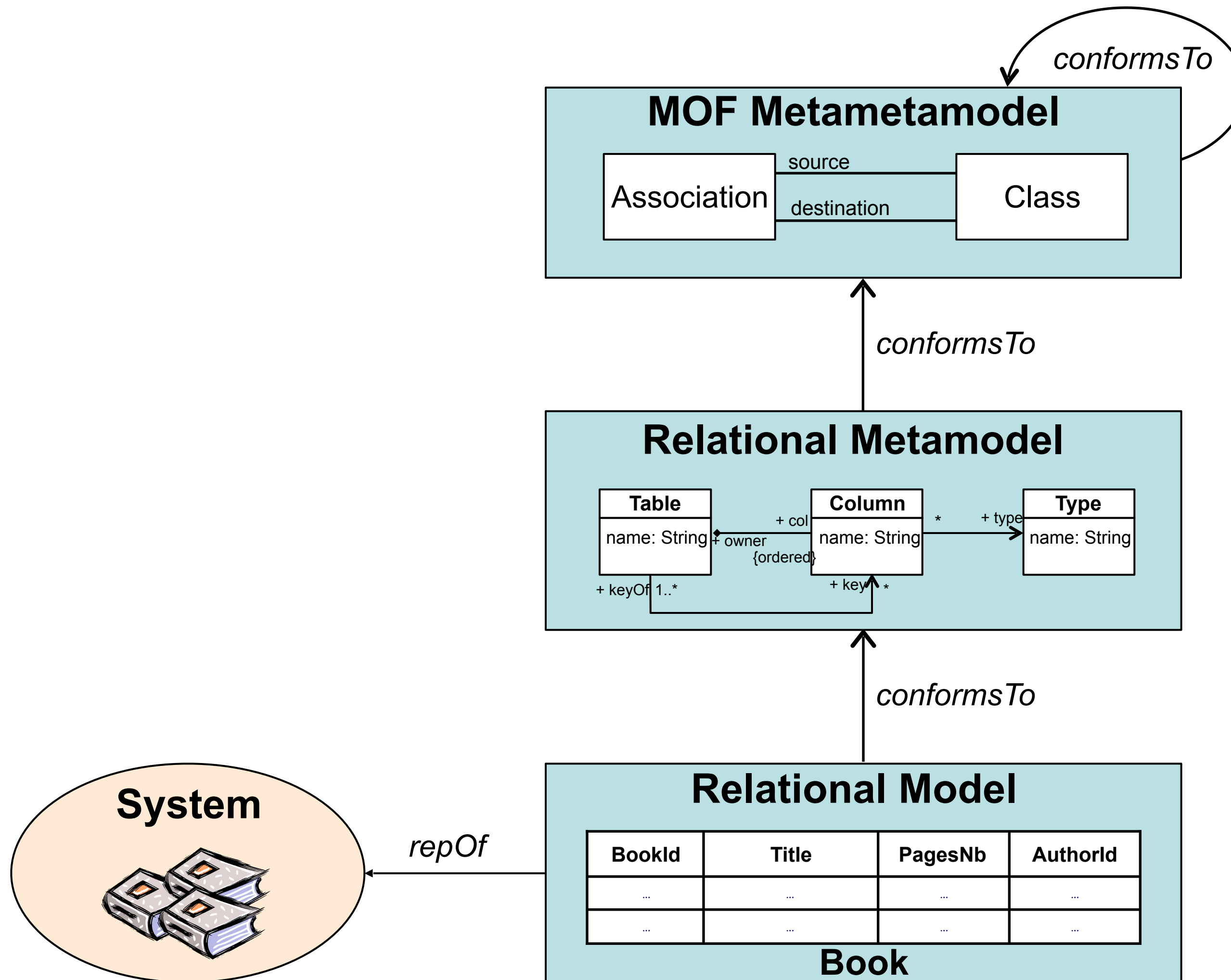
MDE Principles

- Current engineering approaches
  - Models not part of engineering processes (documentation)
- MDE approach
  - Models as first class entities
  - Need for dedicated tools
- MDE basic concepts
  - System – real world situation
  - Model – abstraction of a system
    - Describe a given aspect of the system
  - Metamodel – rules to define an abstraction

Model-Driven Architecture



Model-Driven Architecture: Example



Operating on Models

- Model persistence
  - Loading/saving
- Model edition
  - Creation/deletion/modification
- Model navigation
  - conformsTo/meta relations

Outline

- Model-Driven Engineering
- <u>The Eclipse platform</u>
  - Plug-in architecture
  - Platform architecture
- Handling models with EMF

What is Eclipse?

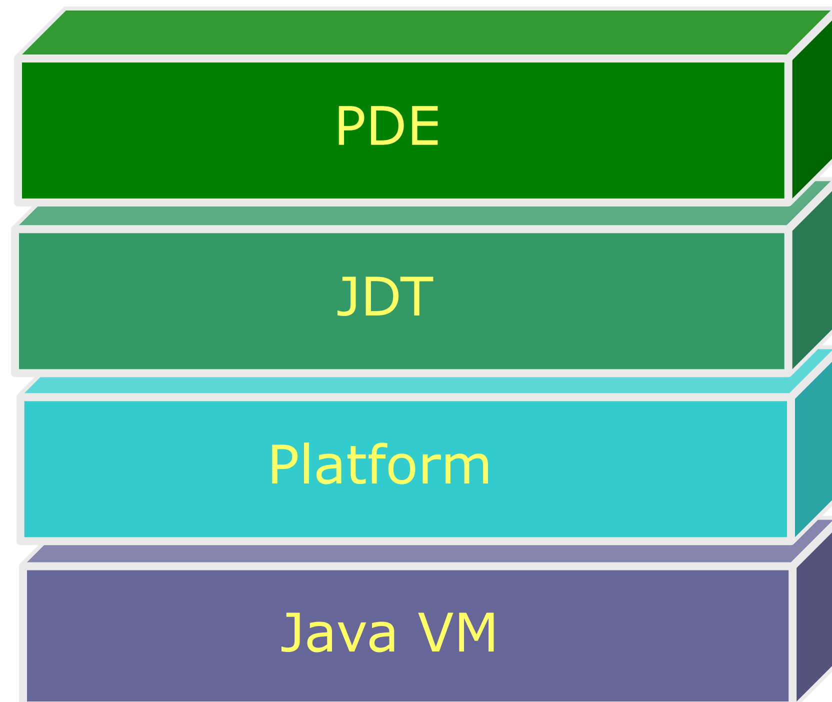
- Eclipse is a universal platform for integrating development tools
- Open, extensible architecture based on plug-ins

Plug-in development environment

Java development tools

Eclipse Platform

Standard Java2 Virtual Machine



Eclipse Plug-in Architecture (1/2)

- **Plug-in** - smallest unit of Eclipse function
  - Big example: HTML editor
  - Small example: Action to create zip files
- **Extension point** - named entity for collecting "contributions"
  - Example: extension point for workbench preference UI
- **Extension** - a contribution
  - Example: specific HTML editor preferences

Eclipse Plug-in Architecture (2/2)

- Each plug-in
  - Contributes to 1 or more extension points
  - Optionally declares new extension points
  - Depends on a set of other plug-ins
  - Contains Java code libraries and other files
  - May export Java-based APIs for downstream plug-ins
  - Lives in its own plug-in subdirectory
- Details spelled out in the **plug-in manifest**
  - Manifest declares contributions
  - Code implements contributions and provides API
  - plugin.xml file in root of plug-in subdirectory

Eclipse Platform Architecture

- Eclipse Platform Runtime is micro-kernel
  - All functionality supplied by plug-ins
- Eclipse Platform Runtime handles start up
  - Discovers plug-ins installed on disk
  - Matches up extensions with extension points
  - Builds global plug-in registry
  - Caches registry on disk for next time

Outline

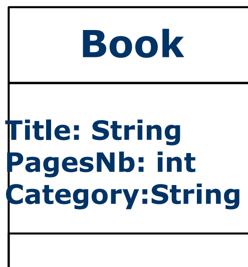
- Model-Driven Engineering
- The Eclipse platform
- Handling models with EMF
  - EMF framework
  - Model persistence
  - Model edition & navigation

EMF Framework (1/2)

- Object oriented framework
  - **Edition** of new EMF models
    - Ecore metametamodel
    - XMI as canonical representation
  - **Import** of existing models
    - UML model
    - XML schema
    - annotated Java code
  - **Export** of Ecore-based models
    - Java code generation
      - Accessor methods
      - Operations skeletons

EMF Framework (2/2)

- Generation of programmatic facilities for generic model handling



- Model package class – accessing model metadata

```
EClass bookClass = libraryPackage.getBook();

EAttribute titleAttribute =
        libraryPackage.getBook_Title();
```

- Model factory class – instantiating modeled classes

```
Book myBook = libraryFactory.createBook();
```

Model Persistence

- EMF Persistence Interface
  - URI (Uniform Resource Identifier)
    - Standard for resource identification/location
  - Resource
    - Common interface for different storage types
  - ResourceSet
    - Interface for Resource collections

Model Persistence: URI Interface

- URI = three parts string
  - Scheme
    - "file", "jar", "platform", etc.
  - Scheme-specific part
    - is scheme-dependent
  - Fragment
    - identifies a part of the contents of the resource specified by the scheme
- Examples
  - The "library.xml" resource
    - platform:/resource/project/library.xml
  - First book in the "library.xml" resource
    - platform:/resource/project/library.xml#//@books.0

Model Persistence: Resource Interface (1/2)

- EMF Resource interface
  - Common API for different storage types
  - Persistent container of EObjects
    - EObject = base object of the EMF framework
- Resource location identified by an URI
  - Resource types identified by URI extensions
- EMF implementations
  - XMLResource
  - XMIResource (default)

Model Persistence: Resource Interface (2/2)

- Resource interface (excerpt)
  - Interactions with persistent storage

```
void load(Map /*option*/);
void save(Map /*option*/);
void unload();
```

  - Updating contents

```
void getContents().add(Object);
void getContents().remove(Object);
```

  - Accessing contents

```
EObject getEObject(String /*URIFragment*/);
String    getURIFromEObject(EObject);
```

Model Persistence: ResourceSet Interface

- ResourceSet = collection of Resources
  - Resources created or loaded together
  - Resource allocation
  - Automatic loading of cross-referenced Resources

- ResourceSet interface (excerpt)
  - Empty Resource creation

```
Resource createResource(URI);
```

  - Accessing contents

```
Resource getResource(URI, boolean /*loadOnDemand*/);
EObject   getEObject(URI, boolean /*loadOnDemand*/);
```

Model Persistence: Example

- Saving to persistent storage

```
Book myBook = new Book(…);
…
ResourceSet myResourceSet = new ResourceSetImpl();
Resource myResource =
        myResourceSet.createResource(URI.createURI("library.xml"));
myResource.getContents().add(myBook);
myResource.save(null);
```

- Loading from persistent storage

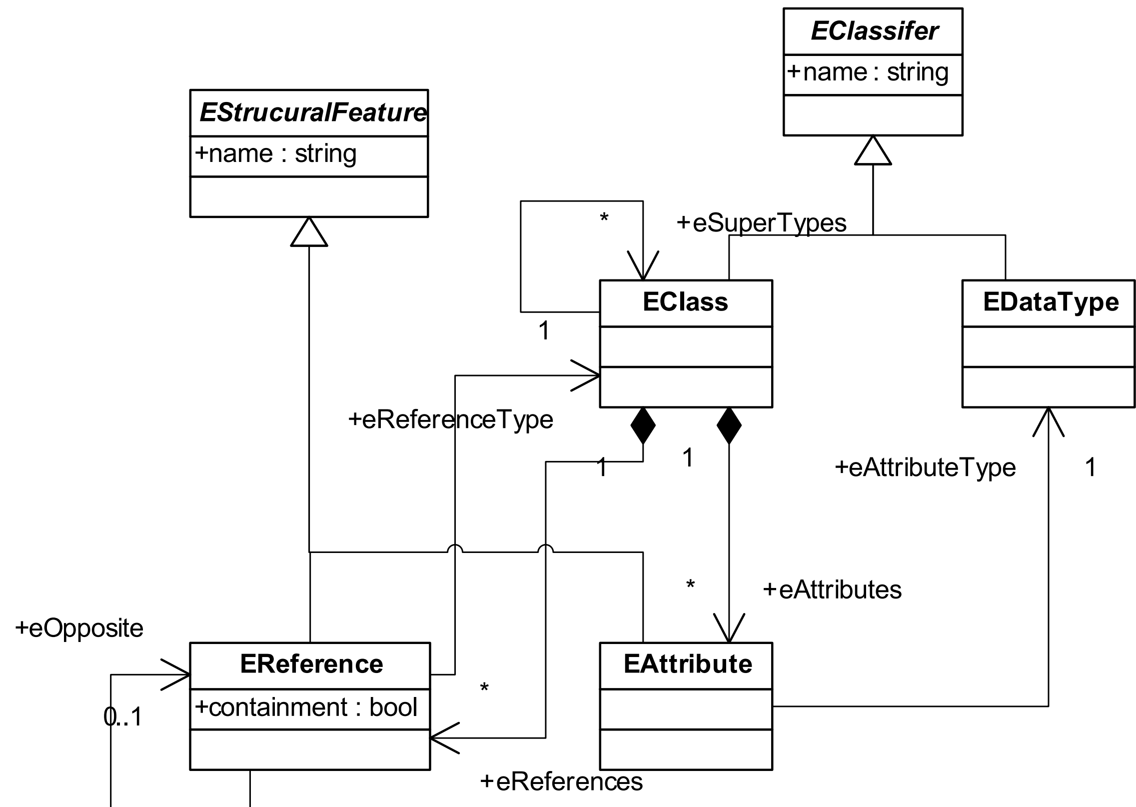
```
ResourceSet myResourceSet = new ResourceSetImpl();
Resource myResource =
        myResourceSet.getResource(URI.createURI("library.xml"), true);
```

Model Edition & Navigation

- EMF hierarchy
  - Ecore metamodel
  - Ecore navigation
  - EObject interface
- Generic APIs
  - EMF reflective API
  - Dynamic EMF

Ecore Kernel

- Ecore is
  - The model used to represent EMF models
  - An EMF model
  - Its own metamodel
  - A metametamodel



Ecore Navigation (excerpt)

- EClass – a modeled class

```
EStructuralFeature getEStructuralFeature(String /*name*/);
EList getEOperations();
boolean isInterface();
boolean isSuperTypeOf(EClass);
```

- EStructuralFeature – a class reference/attribute

```
Object getDefaultValue();
EClass getEContainingClass();
```

- EReference – an association end

```
EReference getEOpposite ();
boolean     isContainment();
```

EObject Interface

- EObject
  - Base interface of all modeled objects
    - EMF equivalent of java.lang.Object

- EObject facilities (excerpt)
  - Model navigation

```
EClass    eClass();
EObject  eContainer();
```

  - Model edition

```
Object  eGet(EStructuralFeature);
void     set(EStructuralFeature, Object /*newValue*/);
void     eUnset(EStructuralFeature);
```

EMF Reflective API

- Generated programmatic facilities
  - Package and Factory classes generated for the model
  - Model-specific classes
- Package generic API
  - Generic access to models metadata

    ```
    EClass  getEClassifier(String /*name*/);
    ```
- Factory generic API
  - Generic allocation of modeled objects

    ```
    EObject  create(EClass);
    ```

EMF Reflective API: Example

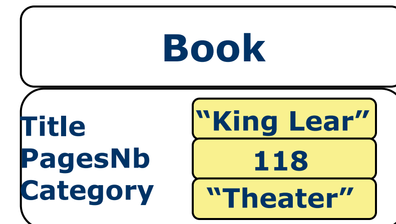
- Instantiating an object using the reflective API

```
EClass bookClass = libraryPackage.getEClassifier("Book");
Book myBook = libraryFactory.create(bookClass);

EAttribute bookTitle =
            bookClass.getEStructuralFeature("Title");
myBook.eSet(bookTitle, "KingLear");

EAttribute bookPagesNb =
            bookClass.getEStructuralFeature("PagesNb");
myBook.eSet(bookPagesNb, new Integer(118));

EAttribute bookCategory =
            bookClass.getEStructuralFeature("Category");
myBook.eSet(bookCategory, "Theater");
```



Dynamic EMF

- Generic programmatic facilities
  - Core Package/Factory classes
    - Handling different models in a generic way
    - Runtime model generation

- EcorePackage - access to Ecore's metadata

```
EClass      getEClass();
EAttribute getEAttribute();
EDataType getEString();
```

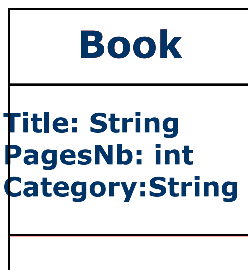
- EcoreFactory - instantiate Ecore model objects

```
EPackage  createEPackage();
EClass      createEClass();
EAttribute createEAttribute();
```

Dynamic EMF: Example (1/2)

- Runtime model generation

Title: String




```
EcoreFactory ecoreFactory  = EcoreFactory.eINSTANCE;
EcorePackage ecorePackage = EcorePackage.eINSTANCE;

EClass bookClass = ecoreFactory.createEClass();
bookClass.setName(“Book”);

EAttribute bookTitle = ecoreFactory.createEAttribute();
bookTitle.setName(“Title”);
bookTitle.setEType(ecorePackage.getEString());
bookClass.getEAttributes().add(bookTitle);

...

EPackage libraryPackage = ecoreFactory.createEPackage();
libraryPackage.setName(“library”);
libraryPackage.getEClassifiers().add(bookClass);
```

Dynamic EMF: Example (2/2)

- Using a dynamically generated model
  - Dynamic creation of instances

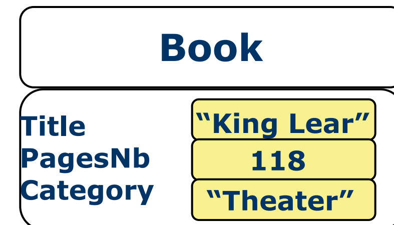
```
EFactory libraryFactory = libraryPackage.getEFactoryInstance();

EObject myBook = libraryFactory.create(bookClass);

myBook.eSet(bookTitle, "King Lear");

myBook.eSet(bookPagesNb, new Integer(118));

myBook.eSet(bookCategory, "Theater");
```



EMF Reflective API vs. Dynamic EMF

- Dynamic EMF
  - Type-unsafe
  - Model-independent
    - Enable runtime model generation
- Reflective API
  - Type-safe
  - Better performance
    - Use less memory
    - Provide faster access to data
  - Model-dependent
    - Require generated classes to be maintained as models evolve