

*whole.sourceforge.net*

# Whole Platform

a Model Driven, Generative technology for  
Developing Software

RICCARDO SOLMI

December 2005

# Summary

---

- **The Whole Platform**
  - What is, and live demo
- **Delegating the Design responsibility**
  - Hand written Design Patterns
  - Generative, Model Driven approaches
- **Whole Languages**
  - Abstracting from a fixed notation and semantics
  - Software Systems as languages
- **Work in Progress**
  - Modeling behavior
- **Related works and conclusions**

# What is the Whole Platform

- *A model driven technology* for engineering the production of software
- *A development environment* for developing new languages and tools
- *A metaprogramming tool* for writing model to model generators

# Whole Platform

---

- **A software platform including:**
  - **Whole IDE:** a *visual programming environment* supporting rich graphical editing of model driven languages.
  - **Whole Languages:** a *family of languages* including popular languages and some new metamodeling languages
  - **Whole Generative System:** a *set of frameworks* providing the runtime and the infrastructure for implementing model driven languages and for giving them a translational semantics

# Hand written Design Patterns

---

## ■ Definition

- A Pattern describes a recurrent design problem and the core of a reusable solution.

## ■ Use

- To *dominate the complexity* of a Software System
- To design and implement Software with the desired flexibility and the desired *ability to evolve* (lifecycle perspective)

## ■ Catalog

- Each Design Pattern has a *name*, an *applicability*, one schema of *solution* and a description of *consequences* of applying the pattern.

# Example of pattern: Abstract Factory

---

## ■ Intent

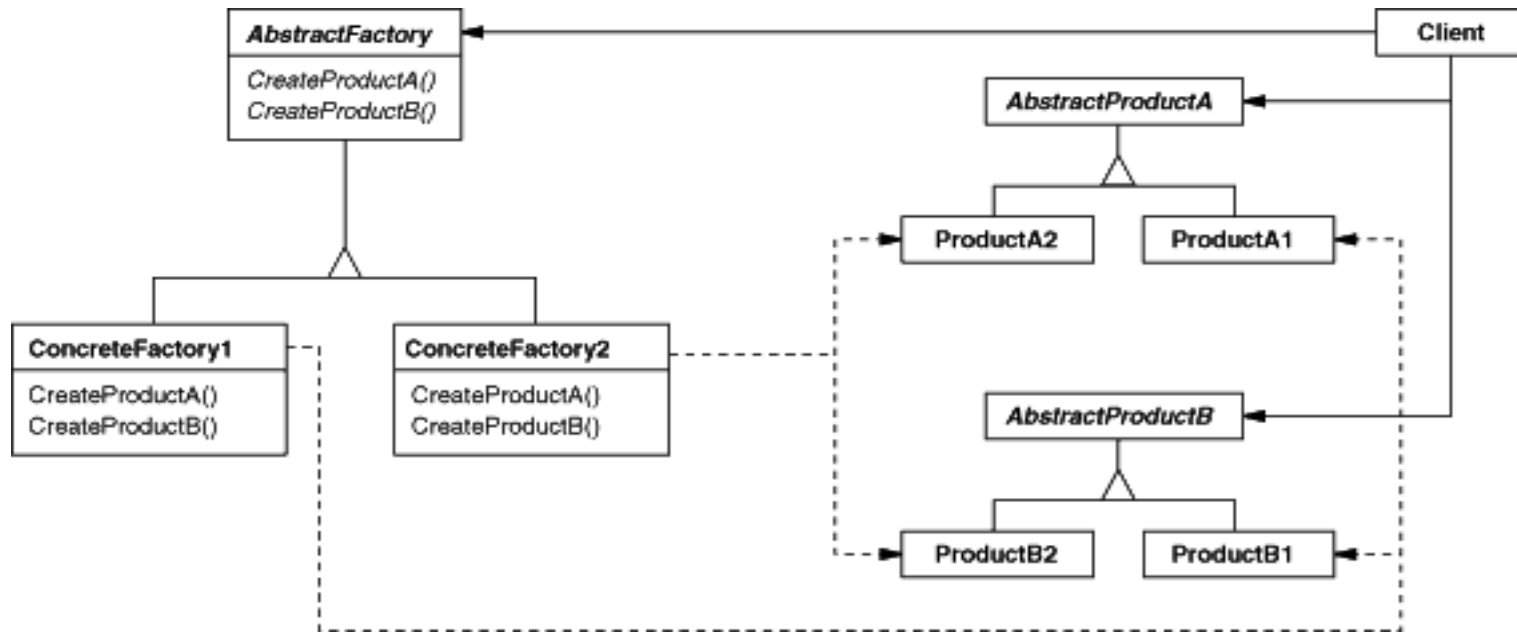
- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

## ■ Applicability

- A system should be independent of how its products are created, composed, and represented.
- A system should be configured with one of multiple families of products.
- A family of related product objects is designed to be used together, and you need to enforce this constraint.
- You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

# Abstract Factory /2

## Structure



## Participants

- **AbstractFactory**
  - declares an interface for operations that create abstract product objects.
- **ConcreteFactory**
  - implements the operations to create concrete product objects.
- **AbstractProduct**
  - declares an interface for a type of product object.
- **ConcreteProduct**
  - defines a product object to be created by the corresponding concrete factory.
  - implements the AbstractProduct interface.
- **Client**
  - uses only interfaces declared by AbstractFactory and AbstractProduct classes

# Abstract Factory /3

---

## ■ Collaborations

- Normally a single instance of a ConcreteFactory class is created at run-time. This concrete factory creates product objects having a particular implementation.
- AbstractFactory defers creation of product objects to its ConcreteFactory subclass

## ■ Consequences

- It isolates concrete classes
- It makes exchanging product families easy
- It promotes consistency among products
- Supporting new kinds of products is difficult

# Abstract Factory /4: First Order Logic example

## ■ Java Implementation (an abstract factory)

```
public interface FirstOrderLogicLanguageFactory {
    public static final FirstOrderLogicLanguageFactory instance = new FirstOrderLogicImplLanguageFactor
    I
    public Theory createTheory(IName name, IAssertions assertions);

    public Assertions createAssertions(Assertion[] entities);

    public Axiom createAxiom(IName name, Formula statement);

    public Theorem createTheorem(IName name, Formula statement, IFormulae proof);

    public Formulae createFormulae(Formula[] entities);

    public PredicateApplication createPredicateApplication(
        IPredicate predicate, IArguments arguments);

    public Implication createImplication(Formula antecedent, Formula consequent);

    public Coimplication createCoimplication(Formula leftFormula,
        Formula rightFormula);

    public Xor createXor(Formula leftFormula, Formula rightFormula);

    public And createAnd(Formula leftFormula, Formula rightFormula);

    public Or createOr(Formula leftFormula, Formula rightFormula);

    public Not createNot(Formula formula);
}
```

# Abstract Factory /5 : First Order Logic example

## ■ Java Implementation (a concrete factory)

```
public class FirstOrderLogicImplLanguageFactory implements
    FirstOrderLogicLanguageFactory {
    public Theory createTheory(IName name, IAssertions assertions) {
        return new TheoryImpl(name, assertions);
    }

    public Assertions createAssertions(Assertion[] entities) {
        return new AssertionsImpl(entities);
    }

    public Axiom createAxiom(IName name, Formula statement) {
        return new AxiomImpl(name, statement);
    }

    public Theorem createTheorem(IName name, Formula statement, IFormulae proof) {
        return new TheoremImpl(name, statement, proof);
    }

    public Formulae createFormulae(Formula[] entities) {
        return new FormulaeImpl(entities);
    }

    public PredicateApplication createPredicateApplication(
        IPredicate predicate, IArguments arguments) {
        return new PredicateApplicationImpl(predicate, arguments);
    }
}
```

# Design Patterns impact on software development

---

## ■ **Pattern abstraction level:**

- Code Patterns, *Design Patterns*, Architectural Patterns
- Analysis Patterns, Organization Patterns, Domain-Specific Patterns

## ■ **Design Pattern classification**

- **Creational** – To abstract the instantiation process
  - Abstract Factory, Builder, Factory Method, Prototype, Singleton
- **Structural** – How to compose classes and objects in larger structures
  - Adapter, Composite, Decorator, Proxy
- **Behavioral** – To deal with algorithms and the assignment of responsibilities between objects
  - Command, Iterator, Observer, Template Method, Visitor

# Design Patterns: application responsibility

---

- **A Pattern *solution* is not a concrete design or implementation.**
- ***Software architects* are responsible to choose and compose the right patterns to build a given Software System.**
- **From ad hoc to reusable design application**
  - *Compound Patterns* are also recurrent
  - The idea is to design a set of Patterns that covers the common requirements of each Software System
  - Several difficulties arise
  - Model Driven, Generative approaches try to overcome (or live with) them

# Difficulties in the use of Design Patterns

---

## ■ Trade-offs

- A combination of Patterns with maximum flexibility and maximum ability to evolve does not exist.
- A Design Pattern makes easy some future development to the detriment of other ones (Example: embedded vs modular behavior).

## ■ Over-engineering

- The code becomes more flexible and sophisticated than it is necessary.
- Early choice of Patterns.

## ■ Computational cost

- The flexibility of a solution costs and involves a higher complexity.
- To use a Design Pattern for *future* choices is a *cost*

# Delegating (part of) the Design Responsibility from Developers to the Whole Platform

---

- **Automatic pattern application requires both Generative and Model Driven technologies**
- **Generative technologies**
  - For generating implementation code starting from reusable template descriptions of Patterns and problem specification.
- **Model Driven technologies**
  - A Pattern is not a modular unit of code free of dependencies (it is an aspectual view of the System).
  - System level knowledge is required for generating implementation of patterns.
  - *Models* capture the problem specification

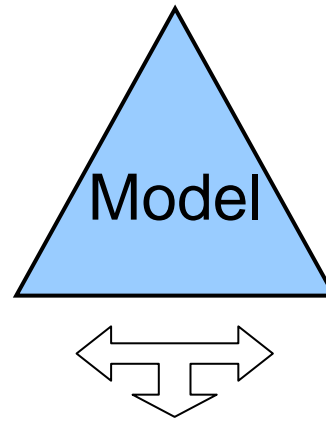
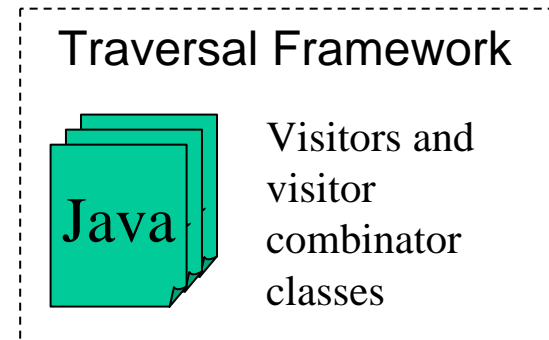
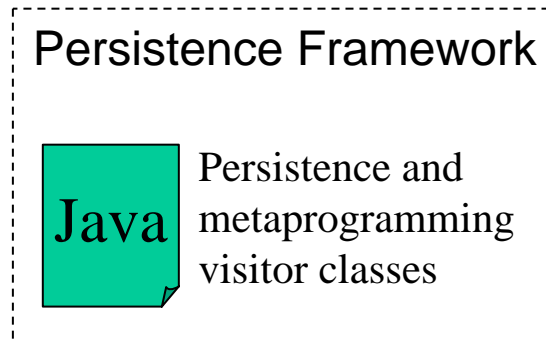
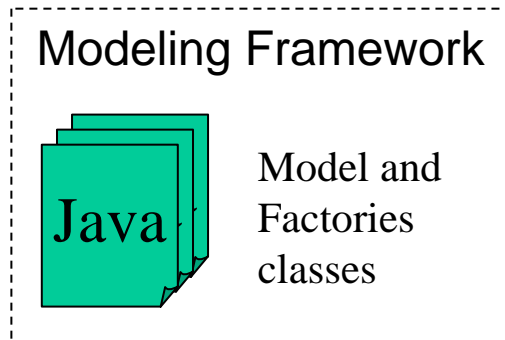
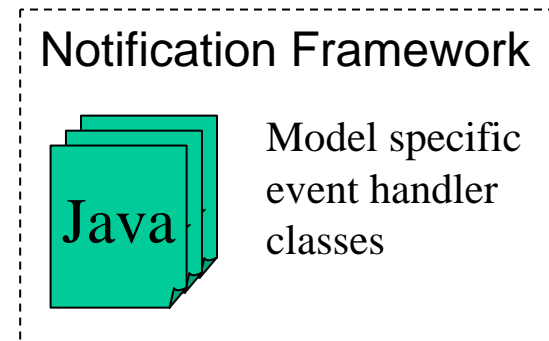
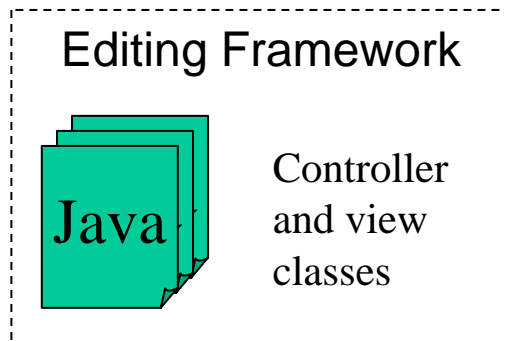
# Modeling with the Whole: the Models language

- ***A Domain is a knowledge area***
  - Concepts, terminology, activities
- ***Software Systems are built over a (part of) domain***
- ***A Model is an abstraction over a (part of) domain***
- ***A Model collects information about a domain defining entities, features and structural constraints (types)***
- **Model example: First Order Logic**

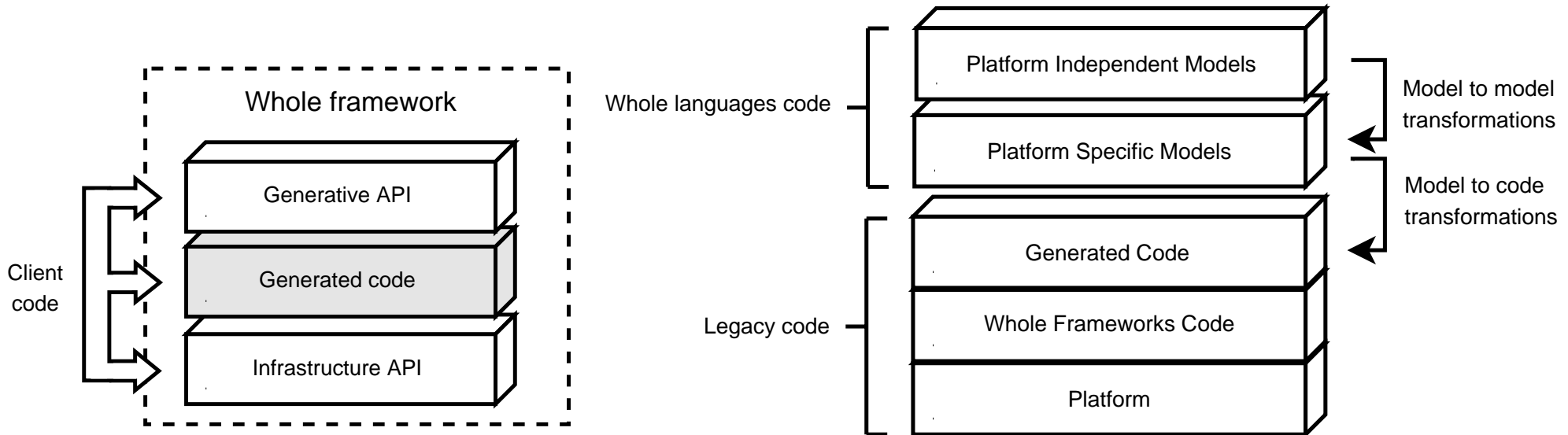
```
model FirstOrderLogic {  
  template types ·  
  
  · entity Theory types ·  
    · feature Name name  
    · feature Assertions assertions  
  
  · entity Assertions types ·  
    ordered composite < Assertion >  
  
  · entity Axiom types Assertion  
    · feature Name name  
    · feature Formula statement  
  
  · entity Theorem types Assertion  
    · feature Name name  
    · feature Formula statement  
    · feature Formulae proof  
  
  · entity Formulae types ·  
    ordered composite < Formula >  
  
  · entity PredicateApplication types Formula  
    · feature Predicate predicate  
    · feature Arguments arguments
```

# Code Generation

- Code generated from the First Order Logic model



# Whole Generative System



# Whole Languages

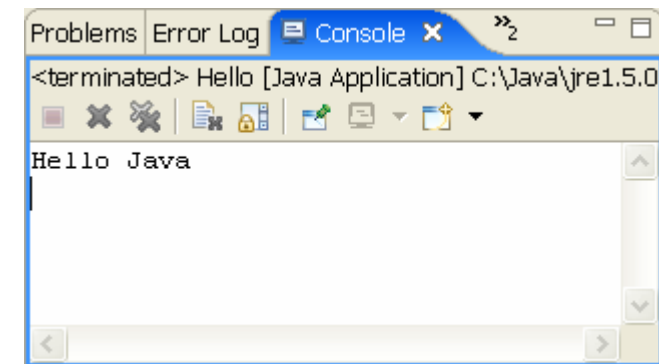
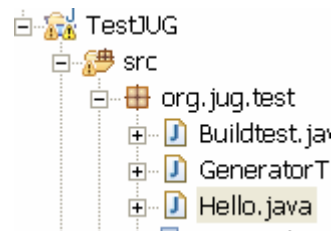
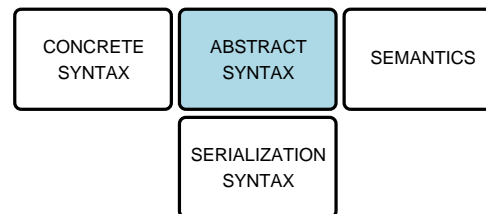
---

- **Languages can include:**
  - General Purpose Languages (GPL)
    - examples: Java, Scheme, XML
  - Domain Specific Languages (DSL)
    - examples: BNF, SQL
  - Set of constructs; examples: declarations, control flow, math, aspects, metaprogramming
  - Composed languages; example: Java + metaprogramming
- **A standard set of languages is provided**
  - Metamodeling DSL: Models, Prototypes, Visitors
  - Popular languages: Java, XML, Text, BNF, SQL
- **New languages can be added and composed**

# Languages

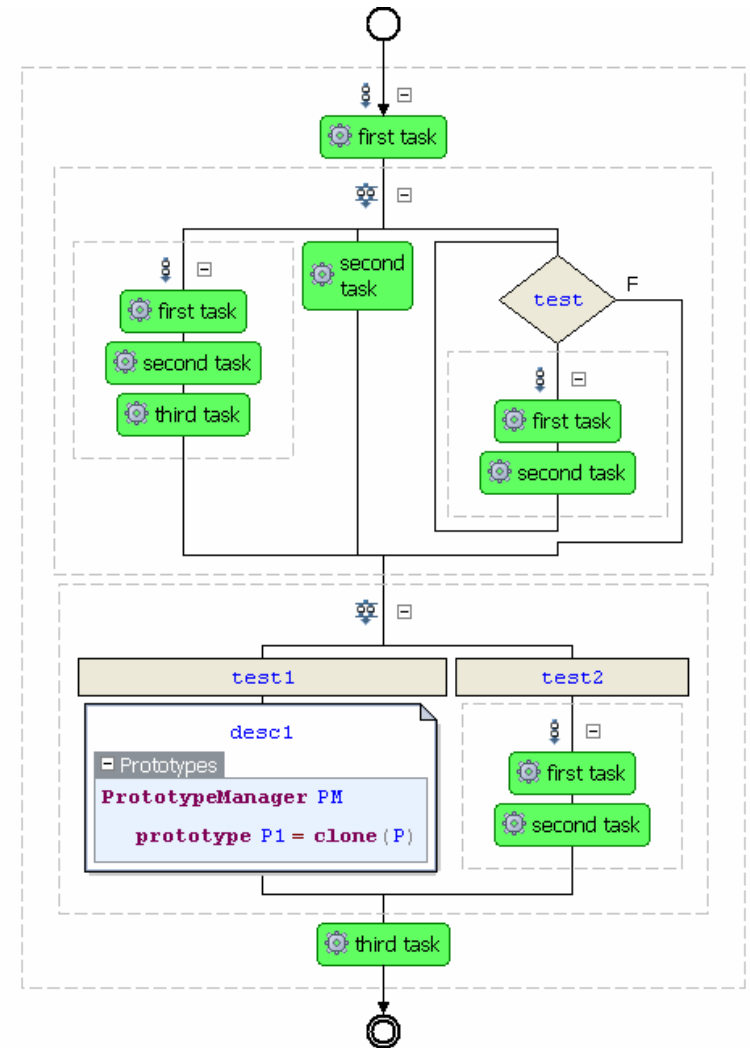
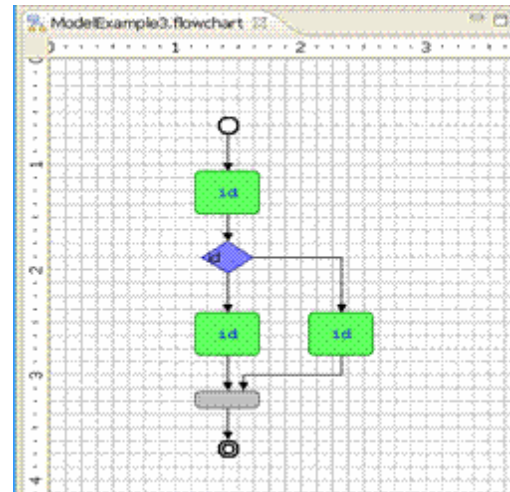
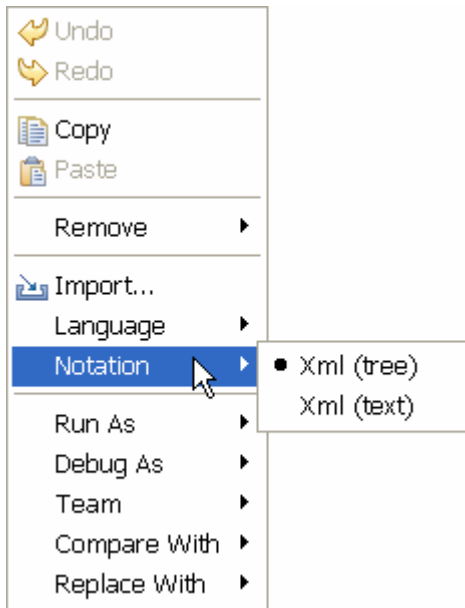
- **Giving a central role to modeling we change languages**
  - From a fixed notation (concrete syntax), serialization and semantics
  - To a fixed model (abstract syntax) with multiple notations, semantics and serializations
- **and language tools in an automation perspective**
  - Avoid parsing and unparsing processes
  - Standard representation of the problem: the models

```
package org.jug.test;  
  
public class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hello Java");  
    }  
}
```



# Notations

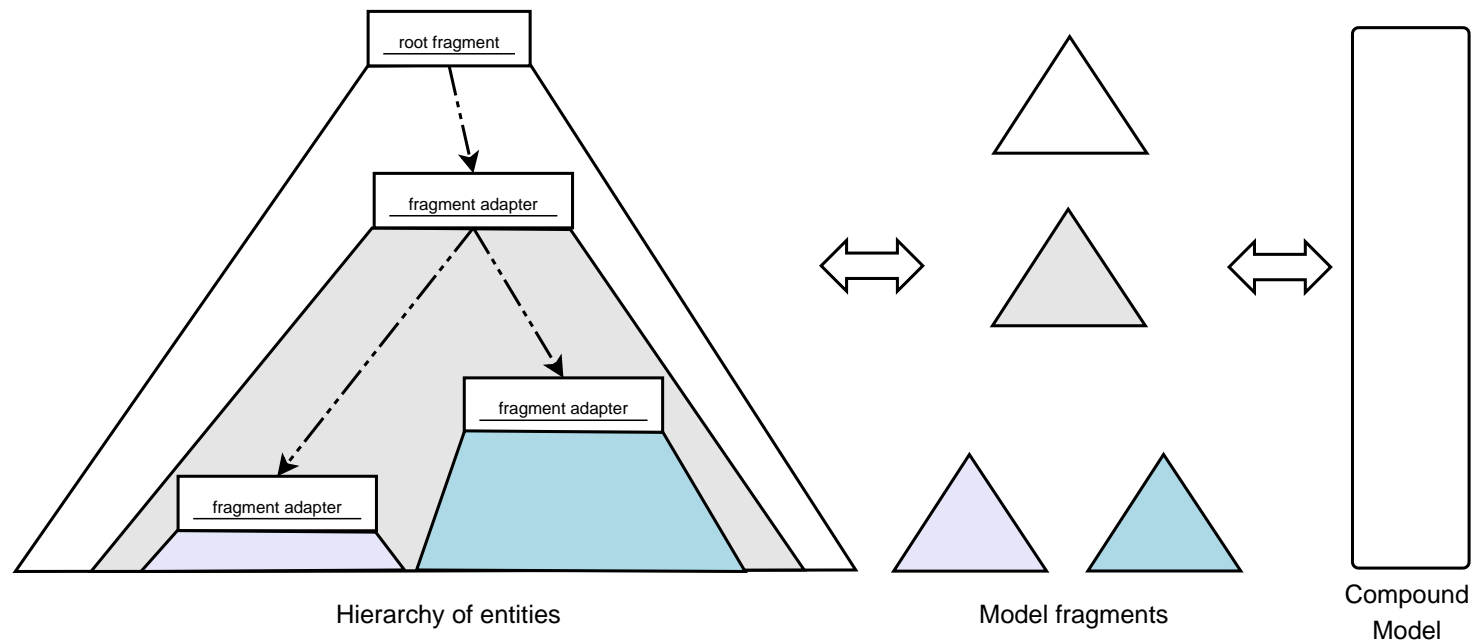
- **The editing framework supports multiple kinds of notations**
  - Both textual or graphical
  - With manual or automatic layouts
- **Multiple notations per language**



# From Monolithic languages to Composable languages

## ■ Language composition support

- Compound models of different languages
- Behavior assimilation and Metaprogramming
- Aspectual composition



# Examples of Language composition

## ■ Modeling fragments

- Models
- Prototypes
- Visitors

## ■ Domain fragments

- Collaboration
- Automotion
- Project management

## ■ Aspectual fragments

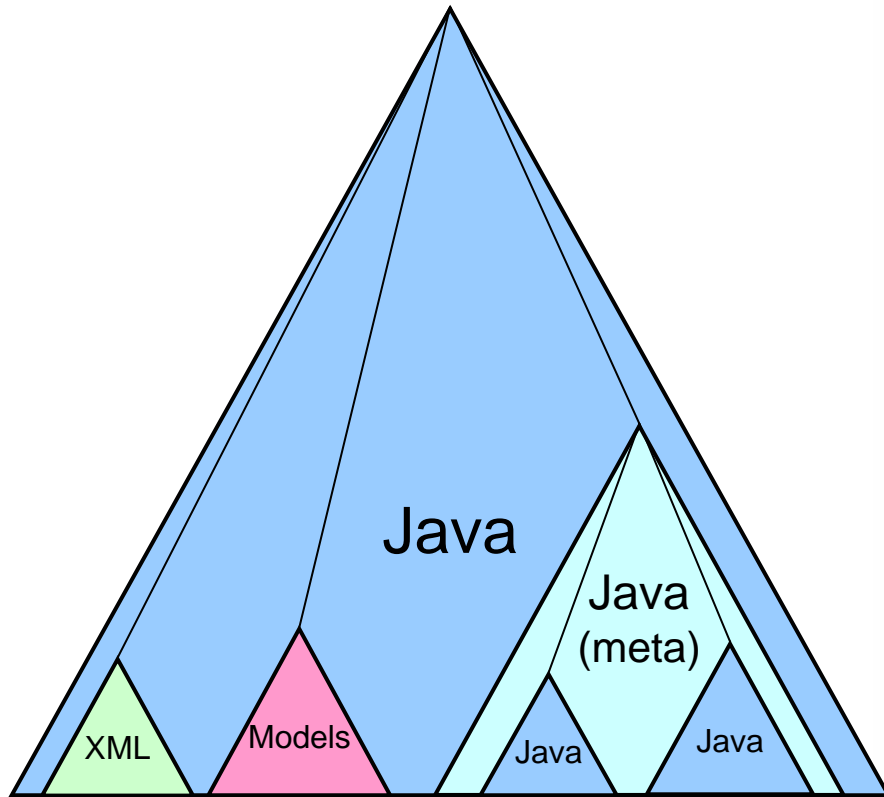
- Documentation
- Presentation
- Validation

```
Prototypes
PrototypeManager Models
  prototype SimpleName = SimpleName( _id_ )
  prototype SimpleEntity = SimpleEntity( clone( SimpleName ) )

  prototype BinaryExp =
    Models
      · entity _id_types Exp
      · feature Exp left
      · feature Exp right
```

```
Models
model Models {
  template types ·
  · entity SimpleName types Type
    value < String >
  · entity SimpleEntity types ModelDeclaration
    · feature EntityModifiers modifiers
    id feature SimpleName name
    · feature Types types
    · feature Features features
}
```

# Example of Metaprogramming



```
public class EditorKitBuilder extends CompilationUnitBuilder {  
    public EditorKitBuilder ( LanguageGenerator generator ) {
```

```
IEntity xmlMod =  
    < tag attr1=" attrValue " >  
        <![CDATA[ ]]>  
    </ tag >  
IEntity modelsModel =  
    . entity_id_types .  
    . feature _id_
```

```
package org.whole.lang.visitors.ui;  
  
public class editorKit extends AbstractEditorKit {  
    public static final String ID = " editorId " ;  
    public String getId ( ) {  
        return ID ;  
    }  
    public String getName ( ) {  
        return " Visitors " ;  
    }  
    public EditPartFactory getPartFactory ( ) {  
        return partFactoryVisitor .instance ( ) ;  
    }  
}
```

# Work in Progress

## ■ Modeling behavior with dedicated DSL languages

- Now is Model Driven but written in Java by hand.
- Based on Traversal and Notification frameworks.

```
public class EditorKitBuilder extends CompilationUnitBuilder {
    public EditorKitBuilder(LanguageGenerator generator) {
        super(generator);

        addClassDeclaration(generator.editorKitName(),
            AbstractEditorKit.class.getName());

        FieldDeclaration fieldDec = newFieldDeclaration(
            "String", "ID", newLiteral(generator.editorKitName()));
        fieldDec.setModifiers(
            Modifier.PUBLIC | Modifier.STATIC | Modifier.FINAL);
        addBodyDeclaration(fieldDec);

        methodDec = newMethodDeclaration("String", "getId");
        addStatement(methodDec,
            newReturnStatement(ast.newSimpleName("ID")));
        addBodyDeclaration(methodDec);

        methodDec = newMethodDeclaration("String", "getName");
        addStatement(methodDec,
            newReturnStatement(newLiteral(generator.EditorName)));
        addBodyDeclaration(methodDec);
    }
}
```

```
visitor Models2PrototypeManagerVisitor {
    when Model (e) {
        pm = Models.PrototypeManager ( name = e )
        traverse modelDeclarations
    }
    when Theory ( name = e assertions = Axiom ( name = A1 statement = e ) ) e {
        relation Entity2Prototype ( Models entity_id_types Prototypes prototype ) {
        }
        relation Feature2Param ( Models feature_id f Prototypes p ) {
        }
    }
}
```

# Related Works

---

- **OMG specifications: *Model Driven Architecture (MDA)***
  - Modeling (MOF), notation (UML), persistence (XMI)
  - TODO: Query, View and Transformations (QVT)
- **Eclipse proposals: *Model Driven Development, Language Development Toolkit and Graphical Modeling Framework***
  - Modeling (EMF), notation (GEF based), persistence (XMI)
  - Metamodels textual notation (Emfatic), transformations (MTF)
- **Vendor technology previews**
  - Software Factories (Microsoft)
  - Meta-Programming System (JetBrains)
  - Intentional Programming (Intentional Software)
  - HyperSenses (Delta Software)

# Conclusions

---

- **Advantages of the Whole Platform**
  - Frameworks are designed using state of the art Patterns
  - Eclipse based Development Environment
  - Model Driven metaprogramming with concrete syntax
  - Integrated language and tools for modeling
- **Used for building real languages and real business applications**
- **The Whole Platform is an *open source* project hosted by *whole.sourceforge.net***



# Traversal API

- Supports definition of polymorphic operations
  - Based on a pattern derived from Visitor
- Supports model specific and generic operations
  - Ex.: generate, compile, execute, store, ...
- Composition of visitors is supported
  - A set of visitor combinators is provided
- Staging of visitors is supported

# Notification API

- Supports definition of intra and inter model constraints
- An *event* is a function call that happens whenever the model is changed
- An *event handler* is a module that defines the behavior associated to given model changes
- Event handlers can be combined
  - A set of event handler combinators is provided
  - New event handlers can be defined and added at runtime

# Persistence and Metaprogramming API

- Is a persistence layer for storing models
- Is a meta representation for generating models
- Based on a stream of building events
  - The default stream serializer generates Java code
  - Looks like the Director part of the Builder pattern
- Supports *multiple languages* at once
- Allows *automatic versioning* through refactoring
- A level 2 generator emits the persistence (generator) layer for a given model