THALES

EGF - Thales Corporate Services - EPM

EGF Tutorial

Benoît Langlois – Thales/EPM

Agenda

- Introduction
- EGF Structure
- Pattern

What is EGF?

- EGF (Eclipse Generation Factories) is an Eclipse open source project under the **EMFT project**.

- **Purpose**: provide a **model-based generation framework.**
- **Operational objectives**:
  - Supporting complex, large-scale and customizable generations
  - Promoting the constitution of generation portfolios in order to capitalize on generation solutions
  - Providing an extensible generation structure

Objective of this tutorial

Understanding:

- **The EGF Structure, with:**
  - Activity, Factory component, Task, Production plan
- **EGF Patterns**

Agenda

- Introduction
- EGF Structure
- Pattern

EGF Structure

Example

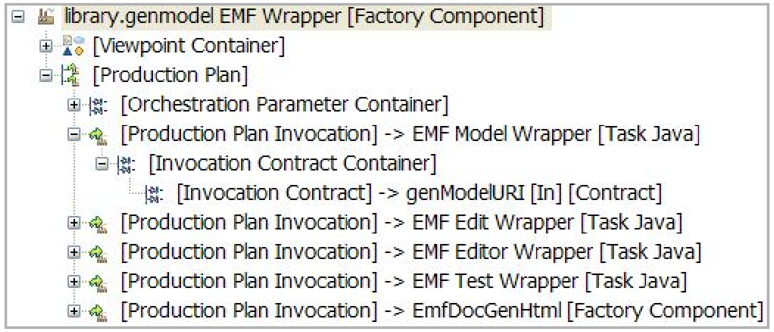
This following slides present snapshots of the EMF Wrapper provided by EGF, which can be activated by a right-click on a genmodel.

There is one generation step for model, edit, editor, test, and documentation generation.

Example – EMF Wrapper

Library.fcore file

contains



```
library.genmodel EMF Wrapper [Factory Component]
  [Viewpoint Container]
  [Production Plan]
    [Orchestration Parameter Container]
    [Production Plan Invocation] -> EMF Model Wrapper [Task Java]
      [Invocation Contract Container]
        [Invocation Contract] -> genModelURI [In] [Contract]
    [Production Plan Invocation] -> EMF Edit Wrapper [Task Java]
    [Production Plan Invocation] -> EMF Editor Wrapper [Task Java]
    [Production Plan Invocation] -> EMF Test Wrapper [Task Java]
    [Production Plan Invocation] -> EmfDocGenHtml [Factory Component]
```

Main Factory Component

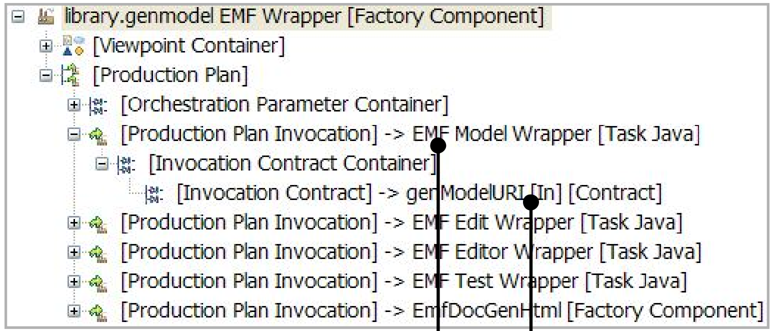
Task and Factory component invocation orchestration

Example – EMF Wrapper

Library.fcore file

contains

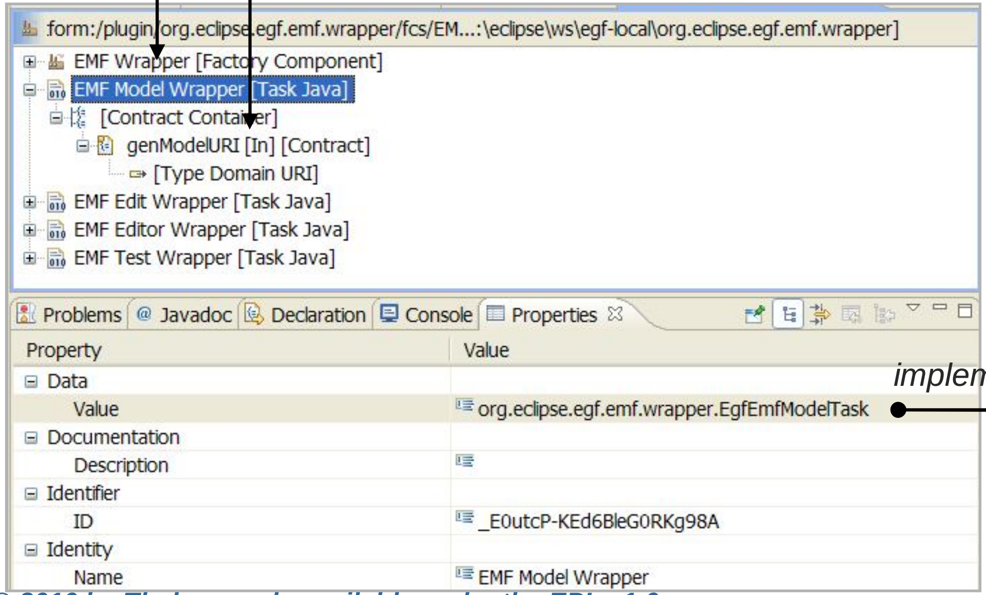
Java Task usage



```
⊟ library.genmodel EMF Wrapper [Factory Component]
  ⊞ [Viewpoint Container]
  ⊟ [Production Plan]
    ⊞ [Orchestration Parameter Container]
    ⊟ [Production Plan Invocation] -> EMF Model Wrapper [Task Java]
      ⊟ [Invocation Contract Container]
          [Invocation Contract] -> genModelURI [In] [Contract]
    ⊞ [Production Plan Invocation] -> EMF Edit Wrapper [Task Java]
    ⊞ [Production Plan Invocation] -> EMF Editor Wrapper [Task Java]
    ⊞ [Production Plan Invocation] -> EMF Test Wrapper [Task Java]
    ⊞ [Production Plan Invocation] -> EmfDocGenHtml [Factory Component]
```

Task reference

Contract value for a contract (= task parameter)

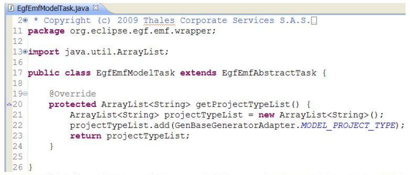


```
form:/plugin/org.eclipse.egf.emf.wrapper/fcs/EM...:\eclipse\ws\egf-local\org.eclipse.egf.emf.wrapper
⊞ EMF Wrapper [Factory Component]
⊟ EMF Model Wrapper [Task Java]
  ⊟ [Contract Container]
    ⊟ genModelURI [In] [Contract]
        [Type Domain URI]
⊞ EMF Edit Wrapper [Task Java]
⊞ EMF Editor Wrapper [Task Java]
⊞ EMF Test Wrapper [Task Java]
```

Property	Value
⊟ Data	
Value	org.eclipse.egf.emf.wrapper.EgfEmfModelTask
⊟ Documentation	
Description	
⊟ Identifier	
ID	_E0utcP-KEd6BleG0RKg98A
⊟ Identity	
Name	EMF Model Wrapper

implementation

Task Java Class



EgfEmfModelTask.java

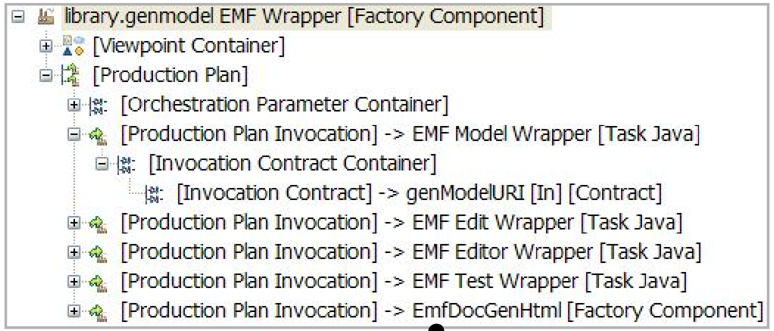
```
 * Copyright (c) 2009 Thales Corporate Services S.A.S.
package org.eclipse.egf.emf.wrapper;

import java.util.ArrayList;

public class EgfEmfModelTask extends EgfEmfAbstractTask {

    @Override
    protected ArrayList<String> getProjectTypeList() {
        ArrayList<String> projectTypeList = new ArrayList<String>();
        projectTypeList.add(GenBaseGeneratorAdapter.MODEL_PROJECT_TYPE);
        return projectTypeList;
    }

}
```

Example – EMF Wrapper

Library.fcore file

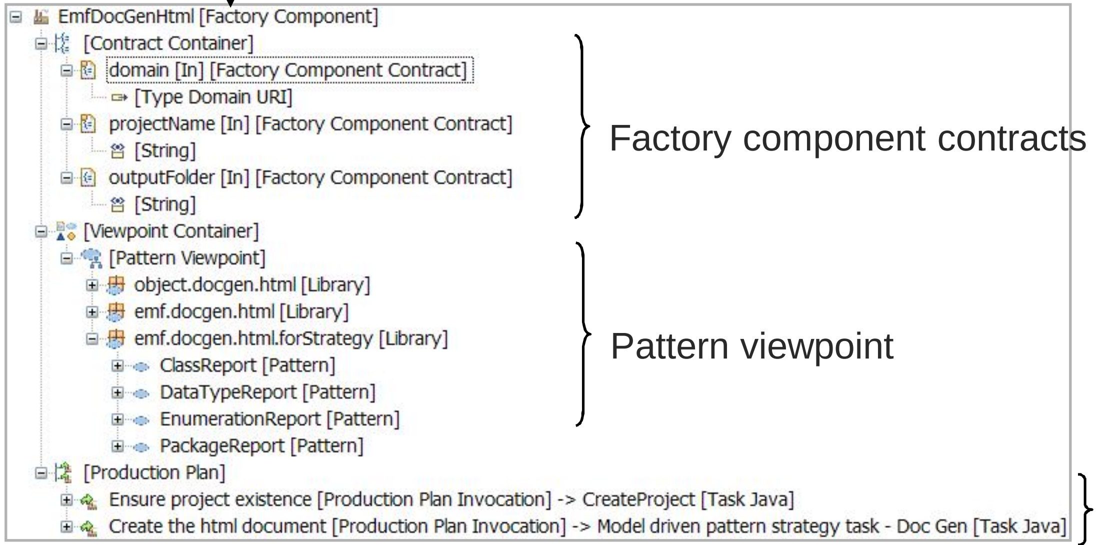
contains

Factory Component usage



- library.genmodel EMF Wrapper [Factory Component]
  - [Viewpoint Container]
  - [Production Plan]
    - [Orchestration Parameter Container]
    - [Production Plan Invocation] -> EMF Model Wrapper [Task Java]
      - [Invocation Contract Container]
        - [Invocation Contract] -> genModelURI [In] [Contract]
    - [Production Plan Invocation] -> EMF Edit Wrapper [Task Java]
    - [Production Plan Invocation] -> EMF Editor Wrapper [Task Java]
    - [Production Plan Invocation] -> EMF Test Wrapper [Task Java]
    - [Production Plan Invocation] -> EmfDocGenHtml [Factory Component]

Factory component reference



- EmfDocGenHtml [Factory Component]
  - [Contract Container]
    - domain [In] [Factory Component Contract]
      - [Type Domain URI]
    - projectName [In] [Factory Component Contract]
      - [String]
    - outputFolder [In] [Factory Component Contract]
      - [String]
  - [Viewpoint Container]
    - [Pattern Viewpoint]
      - object.docgen.html [Library]
      - emf.docgen.html [Library]
      - emf.docgen.html.forStrategy [Library]
        - ClassReport [Pattern]
        - DataTypeReport [Pattern]
        - EnumerationReport [Pattern]
        - PackageReport [Pattern]
  - [Production Plan]
    - Ensure project existence [Production Plan Invocation] -> CreateProject [Task Java]
    - Create the html document [Production Plan Invocation] -> Model driven pattern strategy task - Doc Gen [Task Java]

Factory component contracts

Pattern viewpoint

Task invocation orchestration

EGF Structure

Activity

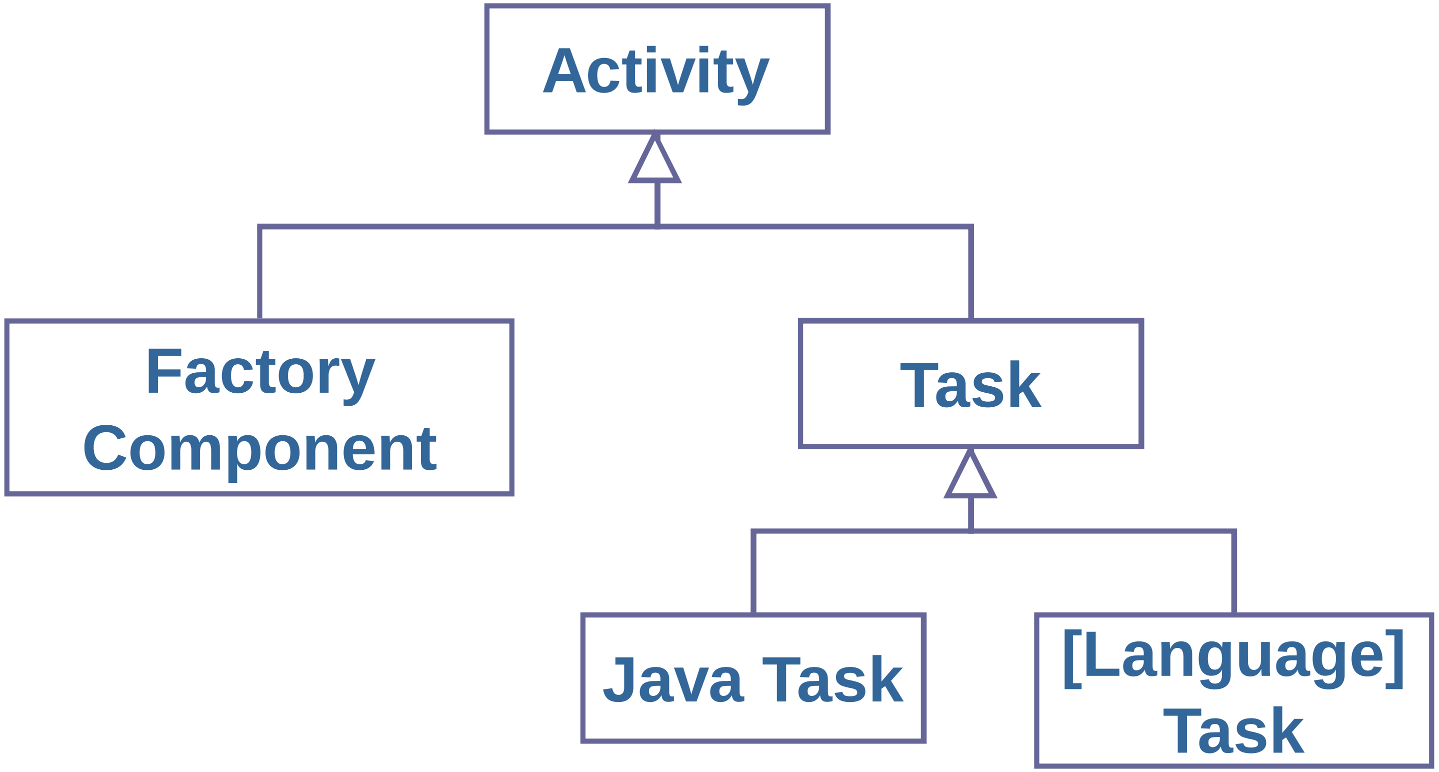
Activity – Definition

- **An activity is the abstract class of any EGF generation unit**
  - Factory component and Task are activities

- **Activity storage**
  - Activities are stored in **fcore files**
  - The same fcore file contains one to several activities

- **Activity properties**
  - Contract declaration
  - Ability to be invoked and to execute a generation action

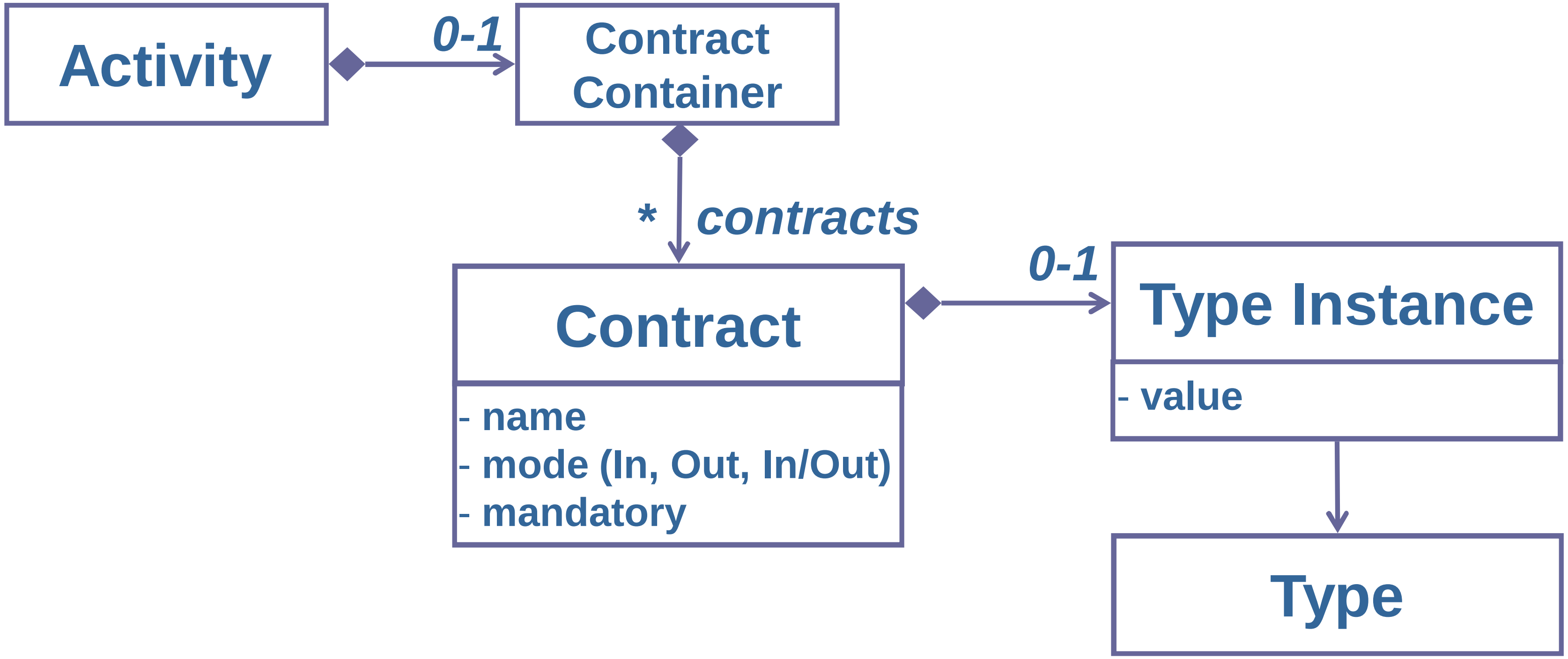
Activity Hierarchy



EGF: Eclipse Generation Factories – Thales Corporate Services/EPM

THALES

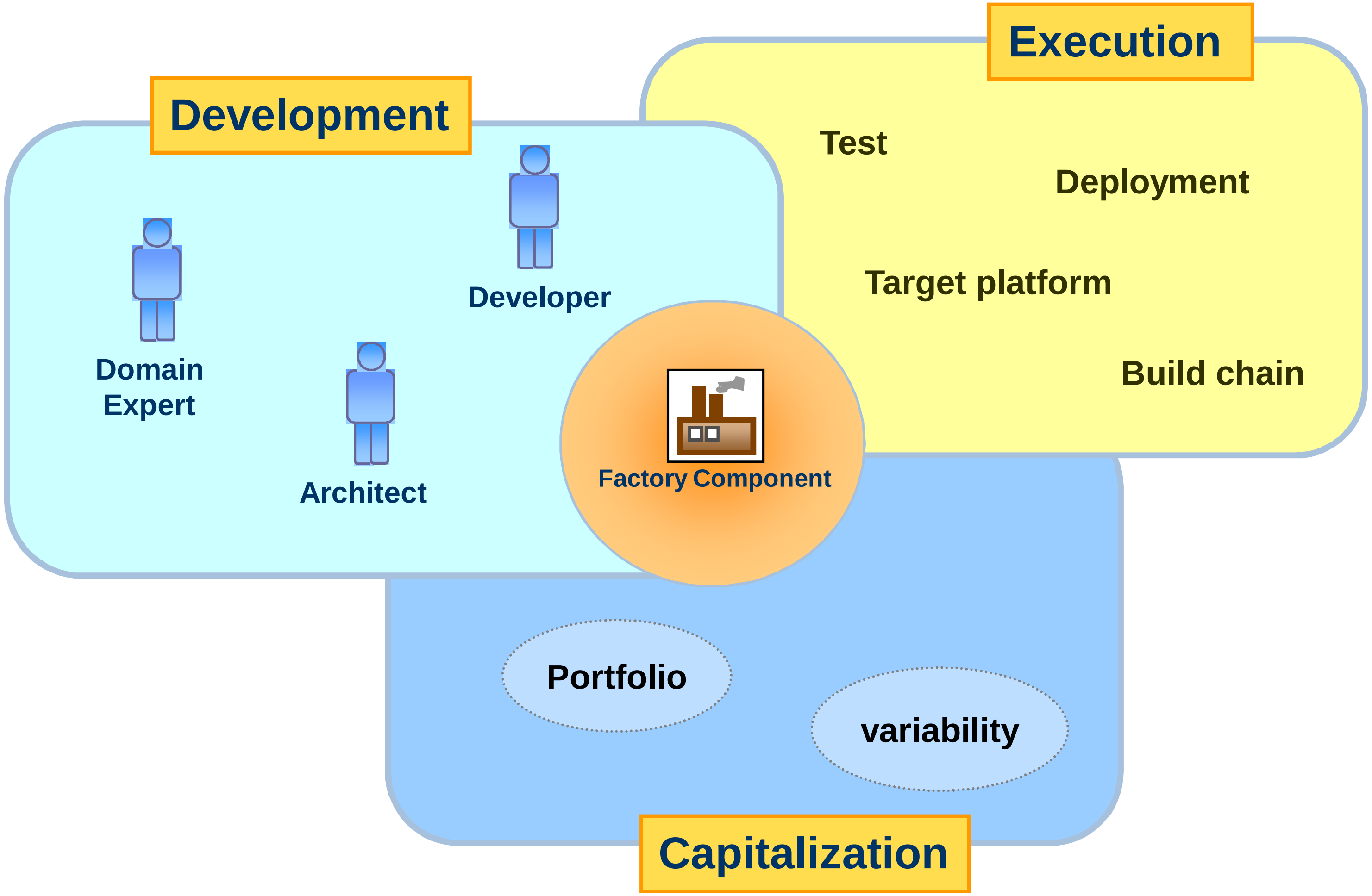
Activity Properties



EGF Structure

Factory Component

Central Role of Factory Component

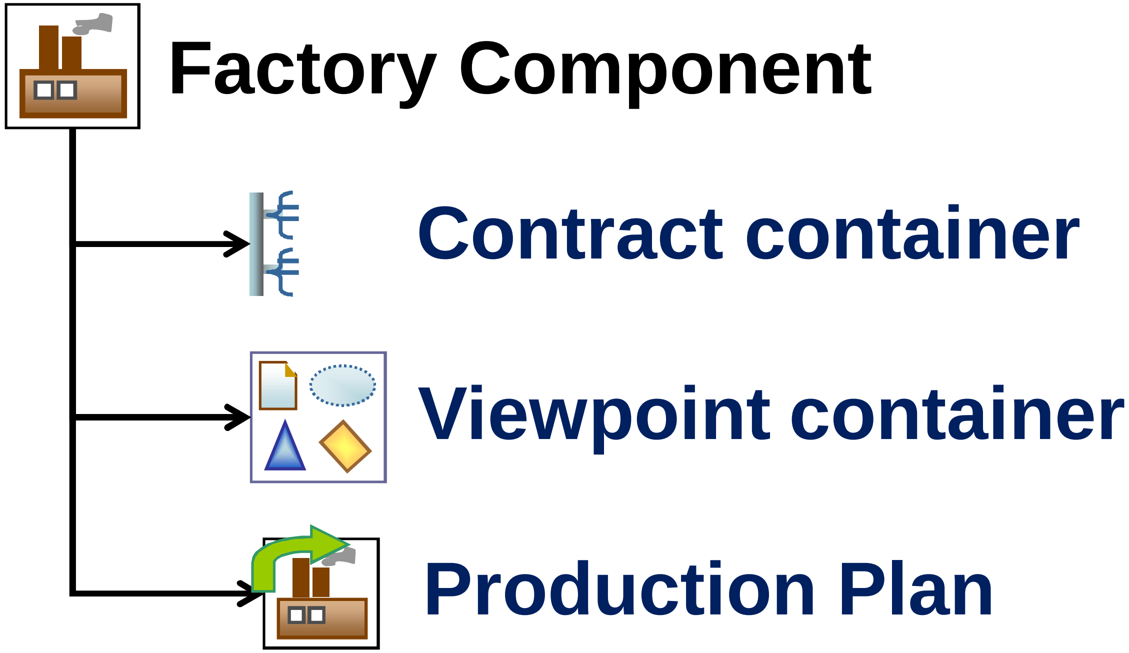


EGF: Eclipse Generation Factories – Thales Corporate Services/EPM

Factory Component

- Unit of generation with a clear **objective of generation**
- Unit of generation with a clear **contract**
- **Assembly** of factory components
  - Delegation
  - Creation of heterogeneous and complex generation chains
- Explicit declaration of generation data organised by **viewpoints**
- **Orchestration** of the generation with a production plan
- Factory Component **Lifecycle**: edition and execution, including validation

Factory Component Structure



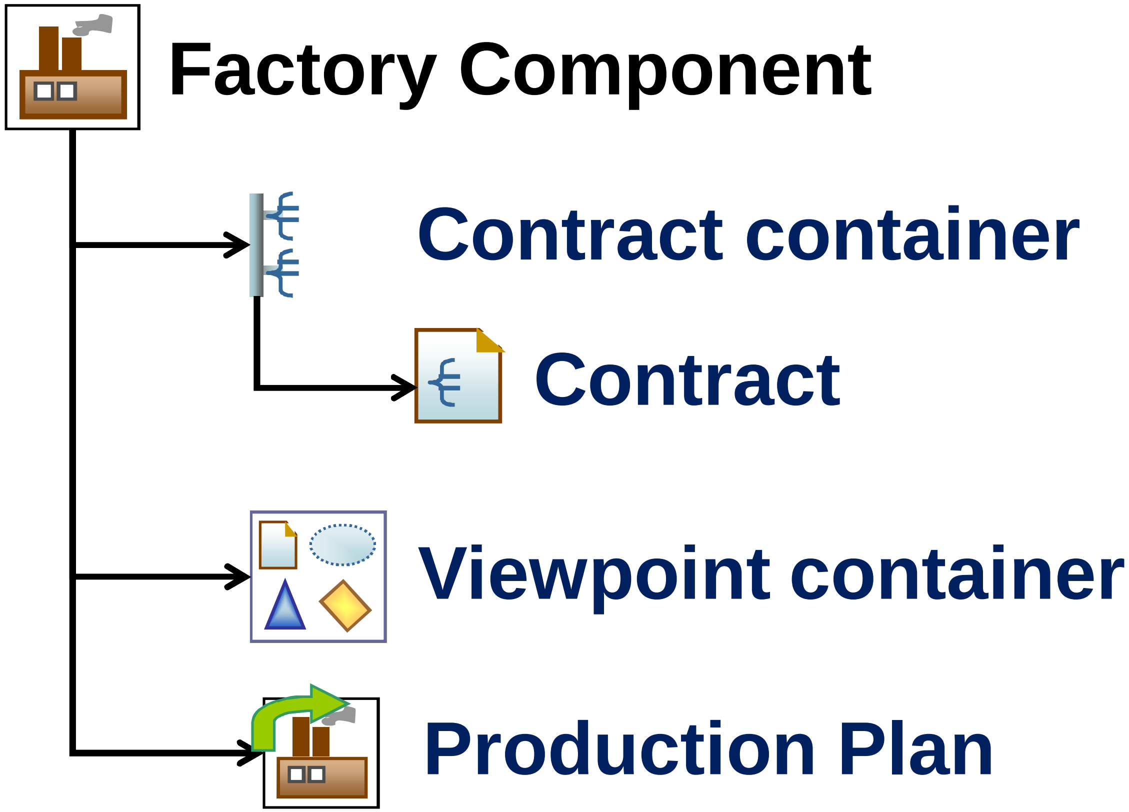
Factory Component

Contract container

Viewpoint container

Production Plan

Factory Component Structure



Factory Component

Contract container

Contract

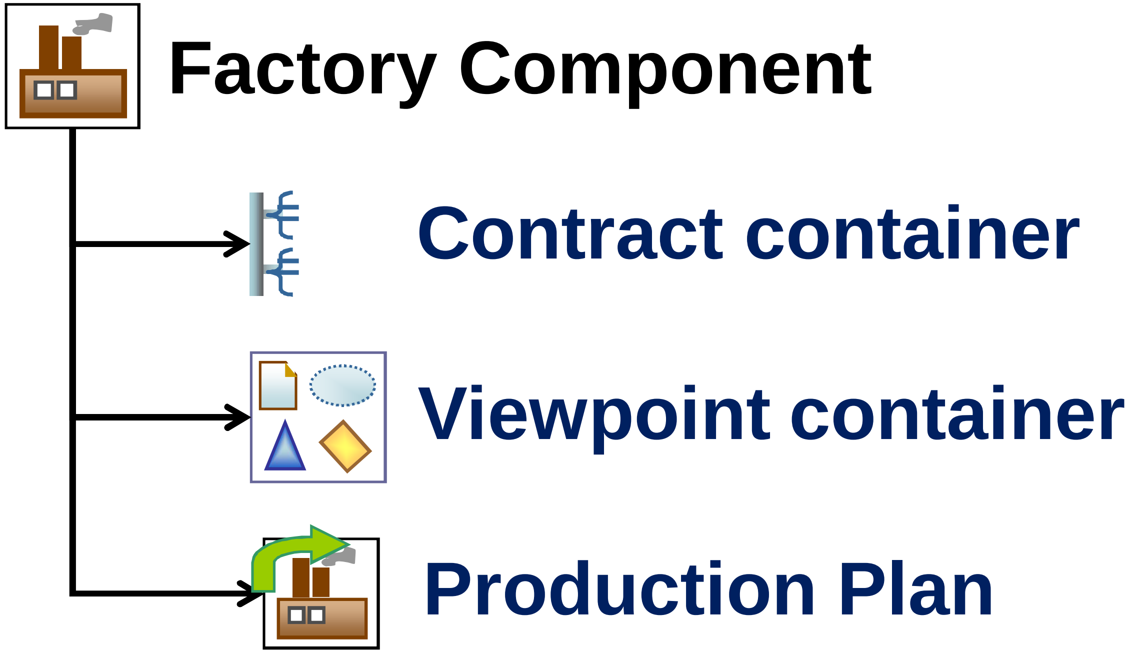
Viewpoint container

Production Plan

Definition:

- **Contract**: Factory Component parameter
- A contract has a type, a passing mode (In/Out/In_Out), a default value or not is mandatory or optional

Factory Component Structure



Factory Component

Contract container

Viewpoint container

Production Plan

Definition:

- **Viewpoint**: area to declare generation perspective data
- Examples of viewpoint:
  - Available today: domain declaration, pattern
  - Candidates: licensing, feature model

EGF Structure

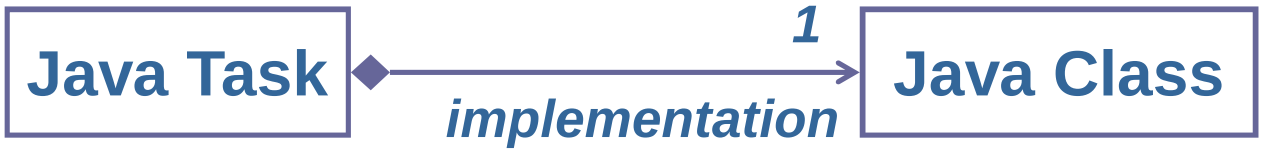
Task

Task – Definition

- **A task is an atomic generation unit**

- **Task declaration:**
  - A task is declared in a fcore file
  - Java task is a kind of Task. With the extensibility mechanism, other Task types could be introduced (e.g., Ruby task).

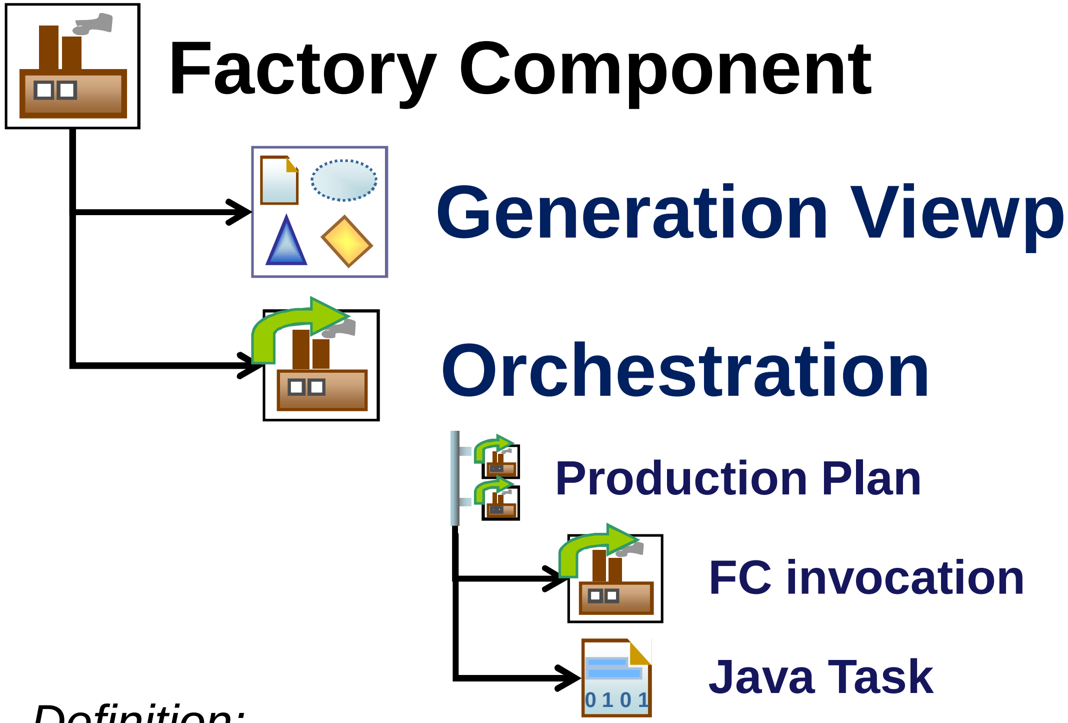
- **Task implementation:**
  - An implementation is associated to a task
  - A JavaTask is implemented by a Java class (which implements ITaskProduction)



EGF Structure

Production Plan

Factory Component Structure



Factory Component

Generation Viewpoint

Orchestration

Production Plan

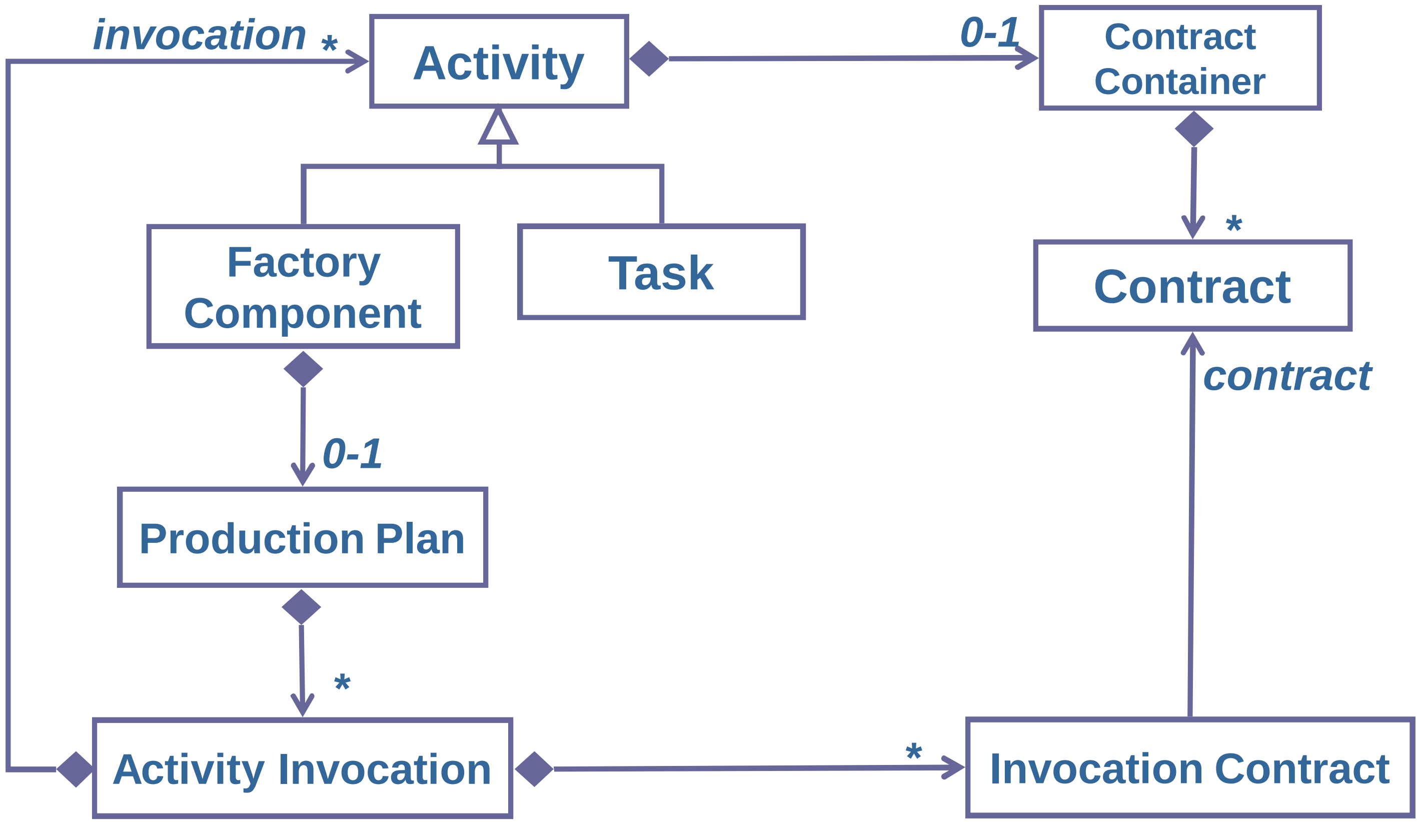
FC invocation

Java Task

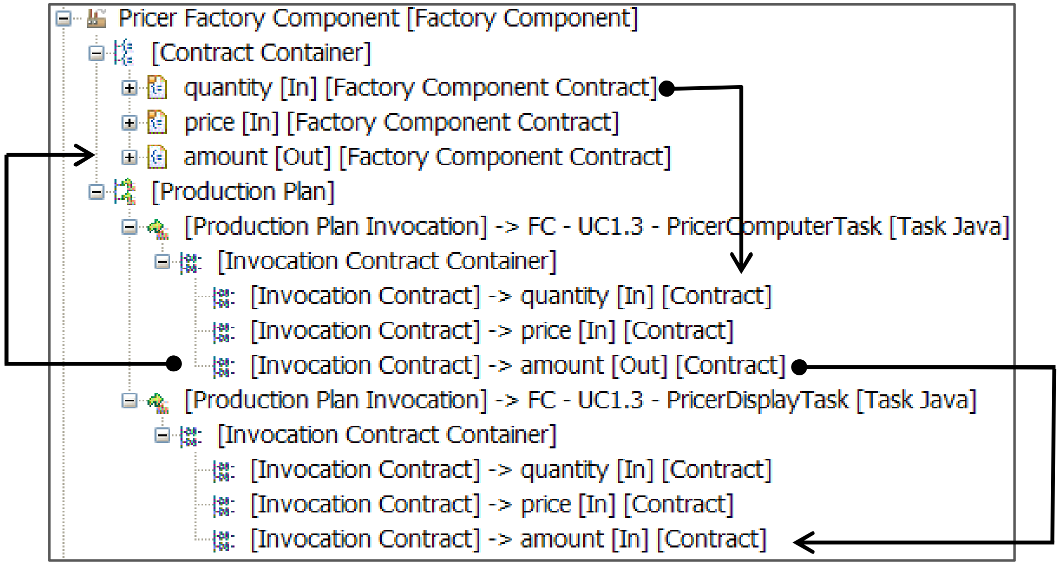
Definition:

- A production plan is a simple kind of generation orchestration
- **Production Plan**: A generation orchestration is a sorted list of factory component or generation task invocations. It describes the successive generation steps, which either call factory components or generation tasks.
- The factory component/task contracts are valued by factory component/task invocation values. Same principle than the parameter values when a Java method is called.

Activity and Production Plan

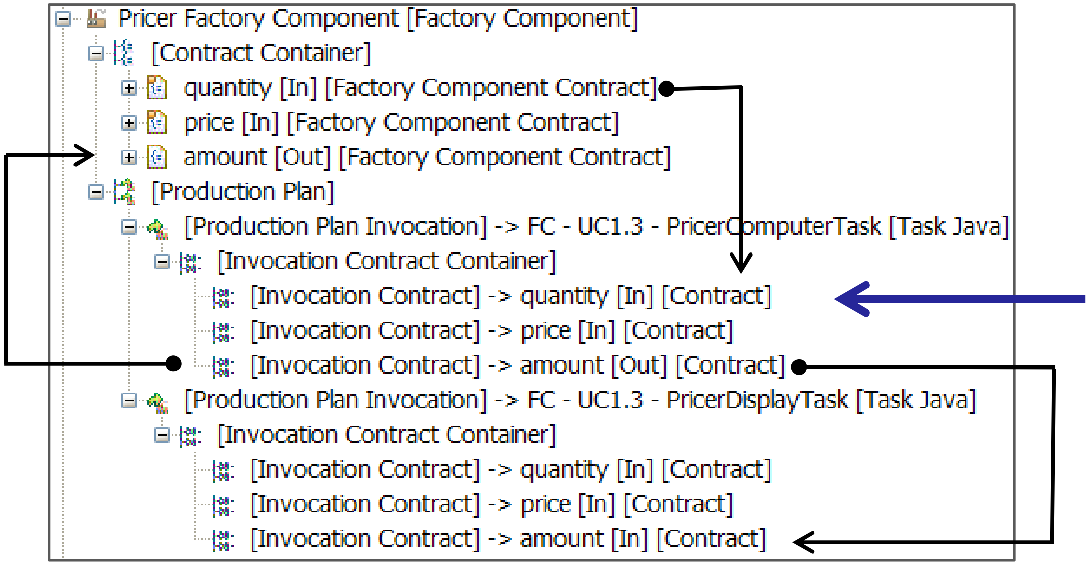


In/Output relationships



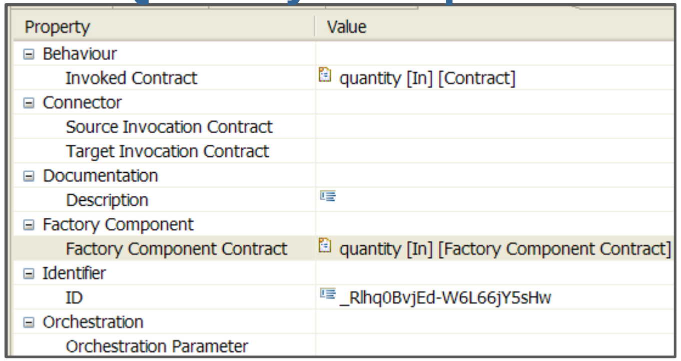
```
Pricer Factory Component [Factory Component]
  [Contract Container]
    quantity [In] [Factory Component Contract]
    price [In] [Factory Component Contract]
    amount [Out] [Factory Component Contract]
  [Production Plan]
    [Production Plan Invocation] -> FC - UC1.3 - PricerComputerTask [Task Java]
      [Invocation Contract Container]
        [Invocation Contract] -> quantity [In] [Contract]
        [Invocation Contract] -> price [In] [Contract]
        [Invocation Contract] -> amount [Out] [Contract]
    [Production Plan Invocation] -> FC - UC1.3 - PricerDisplayTask [Task Java]
      [Invocation Contract Container]
        [Invocation Contract] -> quantity [In] [Contract]
        [Invocation Contract] -> price [In] [Contract]
        [Invocation Contract] -> amount [In] [Contract]
```

In/Output relationships



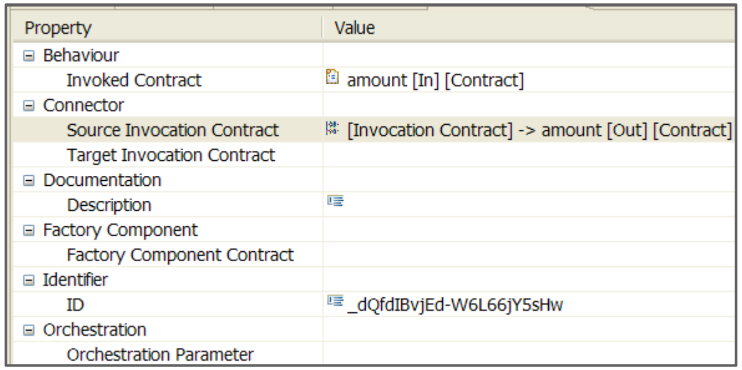
```
Pricer Factory Component [Factory Component]
  [Contract Container]
    quantity [In] [Factory Component Contract]
    price [In] [Factory Component Contract]
    amount [Out] [Factory Component Contract]
  [Production Plan]
    [Production Plan Invocation] -> FC - UC1.3 - PricerComputerTask [Task Java]
      [Invocation Contract Container]
        [Invocation Contract] -> quantity [In] [Contract]
        [Invocation Contract] -> price [In] [Contract]
        [Invocation Contract] -> amount [Out] [Contract]
    [Production Plan Invocation] -> FC - UC1.3 - PricerDisplayTask [Task Java]
      [Invocation Contract Container]
        [Invocation Contract] -> quantity [In] [Contract]
        [Invocation Contract] -> price [In] [Contract]
        [Invocation Contract] -> amount [In] [Contract]
```

Quantity's Properties



Property	Value
⊟ Behaviour	
Invoked Contract	quantity [In] [Contract]
⊟ Connector	
Source Invocation Contract	
Target Invocation Contract	
⊟ Documentation	
Description	
⊟ Factory Component	
Factory Component Contract	quantity [In] [Factory Component Contract]
⊟ Identifier	
ID	_Rlhq0BvjEd-W6L66jY5sHw
⊟ Orchestration	
Orchestration Parameter	

Amount's Properties



Property	Value
⊟ Behaviour	
Invoked Contract	amount [In] [Contract]
⊟ Connector	
Source Invocation Contract	[Invocation Contract] -> amount [Out] [Contract]
Target Invocation Contract	
⊟ Documentation	
Description	
⊟ Factory Component	
Factory Component Contract	
⊟ Identifier	
ID	_dQfdIBvjEd-W6L66jY5sHw
⊟ Orchestration	
Orchestration Parameter	

Agenda

- Introduction
- EGF Structure
- Pattern

Pattern – Definition

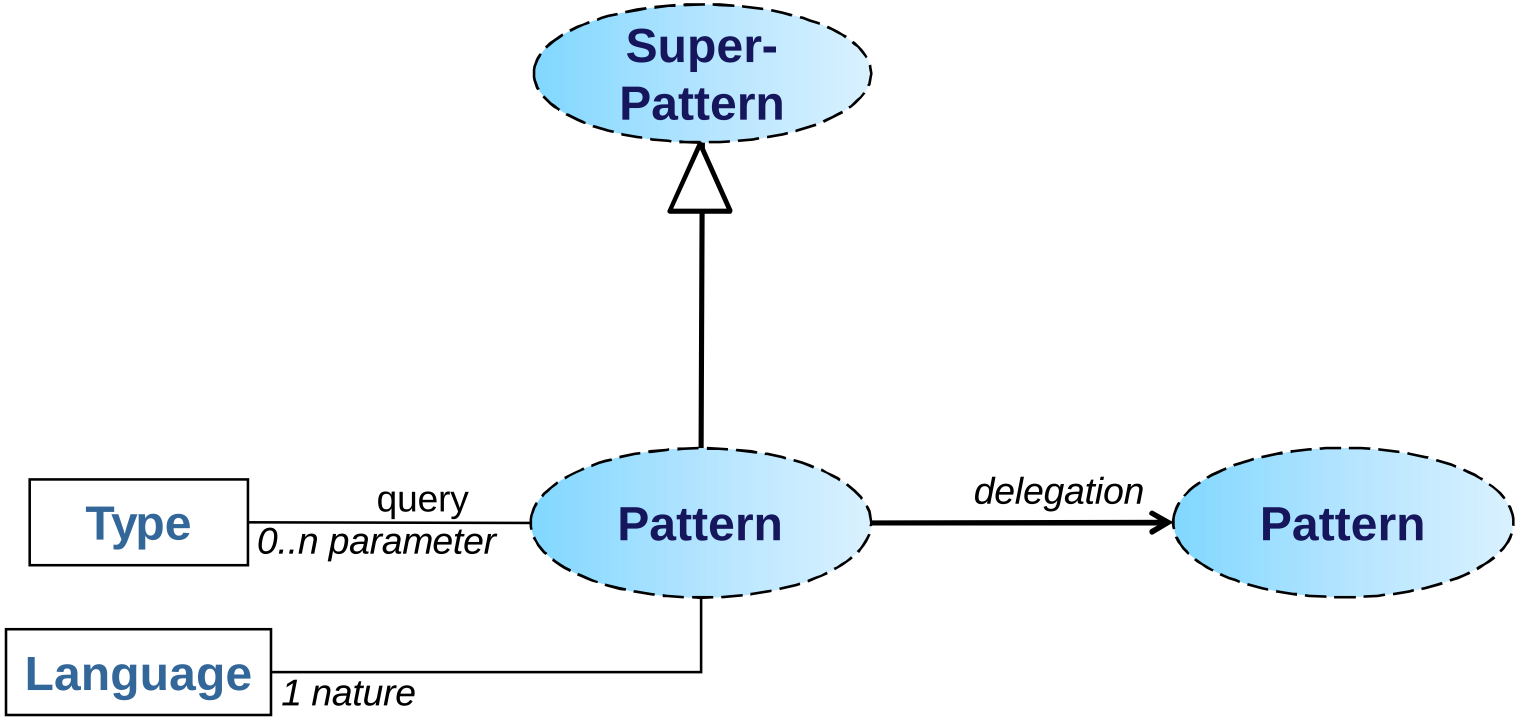
- **Definition:**
  - A pattern is a solution to a recurrent generation problem

- **Purpose**
  - Applying a systematic behavior onto a resource
  - Clearly dissociating the specification (external view) from the implementation (internal view) of the behavior
  - Reusing and customizing a pattern in different contexts
  - Supporting multilingual patterns in order to apply the best programming language to a situation, and then supporting multi-paradigm (M2T, M2M, T2M, T2T)

EGF Structure

Pattern Structure

Pattern – External View



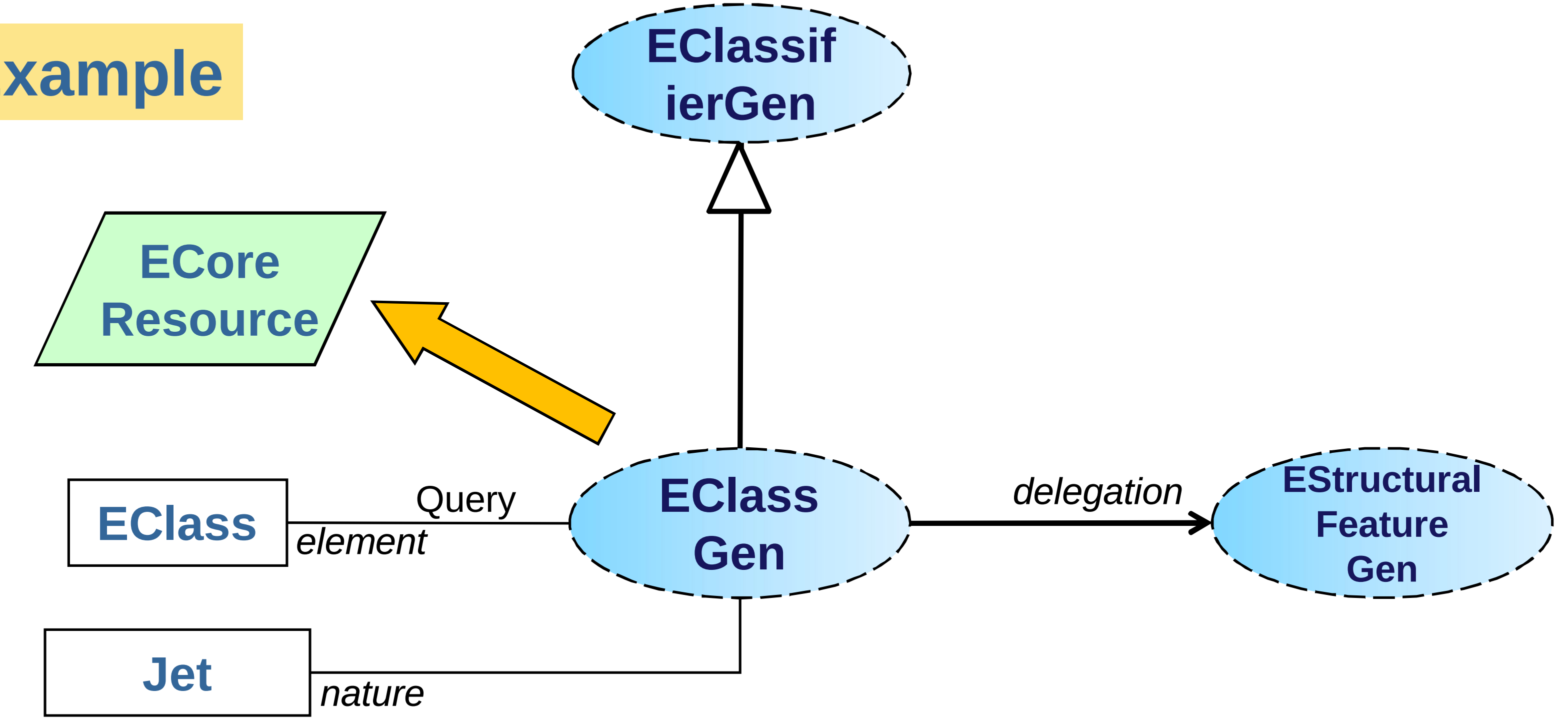
Definition:

- query/parameter: query for object selection onto a resource
- nature: language used for the pattern implementation

THALES

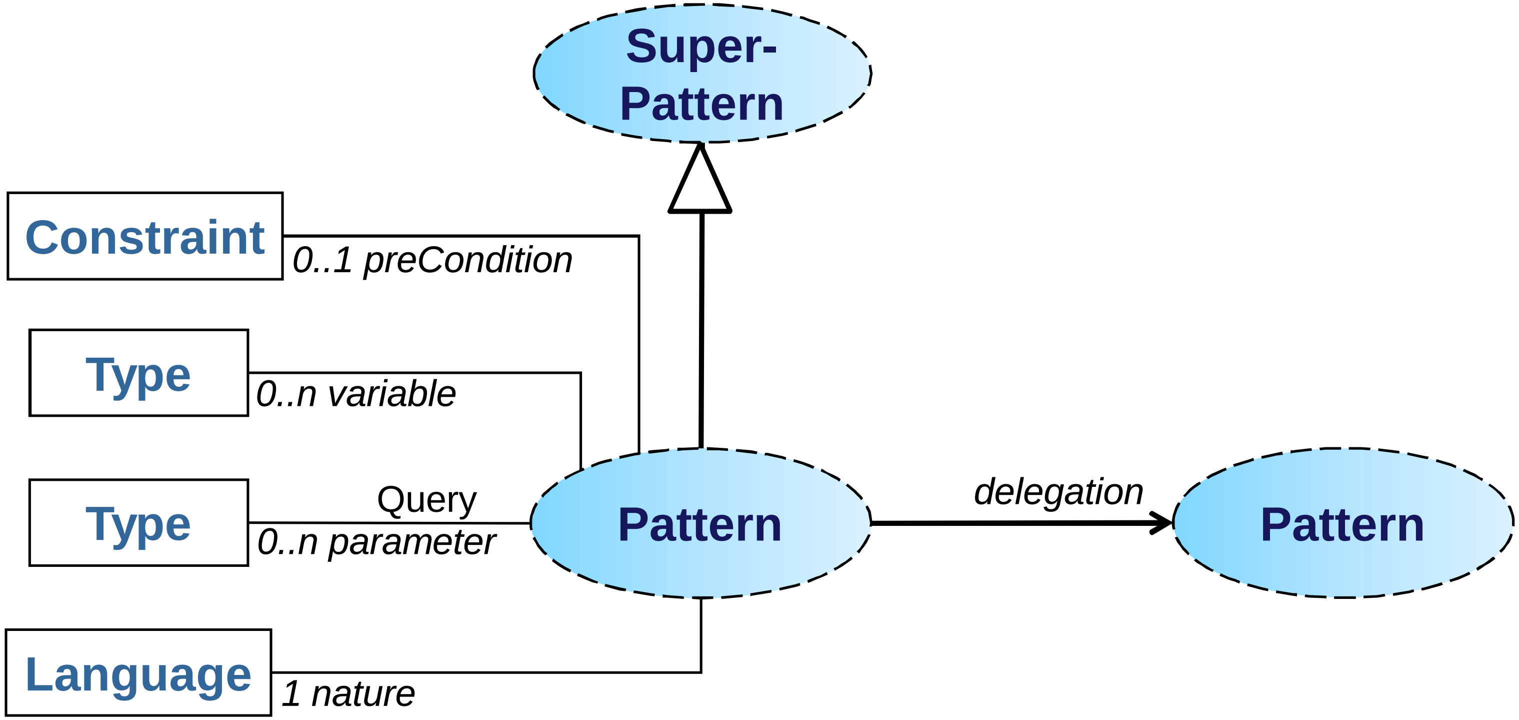
Pattern – External View

Example



- The EClassGen pattern is applied on a Ecore resource
- Objects selected on the ecore resource: EClass instances
- It specializes the EClassifierGen pattern
- It applies a model-to-text generation in Jet
- Its also applies a generation on its features by delegation to the EStructuralFeatureGen pattern

Pattern – External View



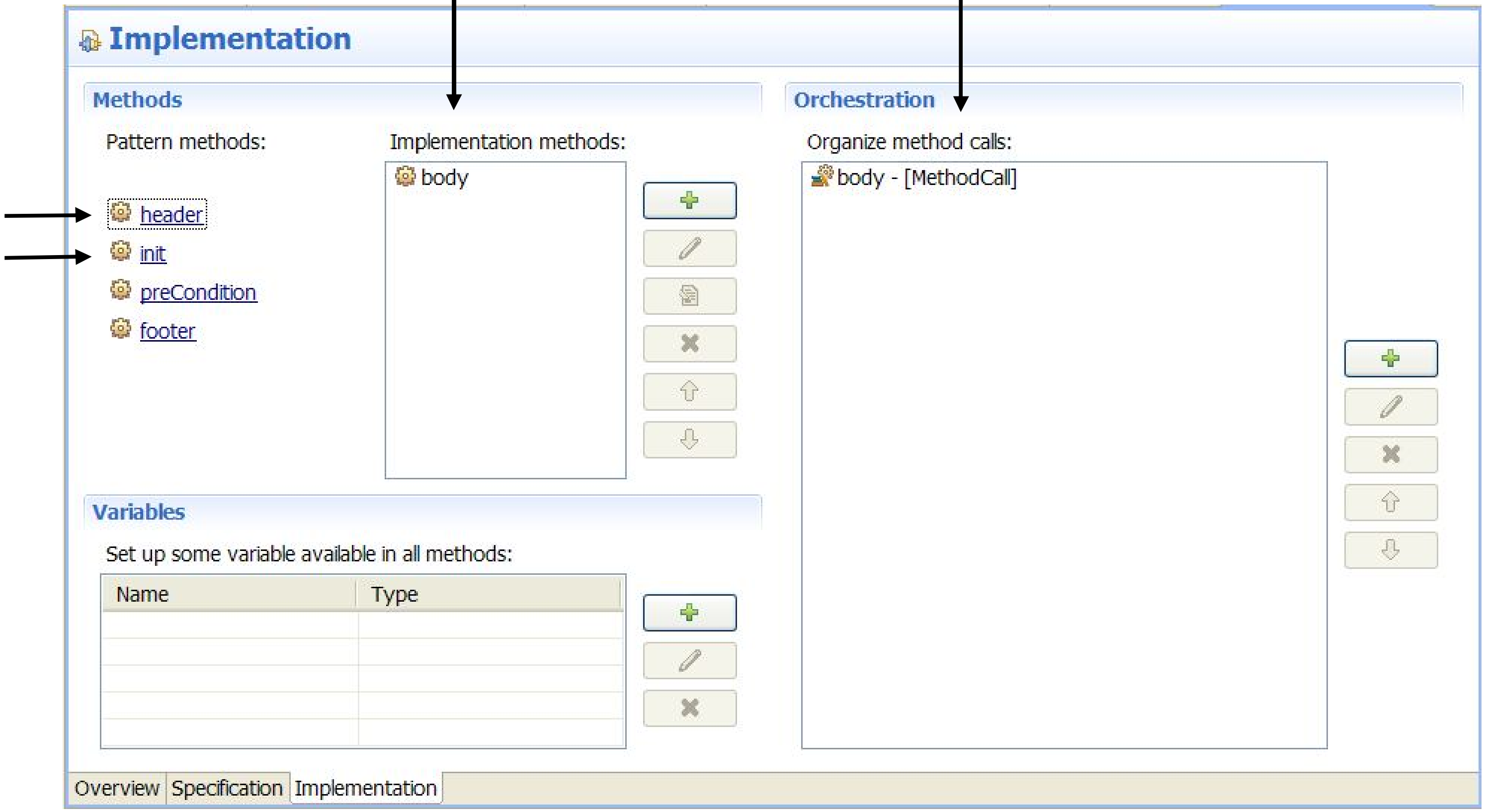
Definition:

- preCondition/Constraint: constraint to be verified before application
- variable/Type: local variable declaration for the pattern implementation

Pattern – Implementation View

Methods which implement the pattern

Order to execute the methods



Implementation

Methods

Pattern methods:

- header
- init
- preCondition
- footer

Implementation methods:

- body

Orchestration

Organize method calls:

- body - [MethodCall]

Variables

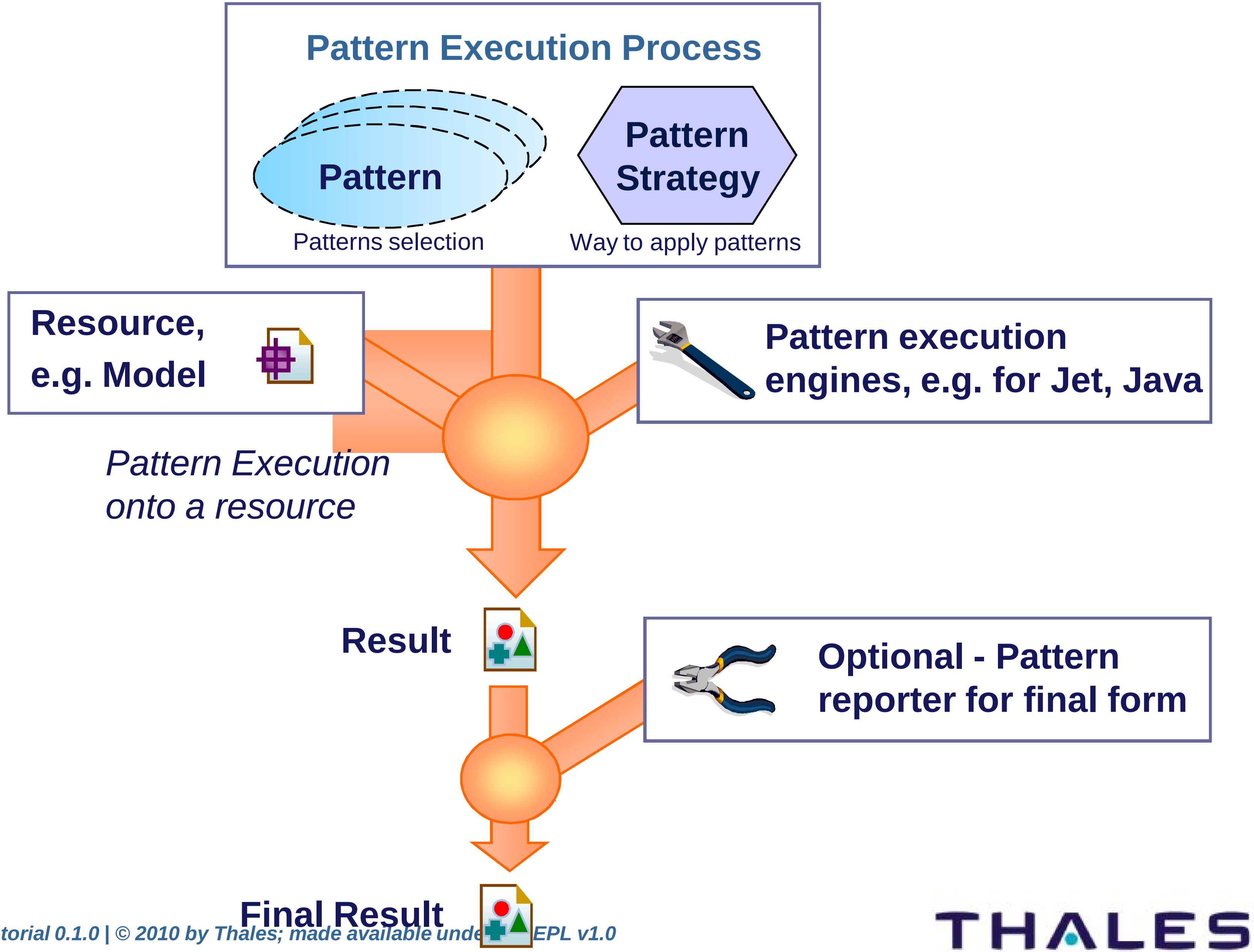
Set up some variable available in all methods:

Name	Type

header: typically used for the Jet header
init: method for pattern initialization (e.g., variable initialization)
A method editor allows to edit pattern methods

Pattern Execution

Pattern execution Big Picture


Pattern Execution Process
Pattern
Patterns selection
Pattern Strategy
Way to apply patterns
Resource, e.g. Model
Pattern execution engines, e.g. for Jet, Java
Pattern Execution onto a resource
Result
Optional - Pattern reporter for final form
Final Result

Pattern Strategy

- **Definition: Way to apply patterns against a resource**

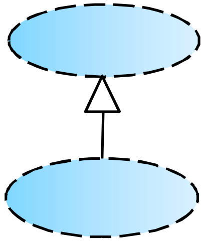
- **Examples of strategies:**
  - *Model-driven pattern strategy*: in-depth navigation over a model, and for each model element, applying a set of patterns
  - *Pattern-driven strategy*: for each pattern, applying the pattern for each model element element
  - *[Data type]-driven strategy*: generalization of the approach; instead of model, it could be any type of resource (e.g., file directory)

- **Strategy parameters:**
  - Resource visitor: the “for each” navigation is a specific case; the visitor function specifies how to navigate over a resource. Examples: in-large navigation, considering only Eclassifiers

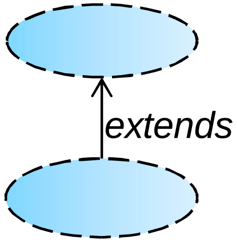
Pattern Composition

Pattern Relationships

Pattern inheritance

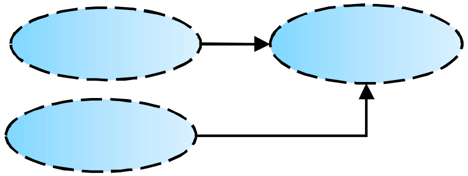


Pattern extension

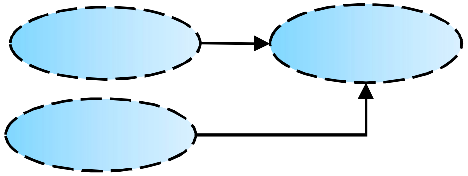


(not available yet)

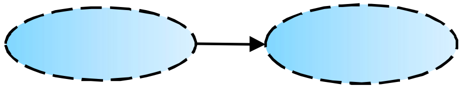
Pattern delegation



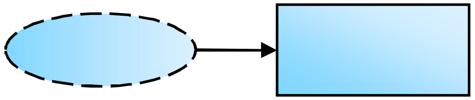
Pattern injection



Multilingual call

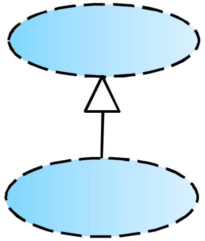


Pattern callback



The Pattern orchestration specifies the pattern relationships
Possibility to combine different pattern relationships in the same orchestration

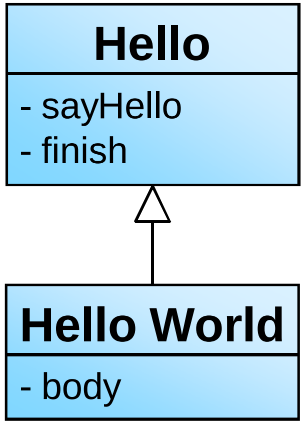
Pattern inheritance



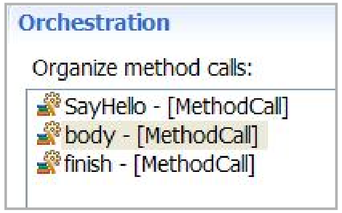
Case 1. Reuse of super-pattern methods

Same mechanism than Class inheritance
Selection of methods from the super-pattern hierarchy

Example



Orchestration of HelloWorld

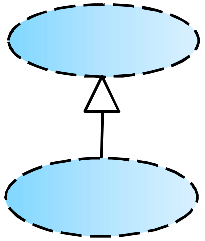


Orchestration

Organize method calls:

SayHello - [MethodCall]
body - [MethodCall]
finish - [MethodCall]

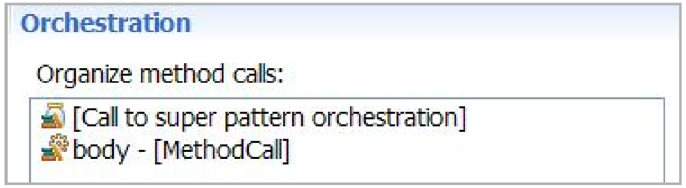
Pattern inheritance



Case 2. Reuse of super-pattern orchestration

Reuse of the orchestration defined in the super-pattern
This abstracts the super-pattern orchestration
This avoids rewriting pattern orchestration
Just adding the methods of the current pattern

Example



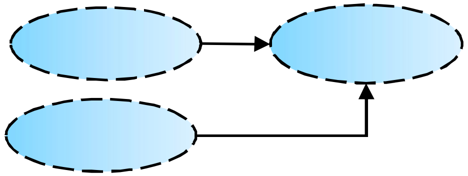
Orchestration

Organize method calls:

[Call to super pattern orchestration]
body - [MethodCall]

Pattern Delegation

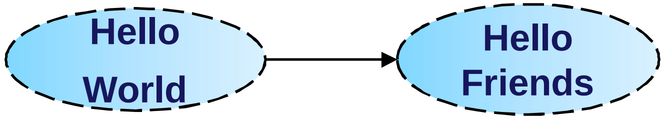
Pattern delegation

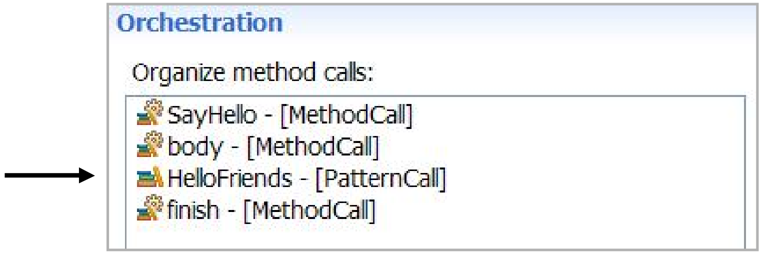


Case. Problem decomposition / Reuse of pattern

- The same pattern is reused in different pattern contexts
- The orchestration of the called pattern is applied
- The Pattern caller provides parameter values to the called pattern
- The parameter values are statically declared at the pattern definition

Example 1



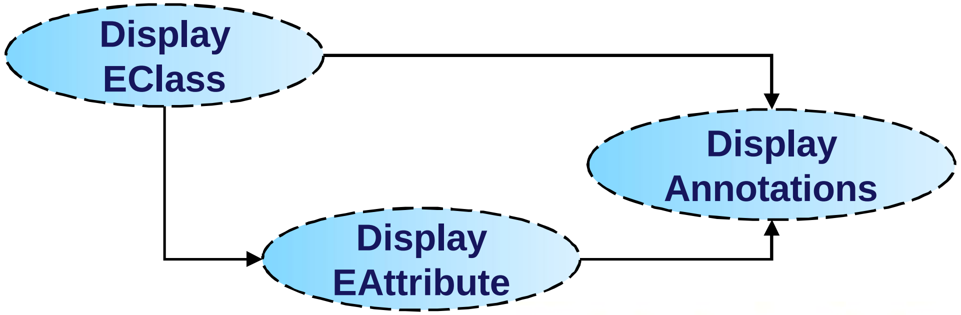


Orchestration

Organize method calls:

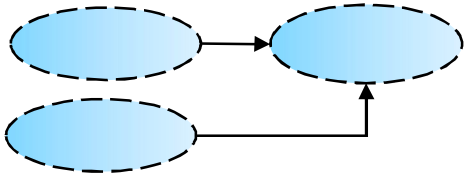
SayHello - [MethodCall]
body - [MethodCall]
HelloFriends - [PatternCall]
finish - [MethodCall]

Example 2



Pattern Injection

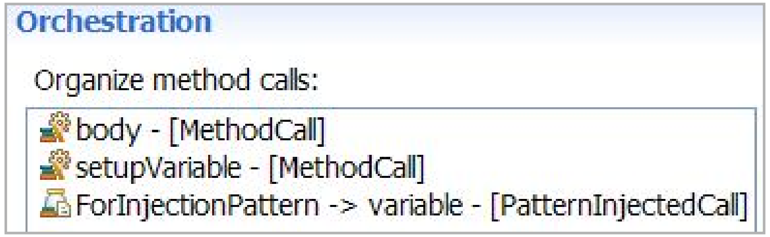
Pattern injection



Case. Reuse of pattern with a dynamic resolution of the injected context

- A Pattern injection corresponds to a Pattern Delegation, but
- The parameter values are dynamically set at pattern execution

Example



Orchestration

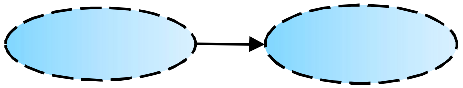
Organize method calls:

body - [MethodCall]
setupVariable - [MethodCall]
ForInjectionPattern -> variable - [PatternInjectedCall]

In this example, the “setupVariable” method sets the injection context

Multilingual Call

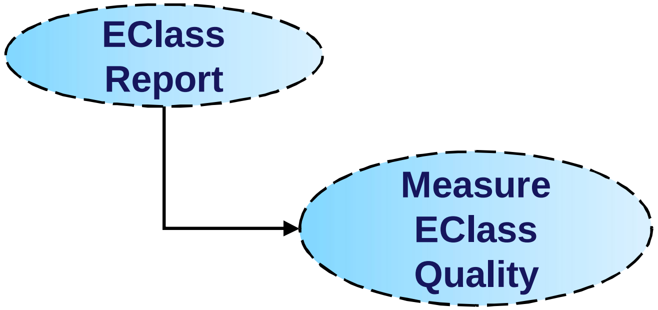
Multilingual Call



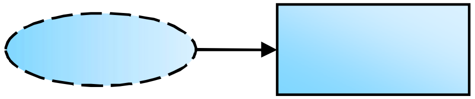
Case. Pattern delegation where implementation languages are different

This corresponds to a Pattern Delegation where Pattern natures are different. For instance, a Pattern with a Jet nature calls a Pattern with a Java nature in order to differently process the same resource.
It is impossible to have different natures in the same Pattern inheritance hierarchy.

Example



Pattern Callback

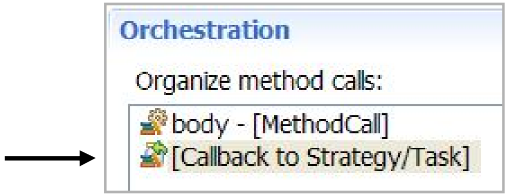


Case 1. Applying a Java call

The callback indicates where the callback on a Java Class is applied

Example

Pattern orchestration

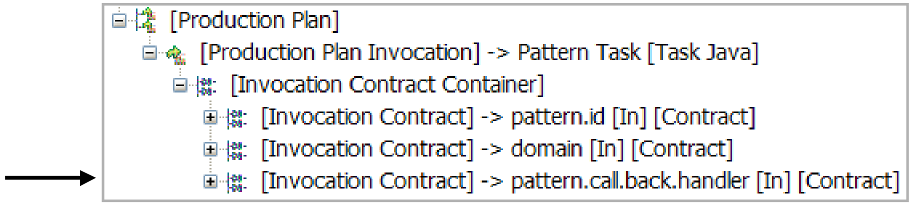


Orchestration

Organize method calls:

- body - [MethodCall]
- [Callback to Strategy/Task]

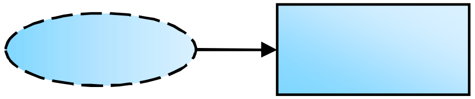
Specification of the Java Class in the production plan



```
[Production Plan]
  [Production Plan Invocation] -> Pattern Task [Task Java]
    [Invocation Contract Container]
      [Invocation Contract] -> pattern.id [In] [Contract]
      [Invocation Contract] -> domain [In] [Contract]
      [Invocation Contract] -> pattern.call.back.handler [In] [Contract]
```

Pattern Callback

Pattern Callback



Case 2. Combination with the Pattern Strategy

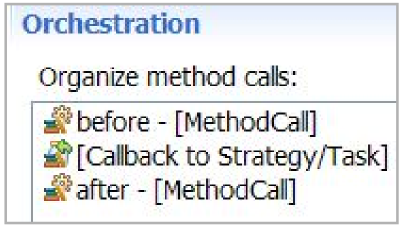
A strategy determines how to apply patterns and how to navigate over a resource. In an orchestration, a callback is the moment before and after a cycle of pattern application, and allows to discriminate the methods to apply before and after it.

Example

Scenario:
The following generation result can be realized with a callback.
- The model-driven strategy navigates over the model
- There is a pattern for each kind of model element with the following pattern orchestration
A generation action is realized before (open) and after (close) the callback.

```
<EPackage name=“P”>
  <EClass name=“C1”>
    <EAttribute = “A1”>
        …
    </EAttribute = “A1”>
  </EClass name=“C1”>
</EPackage name=“P”>
```

Generation result



Orchestration

Organize method calls:

before - [MethodCall]
[Callback to Strategy/Task]
after - [MethodCall]