PATTERN

Author: Benoît Langlois – benoit.langlois@thalesgroup.com

Version: 1.0

DEFINITION

A Pattern is a means to apply a systematic transformation (e.g., model-to-text) onto a resource.

OBJECTIVES

The objectives of a Pattern are to:

- Provide a formalism to express systematic transformation onto a resource.

The interests are to:

- Create and promote transformation portfolios,
- Adapt a portfolio to a new context and promote it as a new portfolio.

CONCERNS

Designer	• In a production plan of a Factory Component, the designer declares patterns used and the way to apply them.
Developer	• The developer defines the pattern specification (i.e., the external view of a pattern). • The developer implements the pattern implementation (i.e., the internal view of a pattern).

PATTERN

STRUCTURE

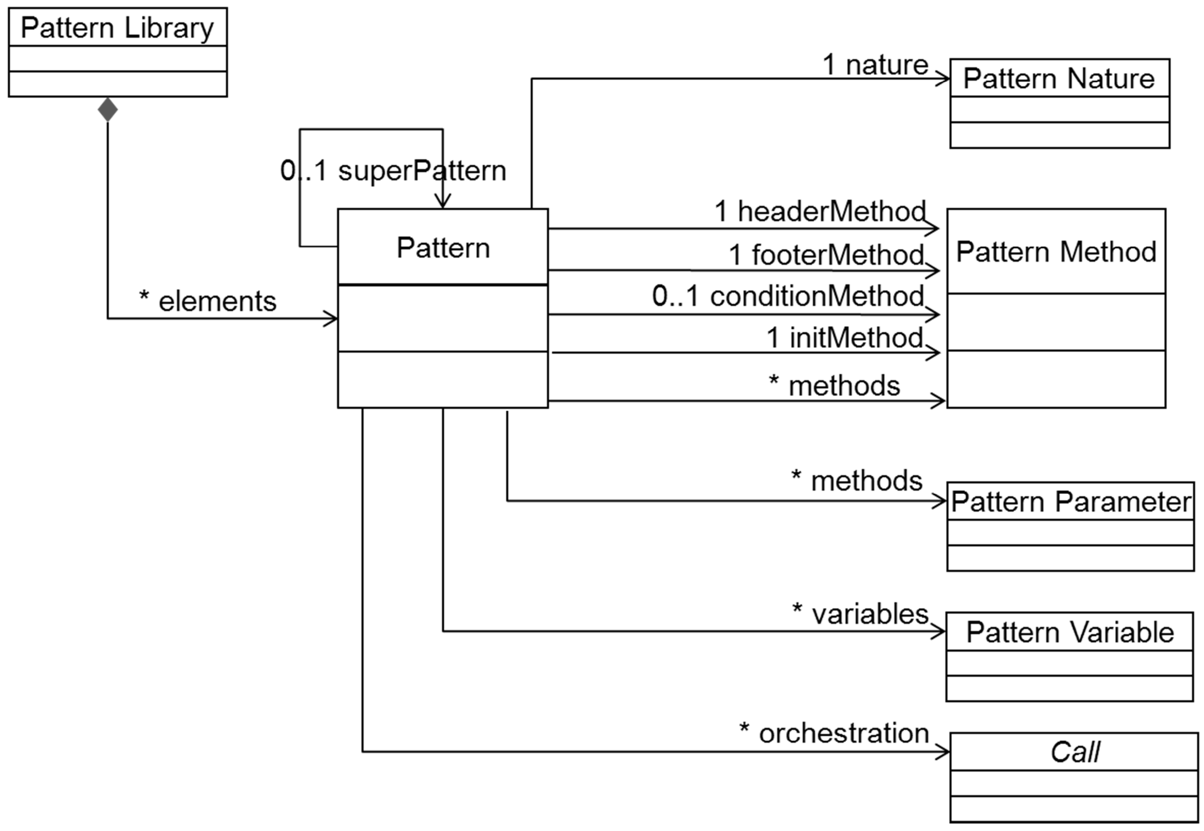
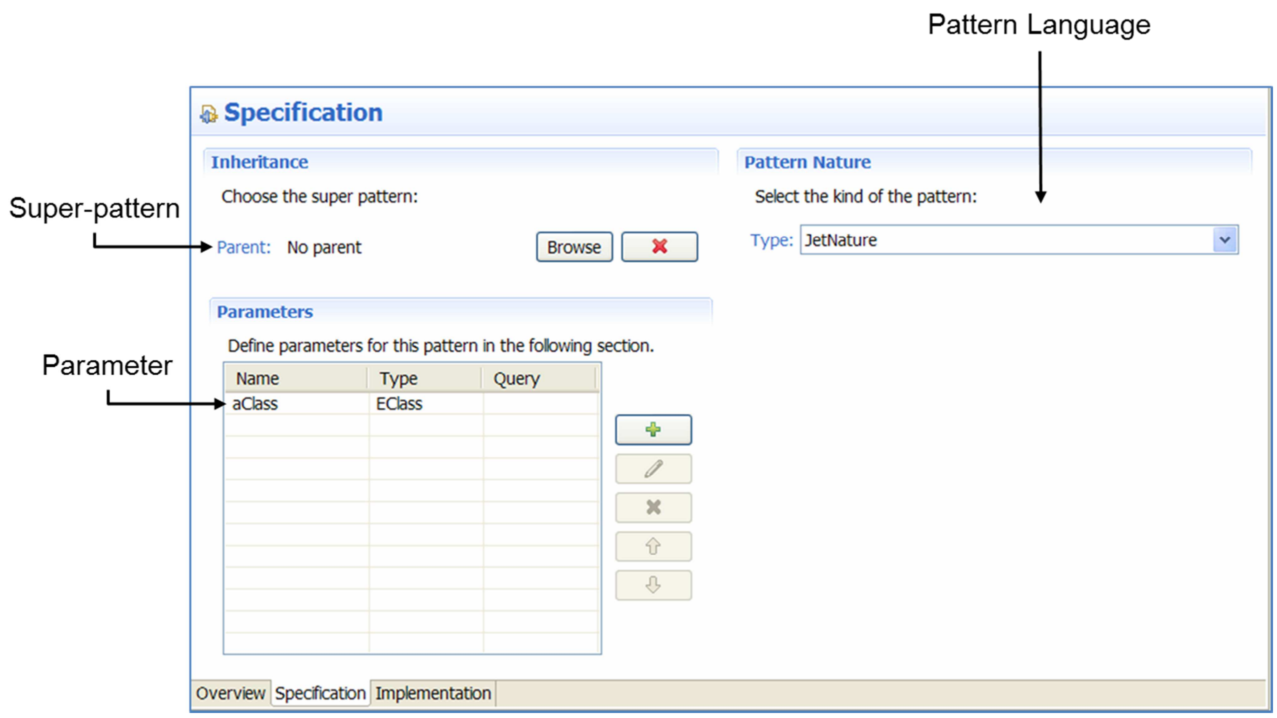


Figure 1. Pattern at the metamodel level

Comprehension:

- A Pattern Library contains a set of Patterns.
- A Pattern is a declarative formalism to process a resource. For instance, a set of Package, Class, Attribute and Operation Patterns are applied over a model which is a hierarchy of Packages, Classes, Attributes and Operations.
- For presentation, a Pattern is made of two main parts:
    - *Specification part*. A Pattern has a name and has a Nature, especially to identify the language used to implement the Pattern. A Pattern can inherit from a Pattern for inheritance of all the Pattern properties (e.g., Variables, Methods). In the specification part, Pattern Parameters define the Pattern call context, like a method is called with parameters. Default Pattern Parameters are Ecore metaclasses but the Pattern Parameter type is open to any Emf class and Java type. (Cf. Figure 2.)
    - *Implementation part*. This part identifies the local Variables, the Pattern Methods, and the Method orchestration. (Cf. Figure 3.)
- Pattern Methods. Several types of Pattern Methods exist:

    - A Header method localizes the declaration part of the Pattern implementation.
    - A set of Pattern methods are declared with their name and description corresponding to a method body. This description conforms to the language identified by the Pattern Nature. For instance, Pattern with the Java Nature has methods implemented in Java.
    - The Footer method localizes the final declaration of a Pattern implementation.
    - An Init Method enables to initialize the Pattern Variables.
    - The PreCondition Method enables to execute or not a Pattern when the PreCondition is satisfied or not.
- Pattern Orchestration. It enables to order the Pattern Method calls. Several kinds of Pattern Calls exist:
    - *Method Call*. It is a simple call a Pattern Method identified by its name (e.g., body).
    - *Super-Pattern Call*. This enables to apply the orchestration defined in the Super-Pattern, and this recursively in the Pattern hierarchy. Patterns in the same Pattern hierarchy must have the Pattern Nature.
    - *Pattern Call*. This enables to call a Pattern as a Pattern Method. It offers the mechanism of delegation. A Pattern Parameter association links each Parameter of the called Pattern to a Pattern Parameter value of the calling Pattern. Pattern Call allows calling Patterns with different Pattern Natures (e.g., a Java Pattern can call a Jet Pattern, and reciprocally).
    - *Callback*. Suppose that you want to automatically open and close sections at different levels of definition, such as a Package contains Packages or Classes, and a Class contains Attributes or Operations. A Callback method is a kind of breaking point: all the Calls before the Callback correspond to a begin section with the current Pattern context (e.g., the Class Pattern context); Calls after the Callback correspond to the end section; a Callback method enables to delegate work to other Patterns (e.g., to Attribute and Operation Patterns when it is time to process Attributes and Operations of the current Class).
    - *Pattern Injection Call*. A Pattern Injection corresponds to a set of Pattern Calls for each value of a query result.



Specification

Inheritance

Choose the super pattern:

Parent: No parent

Pattern Nature

Select the kind of the pattern:

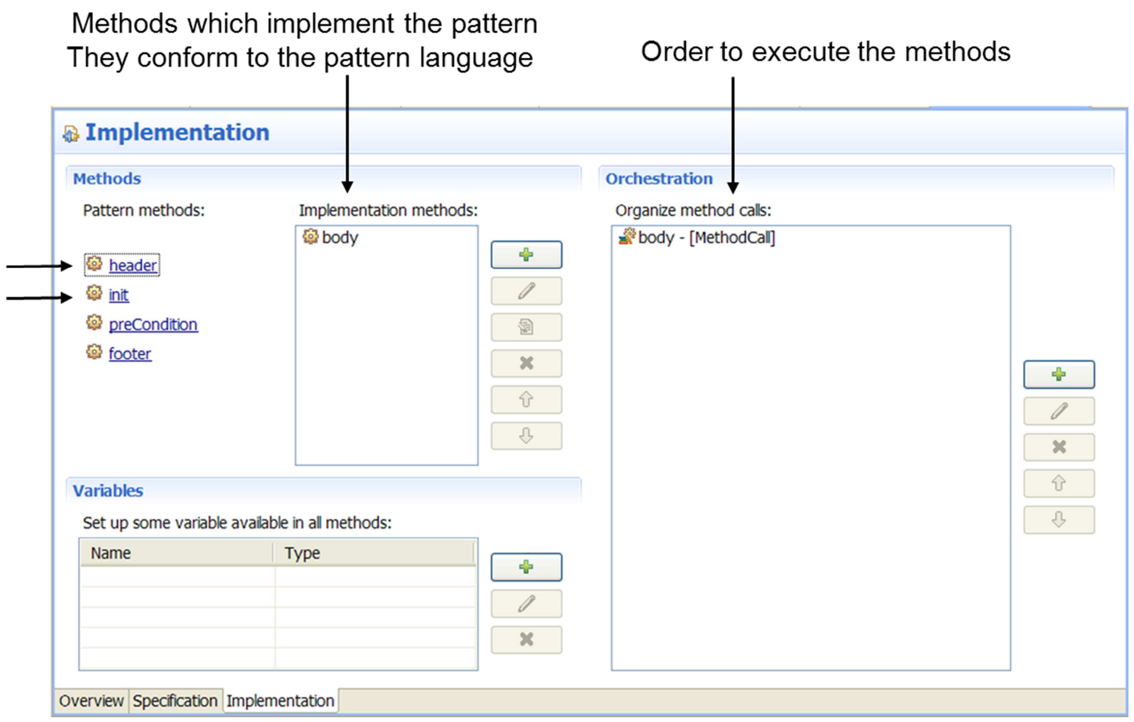
Type: JetNature

Parameters

Define parameters for this pattern in the following section.

Name	Type	Query
aClass	EClass	

Figure 2. Pattern Specification part


Methods which implement the pattern
They conform to the pattern language
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:

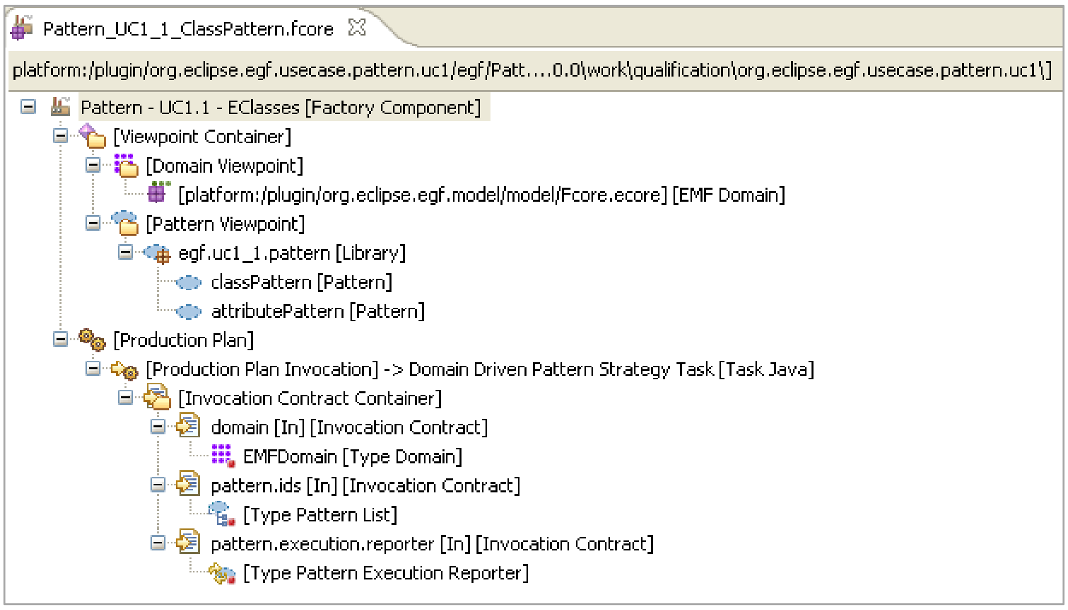
Name	Type

Overview
Specification
Implementation

Figure 3. Pattern Implementation part

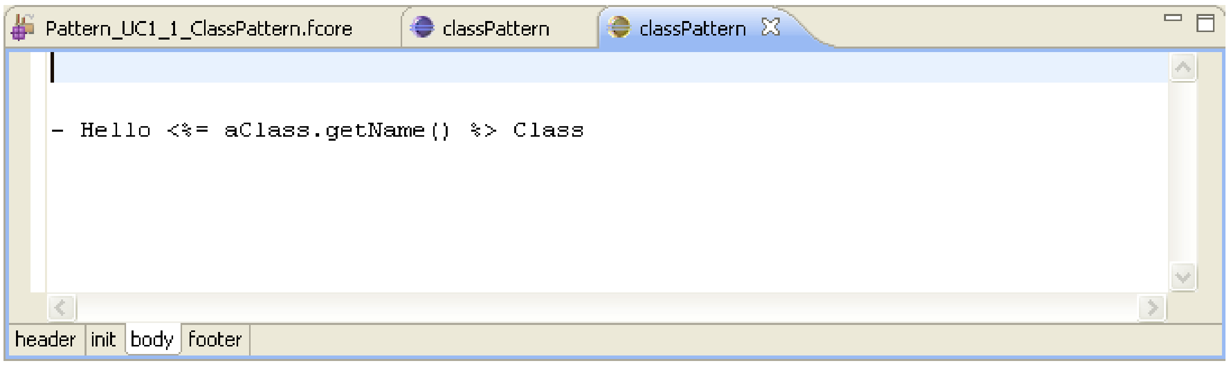
EXAMPLES

Figure 4 shows a Factory Component with two Patterns, “classPattern” and “attributePattern, which respectively display the Class names, and the Attributes names of each Class. The specification and implementation parts of the classPattern are shown in Figure 2 and Figure 3. Code of the classPattern and attributePattern body methods (see Figure 5 and Figure 6) is written in Jet. The Domain is the EGF Fcore model (see Figure 4 in the Domain Viewpoint part). Patterns are applied with the “Domain Driven Pattern Strategy” Task (see Figure 4 in the Production Plan) which applies Patterns over the Fcore model. The result is shown in Figure 7.



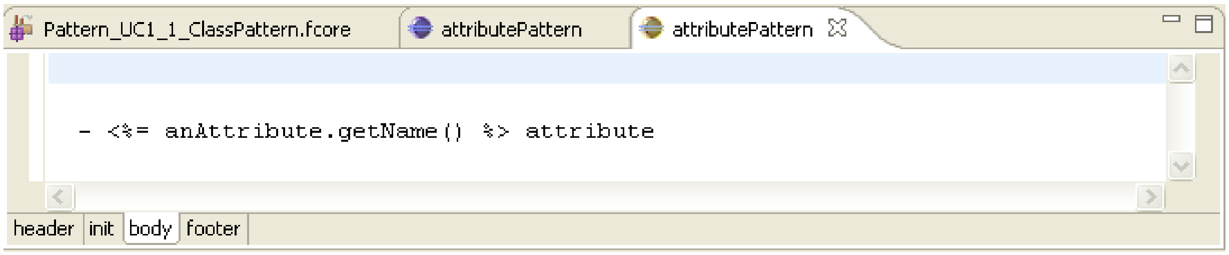
```
Pattern_UC1_1_ClassPattern.fcore
platform:/plugin/org.eclipse.egf.usecase.pattern.uc1/egf/Patt....0.0\work\qualification\org.eclipse.egf.usecase.pattern.uc1\]
Pattern - UC1.1 - EClasses [Factory Component]
  [Viewpoint Container]
    [Domain Viewpoint]
      [platform:/plugin/org.eclipse.egf.model/model/Fcore.ecore] [EMF Domain]
    [Pattern Viewpoint]
      egf.uc1_1.pattern [Library]
        classPattern [Pattern]
        attributePattern [Pattern]
  [Production Plan]
    [Production Plan Invocation] -> Domain Driven Pattern Strategy Task [Task Java]
      [Invocation Contract Container]
        domain [In] [Invocation Contract]
          EMFDomain [Type Domain]
        pattern.ids [In] [Invocation Contract]
          [Type Pattern List]
        pattern.execution.reporter [In] [Invocation Contract]
          [Type Pattern Execution Reporter]
```

Figure 4. *Example of Factory Component with Class and Attribute Patterns*



```
- Hello <%= aClass.getName() %> Class
```

Figure 5. Implementation of the classPattern body method



```
- <%= anAttribute.getName() %> attribute
```

Figure 6. Implementation of the attributePattern body method

```
--------------
Result of pattern:

- Hello ModelElement Class
  - iD attribute
  - description attribute
- Hello NamedModelElement Class
  - name attribute
- Hello Activity Class
- Hello Contract Class
  - mandatory attribute
  - mode attribute
- Hello FactoryComponent Class
- Hello ContractContainer Class
- Hello FactoryComponentContract Class
- Hello ViewpointContainer Class
- Hello Viewpoint Class
- Hello Orchestration Class
- Hello OrchestrationParameterContainer Class
- Hello OrchestrationParameter Class
- Hello Invocation Class
- Hello InvocationContractContainer Class
- Hello InvocationContract Class
```

Figure 7. Result in the console

PATTERN

PATTERN STRATEGIES

A Pattern is only a process unit. As quickly introduced in the example, a Strategy determines the way to apply Patterns over a resource (e.g., an Ecore model, File). The following sections present the two Strategies provided with EGF. Other strategies can be developed (e.g., in-large navigation, navigation managing proprieties for model transformations). A Pattern Strategy, which is a Task, requires parameter values (i.e., contract values), which can be considered as the interface signature of the Strategy. Those parameters emphasize the interest of decoupling concerns, for instance decoupling model-to-text transformation from its output devoted to a reporter which can be changed at any time when it is considered as a parameter value.

Domain Driven Pattern Strategy Task

Strategy description

While the domain resource is parsed (e.g., an EMF model), a set of patterns are applied.

Interface

Name	*Type*	*Description*	*Optional*
domain	Domain	Domain resource to be processed by the patterns.	No
pattern.execution.reporter	PatternExecutionReporter	Reporter Class responsible for the output of model-to-text transformation. The default output is the console.	Yes
pattern.call.back.handler	PatternCallBackHandler	Callback Class called for a Java callback.	Yes
pattern.domain.driven.visitor	PatternDomainVisitor	Visitor Class called when each domain element is visited.	Yes
pattern.ids	Ordered list of Patterns and PatternLibrary	Patterns or Pattern Libraries to be applied onto a resource.	No
pattern.substitutions	PatternSubstitution	A Pattern substitution is a list of Pattern replacements. A replacement consists in replacing Patterns in pattern.ids by other patterns.	Yes
Pattern.output.processor	PatternOutputProcessor	For model-to-text transformation, a Class which post-processes the result of a reporter.	Yes

Algorithm

```
In-depth navigation over a domain. For each Domain element:
    For each Pattern of pattern.ids:
        Apply the Pattern on the current element
```

Pattern Driven Strategy Task

Strategy description

An ordered list of patterns is successively applied to each element of a domain resource.

Interface

Same interface than the “Domain Driven Pattern Strategy Task”.

Algorithm

```
For each Pattern of Pattern Libraries:
    In-depth navigation over a domain. For each Domain element:
        Apply the Pattern on the current element
```

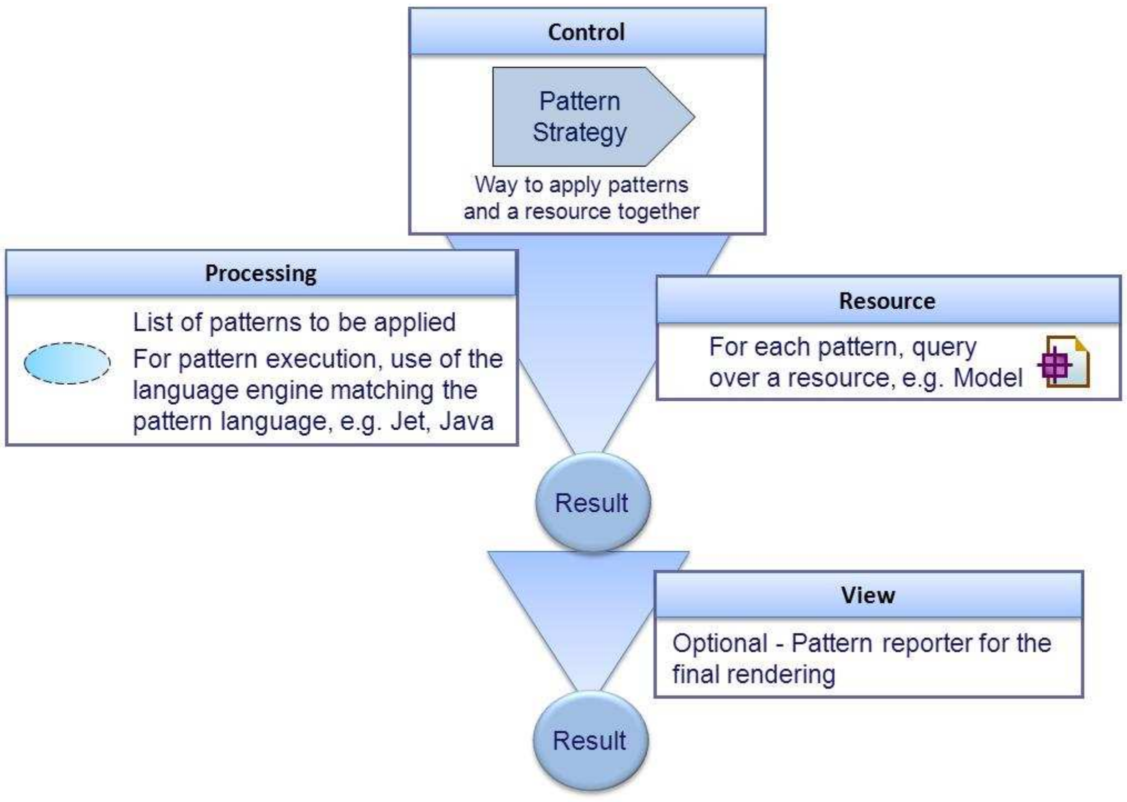
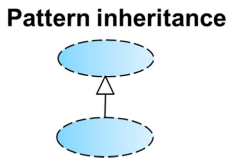

Control
Pattern Strategy
Way to apply patterns and a resource together
Processing
List of patterns to be applied
For pattern execution, use of the language engine matching the pattern language, e.g. Jet, Java
Resource
For each pattern, query over a resource, e.g. Model
Result
View
Optional - Pattern reporter for the final rendering
Result

Figure 8. General process with the application of a Pattern Strategy

PATTERN RELATIONSHIPS

An issue with Patterns is to articulate Patterns together and Pattern Strategy Parameter values. This section explains every kind of Pattern relationship, even if some of them were already presented.

Pattern inheritance



Like Class inheritance, Pattern inheritance enables to communalize properties (i.e., Pattern Parameters, Pattern variables, methods, method orchestration) in a mono-inheritance Pattern hierarchy.

The following picture shows that the HelloWorld Pattern inherits of methods from its Hello Super-Pattern for its orchestration.

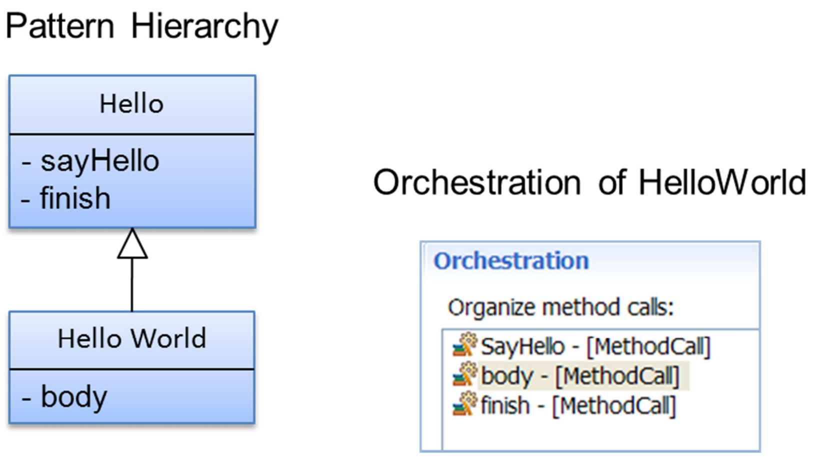
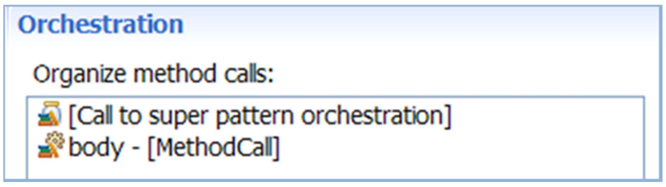


Figure 9. Super-Pattern method inheritance

A “Call to super pattern orchestration” in the Pattern orchestration enables to abstract and ignore the Super-Pattern orchestration.



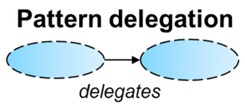
Orchestration

Organize method calls:

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

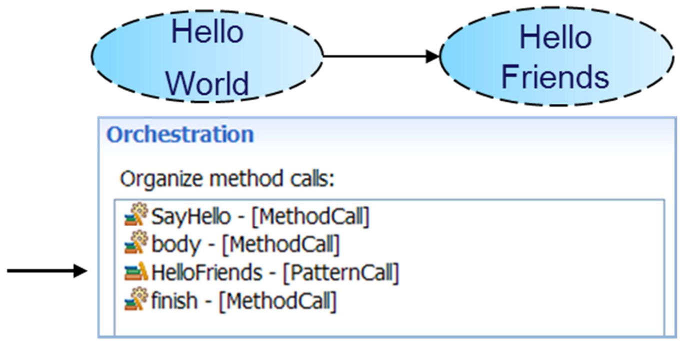
Figure 10. Super-Pattern method inheritance

Pattern delegation (aka Pattern call)



In a Pattern orchestration, Pattern delegation enables problem decomposition and reuse of patterns in different 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.

The following picture presents the call to the HelloFriends Pattern by the HelloWorld Pattern.



Orchestration

Organize method calls:

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

Figure 11. Pattern delegation – Pattern Call in the Pattern orchestration

The following picture shows how to reuse a common behavior, Displaying Annotations, for the Ecore EClass and EAttribute Classes.

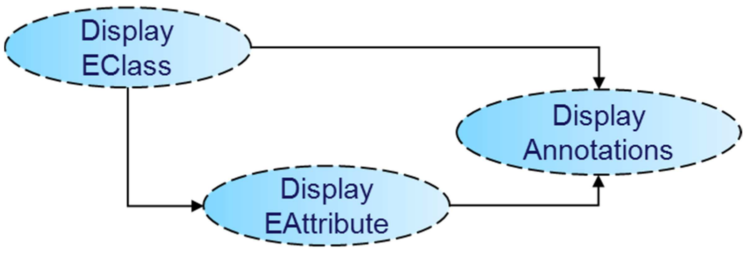
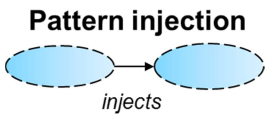


Figure 12. Pattern delegation – Reuse of behavior

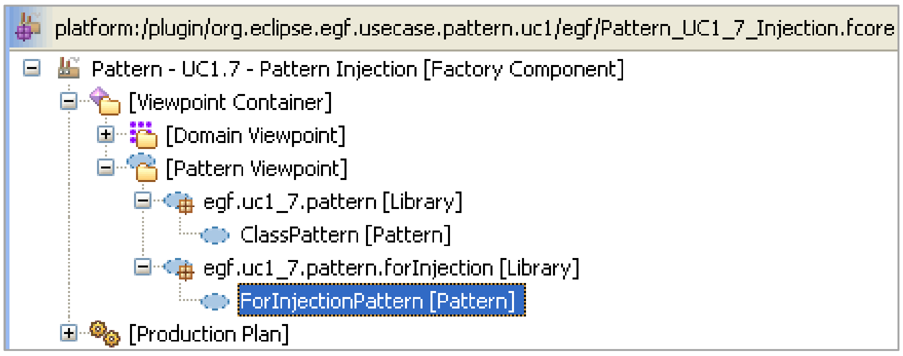
The multilingual call corresponds to a Pattern Delegation where Pattern natures are different. For instance, a Pattern with a Jet nature calls a Pattern with a Java nature in order to differently process the same resource. A warning: the model-to-text processing and Java call have two different lifecycles; then, the result of the Java calls can be achieved before the end of the Jet processing.

Pattern Injection



A Pattern injection corresponds to a Pattern Delegation, but the value of a Pattern parameter is dynamically set at pattern execution with a query.

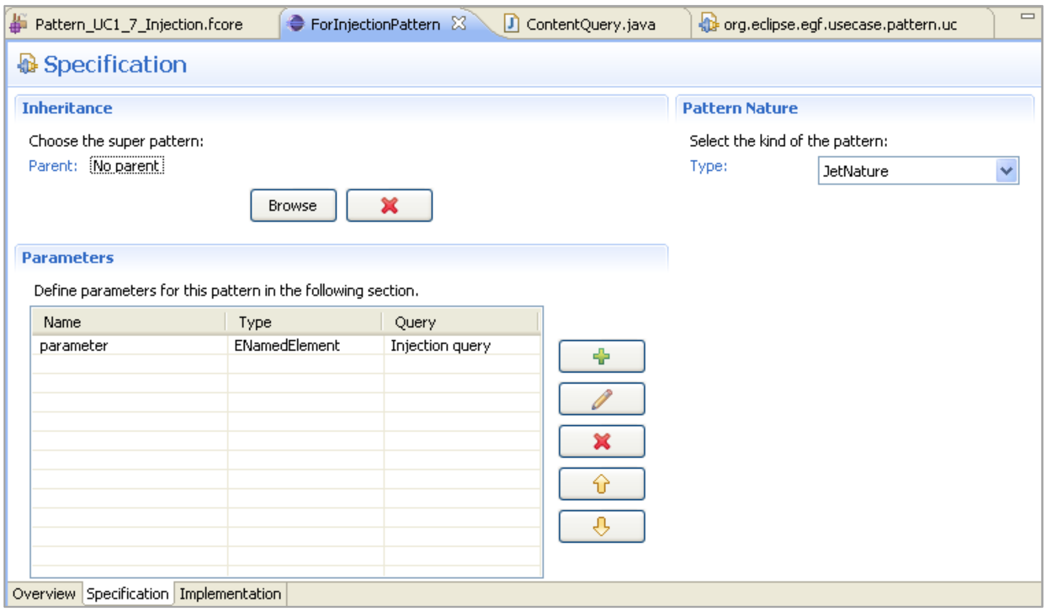
In the following example, "ClassPattern" calls by injection "ForInjectionPattern".



```
platform:/plugin/org.eclipse.egf.usecase.pattern.uc1/egf/Pattern_UC1_7_Injection.fcore
Pattern - UC1.7 - Pattern Injection [Factory Component]
  [Viewpoint Container]
    [Domain Viewpoint]
    [Pattern Viewpoint]
      egf.uc1_7.pattern [Library]
        ClassPattern [Pattern]
      egf.uc1_7.pattern.forInjection [Library]
        ForInjectionPattern [Pattern]
  [Production Plan]
```

Figure 13. Pattern injection - Patterns

The Pattern Parameter of the injected Parameter is associated to a query ("Injection query" in the example). The Pattern is applied for each element of the query result, in the example for each ENamedElement of the Class.



Specification

Inheritance

Choose the super pattern:

Parent: No parent

Pattern Nature

Select the kind of the pattern:

Type: JetNature

Parameters

Define parameters for this pattern in the following section.

Name	Type	Query
parameter	ENamedElement	Injection query

Figure 14. Pattern injection – Pattern for injection – Specification part

In order to be recognized, in the plugin.xml, an extension declares the query name and Class which implements the query.

```
<extension
      point="org.eclipse.egf.pattern.query">
   <query
         class="org.eclipse.egf.usecase.pattern.uc1.query.ContentQuery"
         id="org.eclipse.egf.usecase.pattern.uc1.query1"
         name="Injection query">
   </query>
</extension>
```

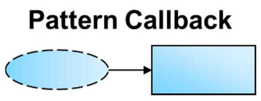
Figure 15. Example of extension for a query declaration

The following code exemplifies a query implementation.

```
public class ContentQuery implements IQuery {

    public List<Object> execute
      (ParameterDescription parameter,
       Map<String, String> queryCtx, PatternContext context) {
        String type = parameter.getType();
        Object loadClass =
            RuntimeParameterTypeHelper.INSTANCE.loadClass(type);
        if (!(loadClass instanceof EClass))
            throw new IllegalStateException(EGFPatternMessages.query_error1);

        Collection<EObject> domain =
            ((EObject) context.getValue(PatternContext.INJECTED_CONTEXT))
                  .eContents();
        if (domain == null)
            throw new IllegalStateException(EGFPatternMessages.query_error8);

        SELECT query = new SELECT(new FROM(domain),
                        new WHERE(new EObjectTypeRelationCondition((EClass)
                        loadClass,
                        TypeRelation.SAMETYPE_OR_SUBTYPE_LITERAL)));
        IQueryResult result = query.execute();
        if (result.getException() != null)
            throw new IllegalStateException(result.getException());
        return new ArrayList<Object>(result.getEObjects());
    }

}
```

Figure 16. Example of code for a query

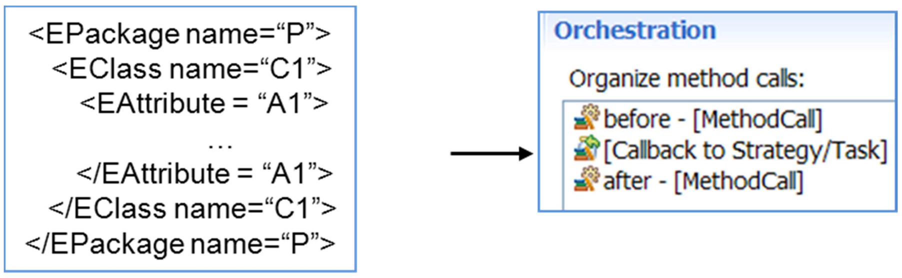
Pattern Callback



A Callback is a breaking point in the Pattern orchestration. When a Callback is encountered, the current Pattern orchestration is stopped; it continues when all the next Patterns to be executed by the Pattern Strategy are executed, and this recursively.

Example 1. Combination of the Domain-Driven Pattern Strategy and a Callback

The model-driven strategy in-depth navigates over the model. There is a pattern for each kind of model element with a containment relationship (e.g., Package, Class, Attribute, Operation). In the following picture, for each Pattern, methods before and after a callback enable to easily generate an XML-like file with open and close sections.

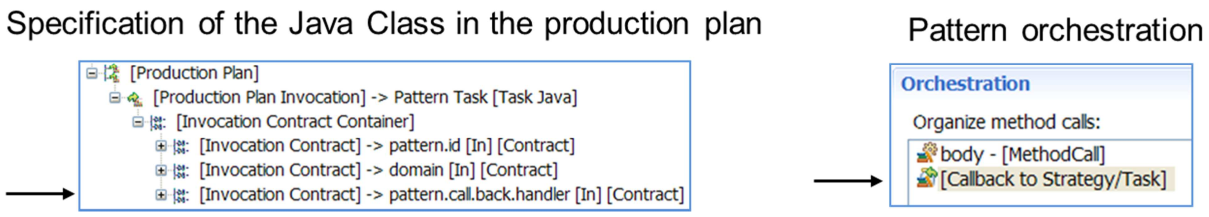


```
<EPackage name="P">
  <EClass name="C1">
    <EAttribute = "A1">
       …
    </EAttribute = "A1">
  </EClass name="C1">
</EPackage name="P">
```

Figure 17. An XML-like file generated by a Callback

Example 2. Specification of Callback handler in the Production Plan

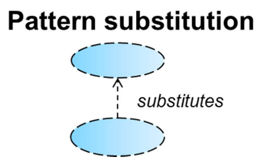
In a Production Plan, a Callback handler enables to specify a Java Class that will be called when a Callback is encountered in a Pattern orchestration.


Specification of the Java Class in the production plan
Pattern orchestration
Orchestration
Organize method calls:
body - [MethodCall]
[Callback to Strategy/Task]

```
[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]
```

Figure 18. Specification of Callback handler in a Production Plan

Pattern Substitution



A Pattern substitution replaces a Pattern by a list of Patterns. This list can be empty (for annihilating a Pattern), another Pattern, or a list of other Patterns (for replacing one Pattern by several). This mechanism enables to adapt a generation to a specific context. It is for instance used for definition of families of Pattern-based code generation.

Principle

Pattern substitution is a means to customize a Factory Component using a Pattern-based transformation. In the following picture, an initial Factory Component, at the top, contains a Pattern Library with two Patterns used in a Production Plan. Another Factory Component, at the bottom, contains a Pattern which substitutes the second Pattern by this Pattern.

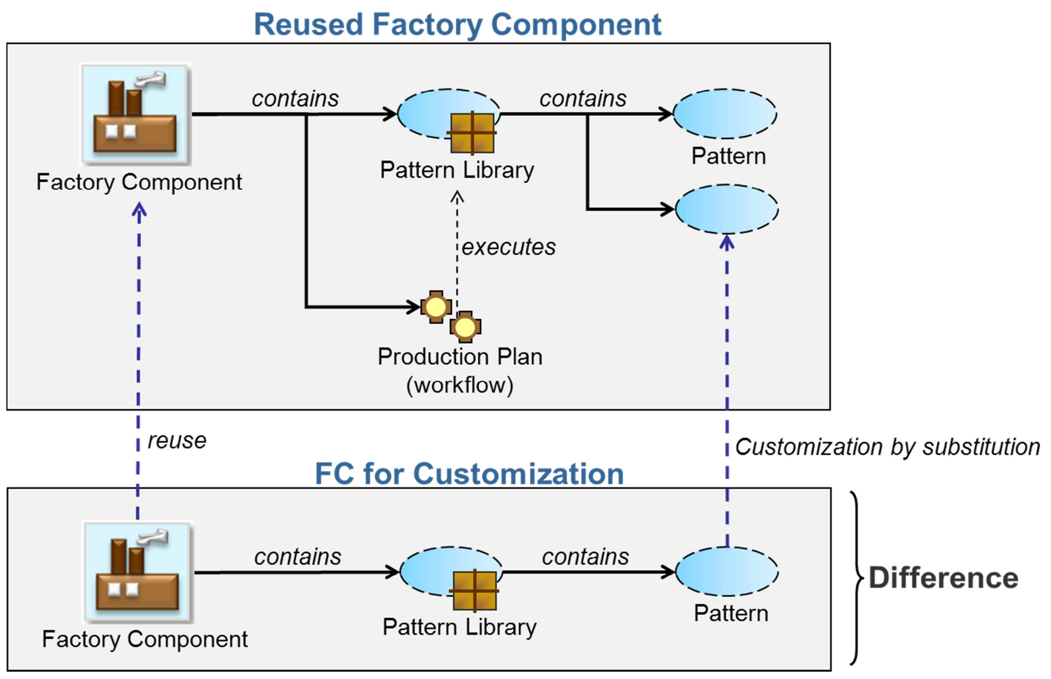


Figure 19. Customization of Factory Component by Pattern substitution

The following first example details this principle of customization. There exist two substitutions:

- The Pattern P1 of an initial Factory Component is replaced by the Patterns PA and PB.
- The Pattern P2 of an initial Factory Component is replaced by the Patterns PC and P2.

Remarks:

- The order of declaration is important. For instance, replacing P2 by PC and P2 is different from replacing by P2 and PC.
- For inhibiting P2, the substitution consisting in replacing P2 by an empty list.

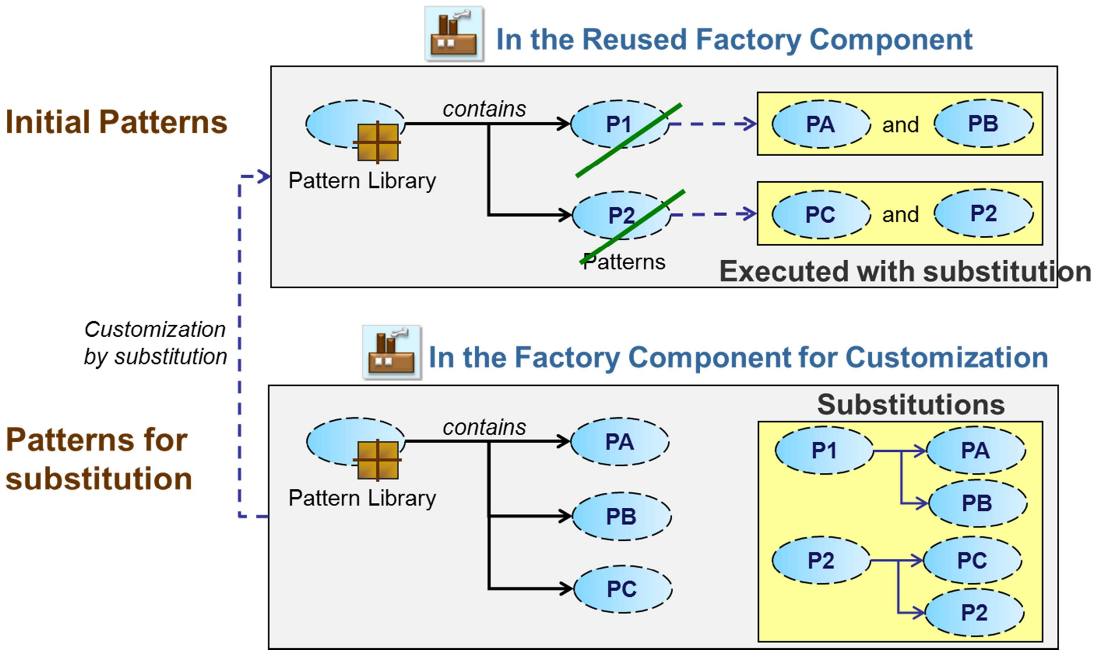
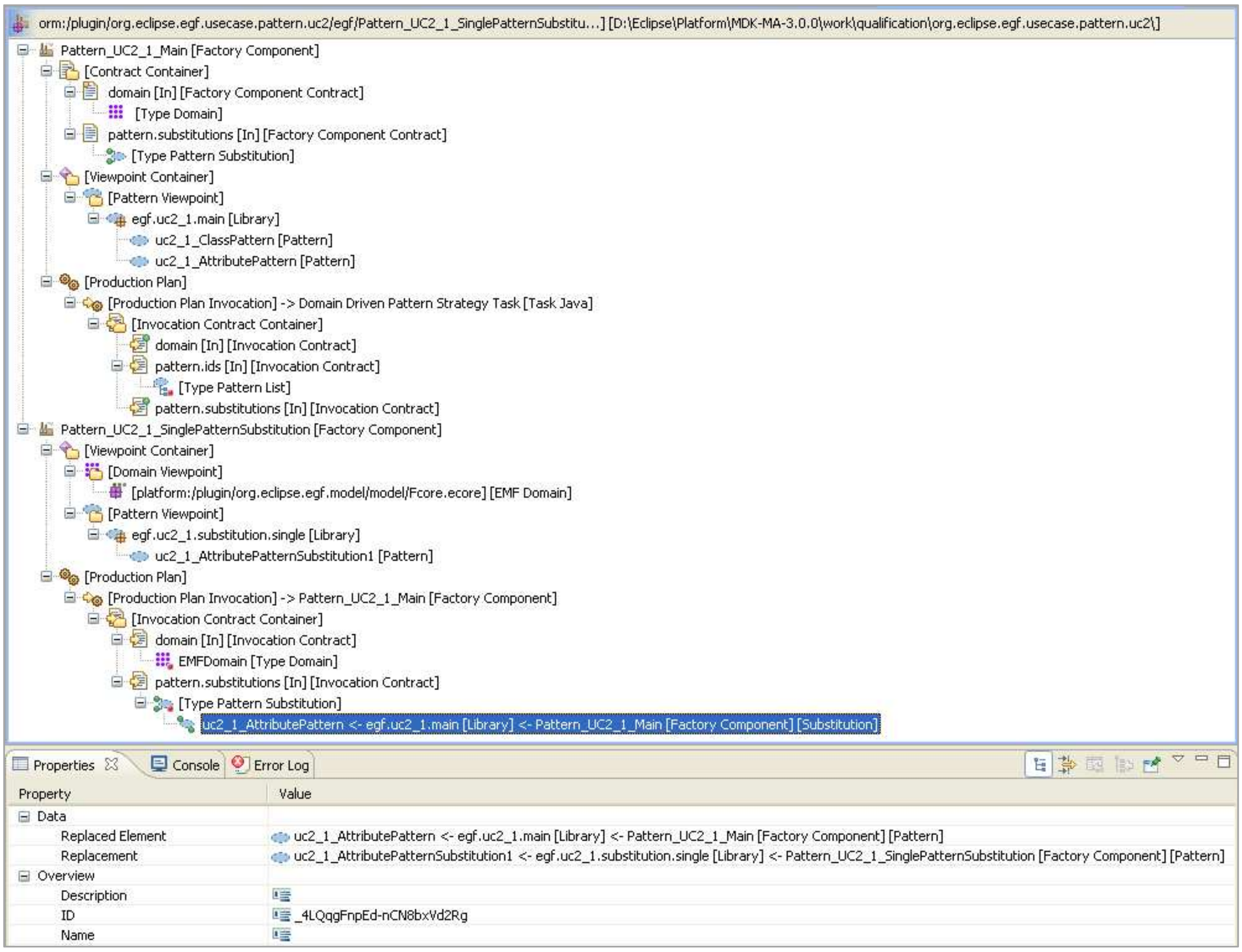

In the Reused Factory Component
Initial Patterns
Pattern Library
contains
P1
P2
Patterns
PA and PB
PC and P2
Executed with substitution
Customization by substitution
In the Factory Component for Customization
Patterns for substitution
Pattern Library
contains
PA
PB
PC
Substitutions
P1
PA
PB
P2
PC
P2

Figure 20. Example 1 of customization by Pattern substitution

In the second example:

- "Pattern_UC2_1_Main" is the reusable Factory Component.
  - It has two parameters: 1) a Domain model, 2) a Pattern substitution.
  - It contains a Pattern Library with two Patterns (i.e., uc2_1_ClassPattern and uc2_1_AttributePattern).
  - It uses a Domain-Driven Pattern Strategy which consumes the Domain model, the Pattern Library, and the Pattern substitution. If the substitution is empty, then Pattern Library is applied, else the Pattern Library is applied with the substitution.
- "Pattern_UC2_1_SinglePatternSubstitution" is the Factory Component which customizes "Pattern_UC2_1_Main":
  - It declares an EMF model for Domain model.
  - It contains one Pattern "uc2_1_AttributePatternSubstitution1".
  - The Production Plan calls "Pattern_UC2_1_Main" with the EMF model and the substitution of "uc2_1_AttributePattern" by "uc2_1_AttributePatternSubstitution1".

When the Pattern "Pattern_UC2_1_SinglePatternSubstitution" is executed, the Pattern "Pattern_UC2_1_Main" is executed by with the Pattern "uc2_1_AttributePatternSubstitution1" instead of the Pattern "uc2_1_AttributePattern".



```
orm:/plugin/org.eclipse.egf.usecase.pattern.uc2/egf/Pattern_UC2_1_SinglePatternSubstitu...] [D:\Eclipse\Platform\MDK-MA-3.0.0\work\qualification\org.eclipse.egf.usecase.pattern.uc2\]
Pattern_UC2_1_Main [Factory Component]
  [Contract Container]
    domain [In] [Factory Component Contract]
      [Type Domain]
    pattern.substitutions [In] [Factory Component Contract]
      [Type Pattern Substitution]
  [Viewpoint Container]
    [Pattern Viewpoint]
      egf.uc2_1.main [Library]
        uc2_1_ClassPattern [Pattern]
        uc2_1_AttributePattern [Pattern]
  [Production Plan]
    [Production Plan Invocation] -> Domain Driven Pattern Strategy Task [Task Java]
      [Invocation Contract Container]
        domain [In] [Invocation Contract]
        pattern.ids [In] [Invocation Contract]
          [Type Pattern List]
        pattern.substitutions [In] [Invocation Contract]
Pattern_UC2_1_SinglePatternSubstitution [Factory Component]
  [Viewpoint Container]
    [Domain Viewpoint]
      [platform:/plugin/org.eclipse.egf.model/model/Fcore.ecore] [EMF Domain]
    [Pattern Viewpoint]
      egf.uc2_1.substitution.single [Library]
        uc2_1_AttributePatternSubstitution1 [Pattern]
  [Production Plan]
    [Production Plan Invocation] -> Pattern_UC2_1_Main [Factory Component]
      [Invocation Contract Container]
        domain [In] [Invocation Contract]
          EMFDomain [Type Domain]
        pattern.substitutions [In] [Invocation Contract]
          [Type Pattern Substitution]
            uc2_1_AttributePattern <- egf.uc2_1.main [Library] <- Pattern_UC2_1_Main [Factory Component] [Substitution]
```

Property	Value
Data	
Replaced Element	uc2_1_AttributePattern <- egf.uc2_1.main [Library] <- Pattern_UC2_1_Main [Factory Component] [Pattern]
Replacement	uc2_1_AttributePatternSubstitution1 <- egf.uc2_1.substitution.single [Library] <- Pattern_UC2_1_SinglePatternSubstitution [Factory Component] [Pattern]
Overview	
Description	
ID	_4LQqgFnpEd-nCN8bxVd2Rg
Name	

Figure 21. Example 2 of customization by Pattern substitution

Customization at different levels

Pattern substitution is a powerful mechanism which enables to customize a pool of Patterns at different levels of definition. The methodological principle consists in defining a core of basic Patterns, sometimes very poor, enriched and fleshed out by Patterns which customize them by substitution. This principle is replicable multiple times. Customization creates a tree in under to split and isolate different branches of customization. This enables for instance to create Enterprise generation portfolio refined to fit different team or project concerns.

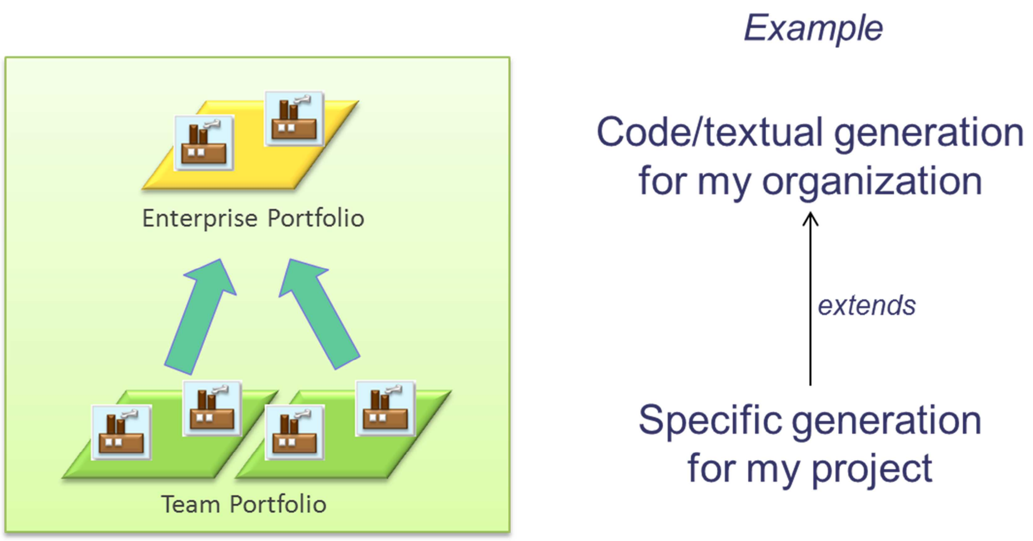
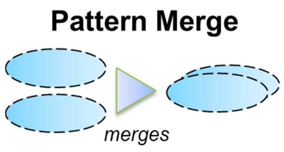

Enterprise Portfolio
Team Portfolio
Example
Code/textual generation for my organization
extends
Specific generation for my project

Figure 22. Successive levels of customization

Pattern Merge

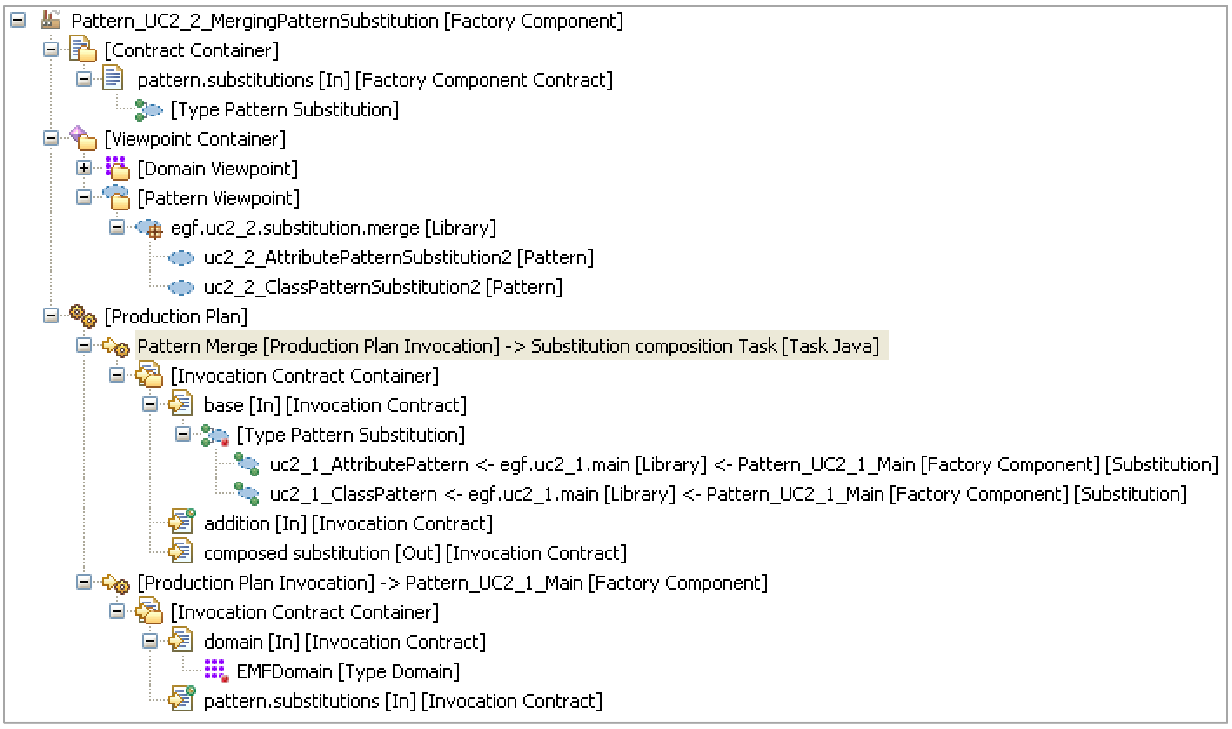


A Pattern merge is an operation implemented by a Task which merges two Pattern substitution lists.

The following example shows a Pattern merge Task in the Production Plan:

- The first parameter, "base", is a Pattern substitution which contains two substitutions:
  - "uc2_1_AttributePattern" substituted by "uc2_2_AttributePatternSubstitution2"
  - "uc2_1_ClassPattern" substituted by "uc2_2_ClassPatternSubstitution2")
- The second parameter, "addition", is the content of the Factory Component Contract "pattern.substitutions"
- The third parameter, "composed substitution", is the result of merge operation of "base" with "addition".

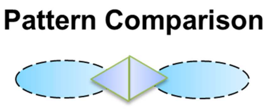
The merge result is next used in the Production Plan by the call to the "Pattern_UC2_1_Main" for the Pattern substitution. The principle of "Merge + Substitution" transmitted to Factory Component contract can be used by each "customizable" Factory Component.



```
Pattern_UC2_2_MergingPatternSubstitution [Factory Component]
  [Contract Container]
    pattern.substitutions [In] [Factory Component Contract]
      [Type Pattern Substitution]
  [Viewpoint Container]
    [Domain Viewpoint]
    [Pattern Viewpoint]
      egf.uc2_2.substitution.merge [Library]
        uc2_2_AttributePatternSubstitution2 [Pattern]
        uc2_2_ClassPatternSubstitution2 [Pattern]
  [Production Plan]
    Pattern Merge [Production Plan Invocation] -> Substitution composition Task [Task Java]
      [Invocation Contract Container]
        base [In] [Invocation Contract]
          [Type Pattern Substitution]
            uc2_1_AttributePattern <- egf.uc2_1.main [Library] <- Pattern_UC2_1_Main [Factory Component] [Substitution]
            uc2_1_ClassPattern <- egf.uc2_1.main [Library] <- Pattern_UC2_1_Main [Factory Component] [Substitution]
        addition [In] [Invocation Contract]
        composed substitution [Out] [Invocation Contract]
    [Production Plan Invocation] -> Pattern_UC2_1_Main [Factory Component]
      [Invocation Contract Container]
        domain [In] [Invocation Contract]
          EMFDomain [Type Domain]
        pattern.substitutions [In] [Invocation Contract]
```

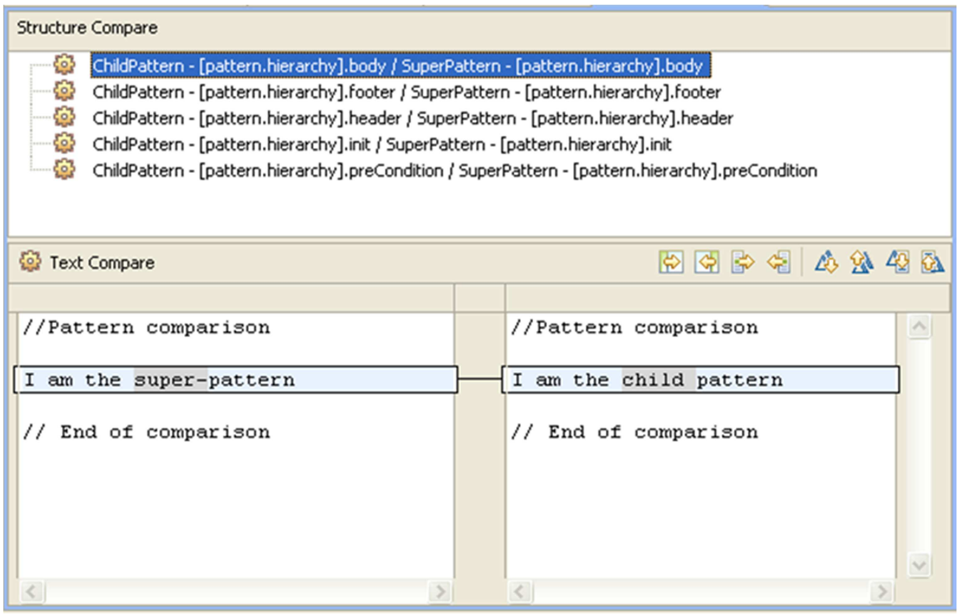
Figure 23. Example of Pattern merge

Pattern Comparison



For Pattern edition, when the number of Patterns rises up, Pattern comparison compares of a Patterns with its Super-Patterns, or Patterns with its cousins.

After selection of a Pattern, a comparison, by right-click, enables to compare the selected with its Super- and Child-Patterns. Multi-selection enables comparison of cousin Patterns. Live edition of compared Patterns is allowed.



ChildPattern - [pattern.hierarchy].body / SuperPattern - [pattern.hierarchy].body
ChildPattern - [pattern.hierarchy].footer / SuperPattern - [pattern.hierarchy].footer
ChildPattern - [pattern.hierarchy].header / SuperPattern - [pattern.hierarchy].header
ChildPattern - [pattern.hierarchy].init / SuperPattern - [pattern.hierarchy].init
ChildPattern - [pattern.hierarchy].preCondition / SuperPattern - [pattern.hierarchy].preCondition

```
//Pattern comparison

I am the super-pattern

// End of comparison
```

```
//Pattern comparison

I am the child pattern

// End of comparison
```

Figure 24. Example of Pattern comparison

PROCESS

The section presents the process dimension from the designer and developer viewpoints.

Designer viewpoint

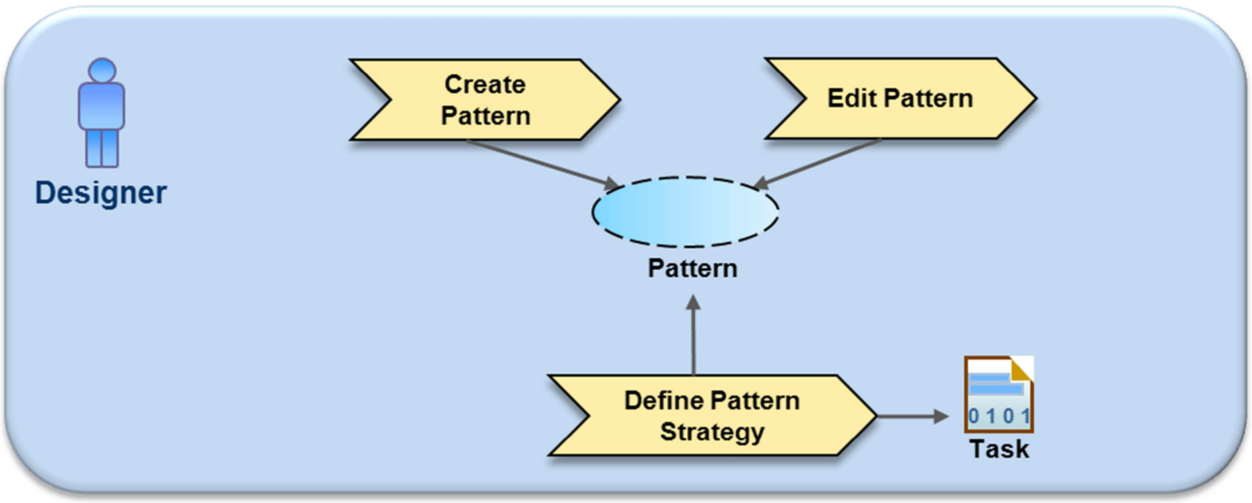


Figure 25. Process – Designer Viewpoint

Create Pattern	The Designer creates a Pattern. He considers all the architecture aspects, such as Pattern hierarchy, Pattern dependencies, Factory Component customization, Pattern substitution, generation framework, or product-line with Patterns.
Edit Pattern	The Designer edits the elements of the Pattern specification (e.g., Pattern parameters) and implementation e.g., methods, method orchestration).
Define Pattern Strategy	The Designer defines all the elements of a Pattern Strategy (e.g., algorithm, Pattern Strategy Task Contracts).

Table 1. Designer activities

The definition and implementation of a Pattern Strategy is limited to the advanced users of Patterns because it requires a good practice of Patterns and need of new strategy.

Developer viewpoint

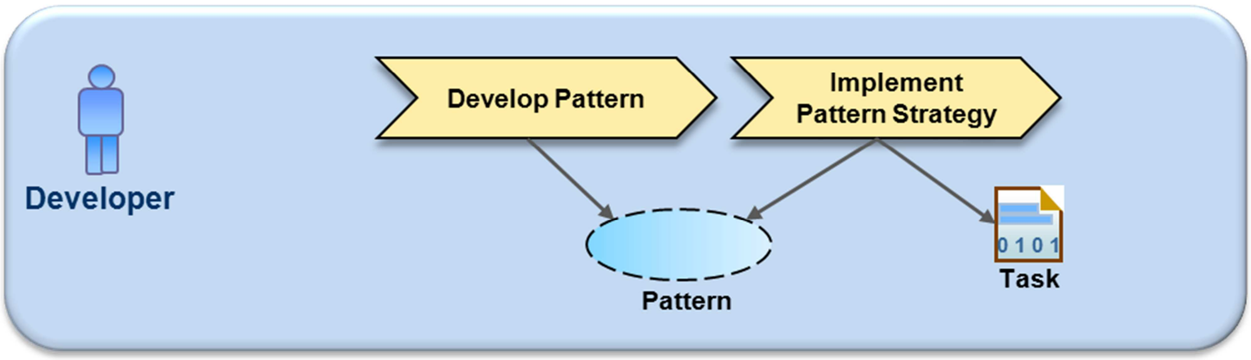


Figure 26. *Process – Developer Viewpoint*

Develop Viewpoint	The Developer implements the Pattern methods in a language with conforms to the Pattern nature.
Implement Pattern Strategy	The Developer implements the Pattern Strategy.

Table 2. Developer activities