TASK

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

Version: 1.0

DEFINITION

A task is a low level service with an implementation.

OBJECTIVES

The objectives of a task are to:

- Execute low level services from code written in a language (e.g., Java, Ant) or for a dedicated purpose (e.g., Acceleo task, text file format tasks).
- Unify the declaration and execution of low level services, whatever the used languages.

The interests are to:

- Define basic services reusable in different contexts.
- Seamlessly execute workflows of code written in different languages and tool contributions.

CONCERNS

Designer	• The designer functionally assembles basic services. • The designer executes tasks.
Developer	• The developer writes basic services in a language.

TASK

STRUCTURE

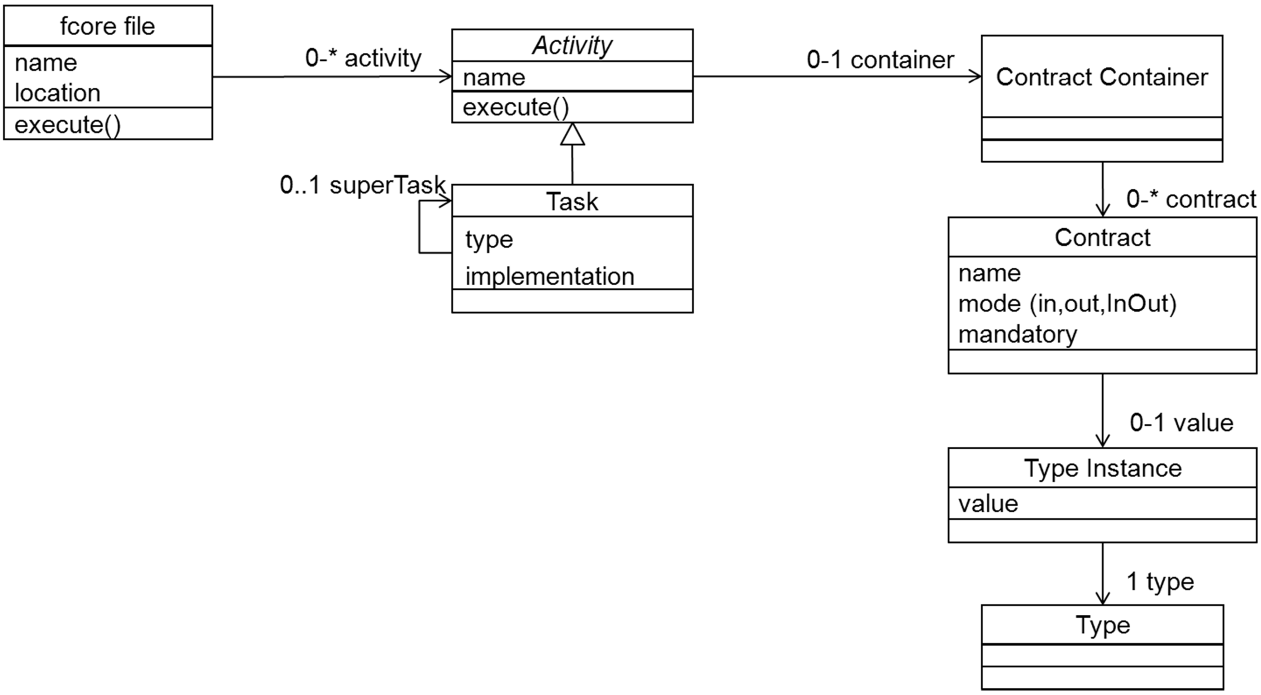


Figure 1. *Task at the metamodel level*

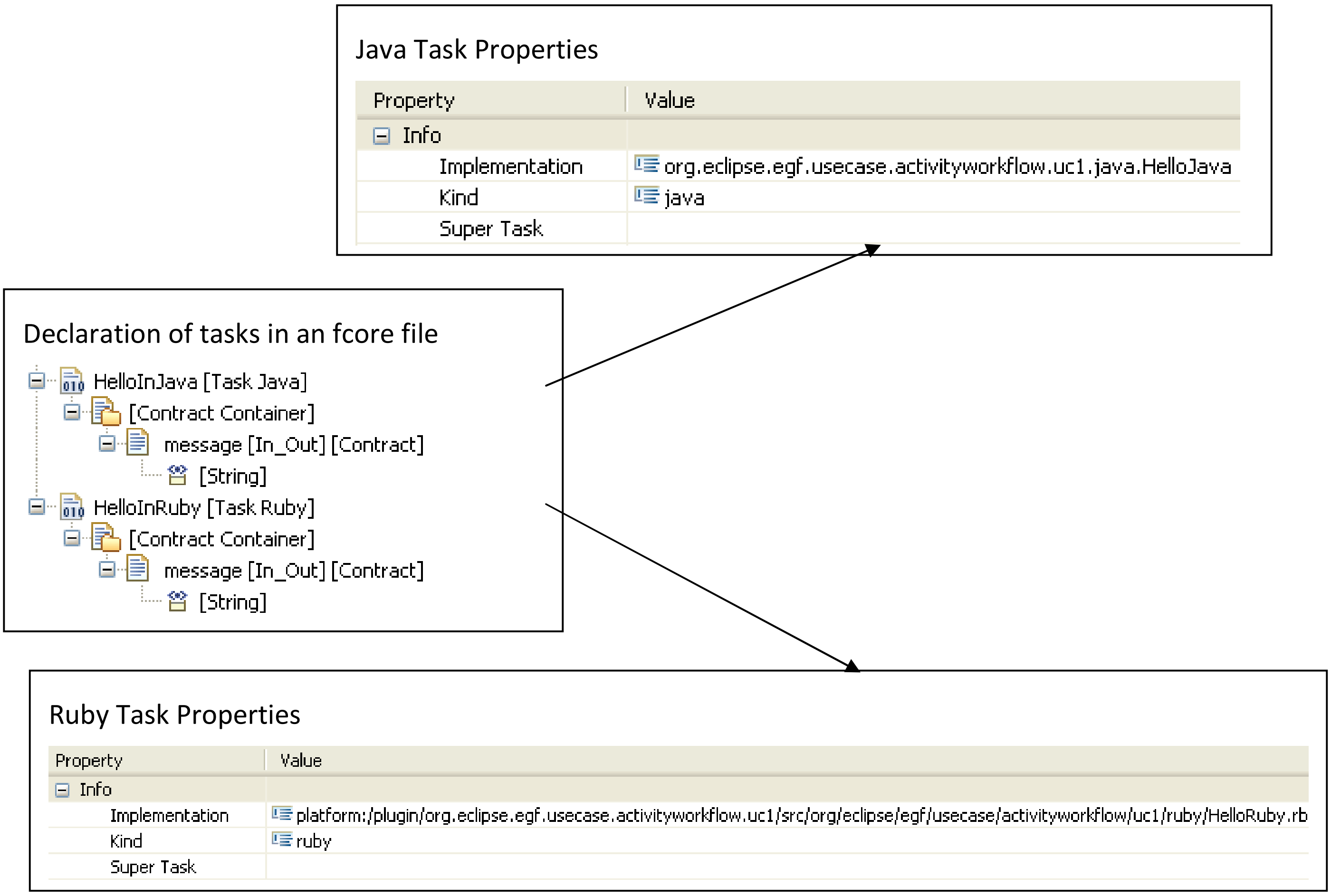
Comprehension:

- *Metamodel level*. A Task:
  - Is categorized by a Task type. A Task is for instance designed to a language (e.g., Java, Ant, Ruby Tasks) or tool (e.g., Acceleo Task).
  - Has an implementation. A Java Task is for instance implemented by a Java Class.
  - Can have an inheritance link with a super Task for Task specialization. In this case, there is parameter inheritance.
- *Task declaration at the instance level*. The Task metaclass defines the formalism to declare Tasks, such as Java or Acceleo Tasks. At the instance level, a Task declaration provides the Task type (e.g., Acceleo), a Class to execute the type of Task, and the execution engine (e.g., an Acceleo Task is executed by the Acceleo interpreter).
- *Tasks at the user level*. Third time, at the user level, all the Task declarations define the context to develop and execute user Tasks, for instance Java “HelloJava” or Ruby “HelloRuby” Tasks.
  - The Developer writes the Task implementation code, for instance the implementation of the Java “HelloJava” Class with all the parameters it requires.
  - The Designer declares a Task in an fcore file and the used implementation, for instance a Java Task uses the Java “HelloJava” Class for implementation.

A user completely ignores the metamodel level and Task declarations. The Developer concern is to implement Task code; the Designer concern is 1) to define the right off-the-shelf Task, 2) to select a Task implementation developed by the Developer, and 3) to define the Task interface with contracts (aka, parameters) in consistency with the parameters required by the implementation.

EXAMPLE

This part exemplifies the case of two Tasks: 1) HelloInJava Task is a Java Task implemented by the HelloJava.java Class, and 2) HelloInRuby Task is a Ruby Task implemented by the HelloRuby.rb script. The two Tasks have an input/output parameter named “message”.



Java Task Properties

Property	Value
⊟ Info	
Implementation	org.eclipse.egf.usecase.activityworkflow.uc1.java.HelloJava
Kind	java
Super Task	

Declaration of tasks in an fcore file

```
⊟ HelloInJava [Task Java]
    ⊟ [Contract Container]
        ⊟ message [In_Out] [Contract]
            [String]
⊟ HelloInRuby [Task Ruby]
    ⊟ [Contract Container]
        ⊟ message [In_Out] [Contract]
            [String]
```

Ruby Task Properties

Property	Value
⊟ Info	
Implementation	platform:/plugin/org.eclipse.egf.usecase.activityworkflow.uc1/src/org/eclipse/egf/usecase/activityworkflow/uc1/ruby/HelloRuby.rb
Kind	ruby
Super Task	

Figure 2. *Task declaration*

Implementation of the HelloJava.java Class

```
/**
 * This Java task
 *  1) Reads the "message" variable from the context
 *  2) Changes the "message" value
 *  3) Changes the "message" variable in the context with the new value
 *
 */

public class HelloJava implements ITaskProduction {

      String message = new String();

      public void preExecute(ITaskProductionContext productionContext,
                  IProgressMonitor monitor) throws InvocationException {

            // Get "message" value from context
            message  = productionContext.getInputValue
                        ("message", String.class);
      }

      public void doExecute(ITaskProductionContext productionContext,
                  IProgressMonitor monitor) throws InvocationException {

            // Change "message" value
          message = message + " from Java";
      }

      public void postExecute(ITaskProductionContext productionContext,
                  IProgressMonitor monitor) throws InvocationException {

            // Change "message" value in the context
            productionContext.setOutputValue("message", message);

      }

}
```

This example shows how the “message” declared in the Java Task is read (in the preExecute method) and updated (in the postExecute method). All the implementation steps can be written in the same method (e.g., doExecute).

Implementation of the HelloRuby.rb script

```
# To read input contract values and write output contract values,
# the class should extends the TaskProductionForRuby java class,
# and override the preExecute(), doExecute() and postExecute() methods.

# This Ruby task
#     1) Reads the "message" variable from the context
#     2) Changes the "message" value
#     3) Changes the "message" variable in the context with the new value

class HelloRuby < TaskProductionForRuby

  # ------- override the superclass preExecute() method --------

  def preExecute(context,monitor)

    # Get "message" value from context
    $_message = context.getInputValue("message", JavaLang::String.java_class);

  end

  # -------- override the superclass doExecute() method --------

  def doExecute(context,monitor)

    #-- Change "message" value from context
    $_message = $_message + " from Ruby"

    #-- Display on console
    puts $_message

  end

  # ------- override the superclass postExecute() method --------

  def postExecute(context,monitor)

    #-- Set "message" value from context
    context.setOutputValue("message", $_message);

  end

end

# instantiate the class
HelloRuby.new()
```

TASK

PROCESS

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

Designer viewpoint

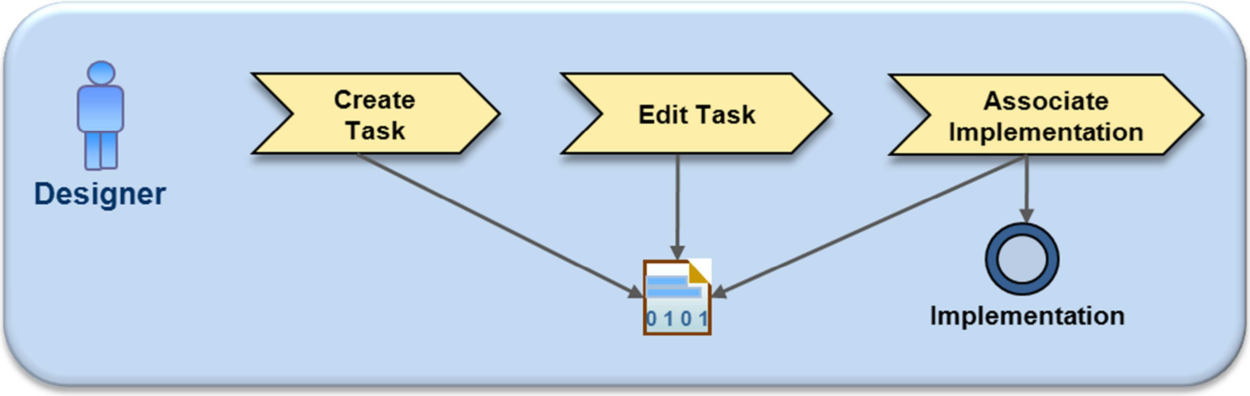


Figure 3. *Process – Designer Viewpoint*

Create Task	The Designer creates a Task with a type of Task in an fcore file.
Edit Task	The Designer sets the Task name, defines the contracts (name, type, input/output mode) as the Task parameters. He sets a super-Task if needed.
Associate Implementation	The Designer associates an implementation to the Task. He checks that the Task parameters match the implementation parameters.

Table 1. Designer activities

The Designer has to take care of two points:

- Task naming: like in Java, a namespace in the Task name may reflect a classification to organize Tasks.
- Task Contracts: a Task, as a low level service, requires parameters which define the Task interface.

Developer viewpoint

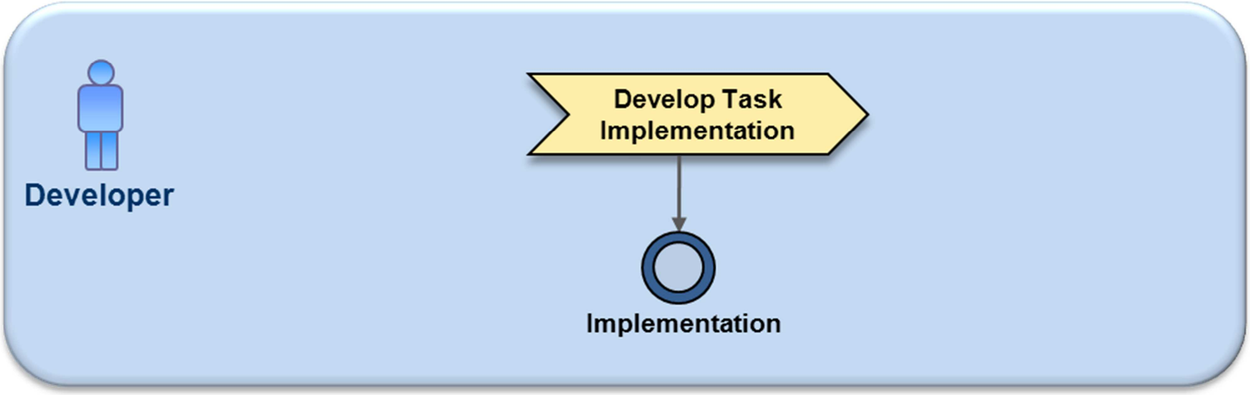


Figure 4. *Process – Developer Viewpoint*

Develop Task Implementation	The Developer realizes a Task implementation in a language or for a given tool. He uses the parameters declared in the Task with the context provided to the implementation.

Table 2. Developer activities

REFERENCES

Basic Tasks provided with EGF:

- Java Task
- Ant Task
- Jet Task
- Acceleo Task

Tasks available in EclipseLabs (http://code.google.com/a/eclipselabs.org/p/egf-portfolio/):

- Jython Task
- JRuby Task
- Acceleo2 Task