


Patterns reuse in the PASSI methodology



Massimo Cossentino¹, Luca Sabatucci¹, Silvio Sorace¹, and Antonio Chella^{1,2}

¹ICAR/CNR – Istituto di Calcolo e Reti ad Alte Prestazioni/ Consiglio Nazionale delle Ricerche

²Dipartimento di Ingegneria Informatica, University of Palermo



The Goal of This Work



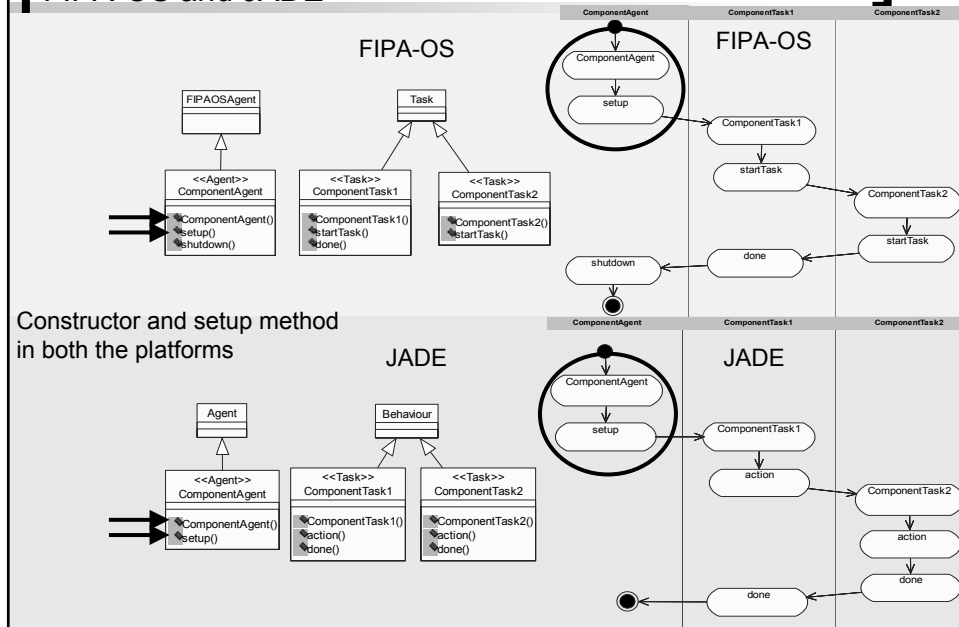
We aim to drastically affect the cost of developing a multi-agent application.

But... it is not easy to achieve this goal without simplifying the problem.

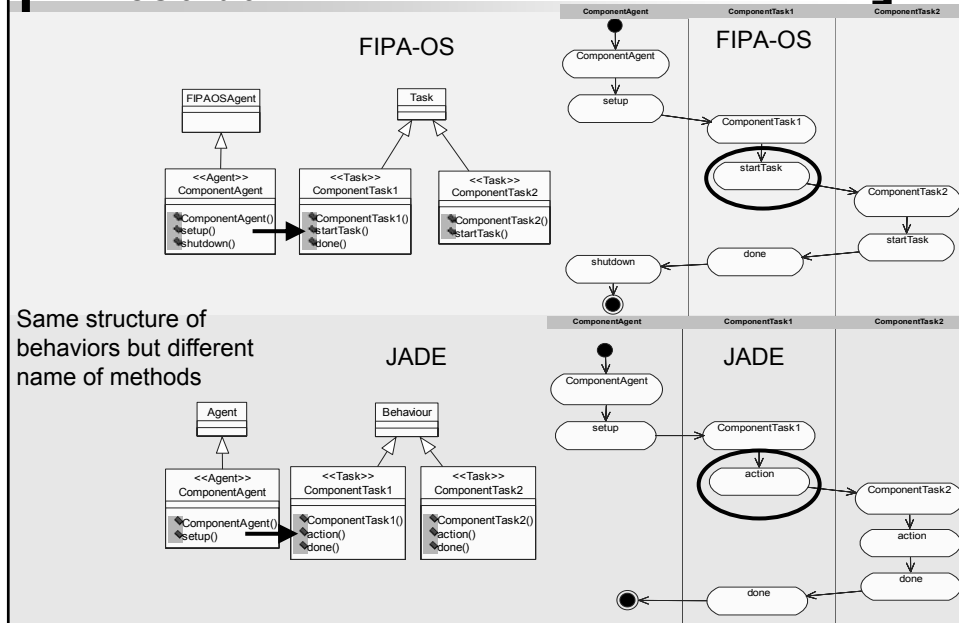
We decided of using FIPA-compliant platforms and this reduces effectively the dimension of the problem (almost all of these platforms are JAVA-based and have a similar structure).

By now we refer to two different platforms: FIPA-OS and JADE

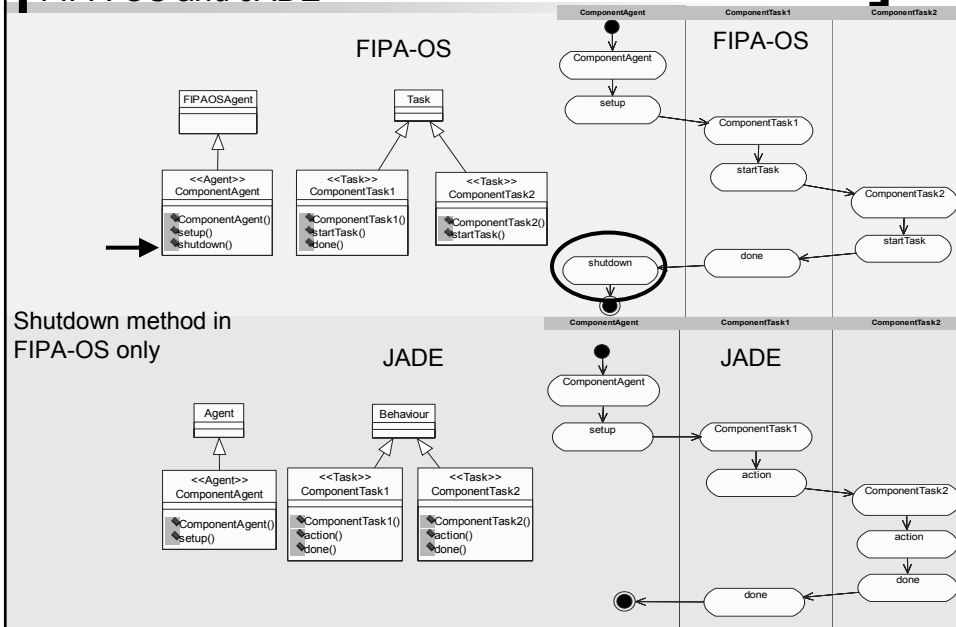
Difference in implementing a simple agent behavior in FIPA-OS and JADE



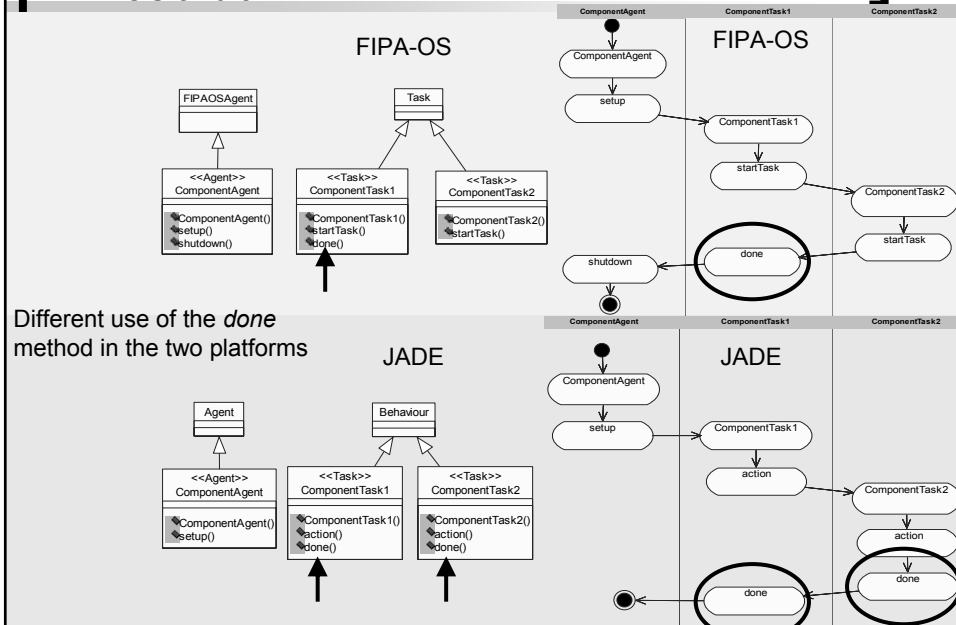
Difference in implementing a simple agent behavior in FIPA-OS and JADE



Difference in implementing a simple agent behavior in FIPA-OS and JADE



Difference in implementing a simple agent behavior in FIPA-OS and JADE



Pattern definition

We consider a pattern of agent as the solution to a common problem and in our approach it is composed of:

- A structure
 - Usually a base agent class and a set of task/behavior classes.
 - Described using UML class diagrams
- A behavior
 - Expressed by the agent using its structural elements
 - Detailed in UML dynamic diagrams (activity/state chart diagrams)
- A portion of code
 - Some lines of code implementing the structure and behavior described in the previous diagram

Classification of Patterns (structural)

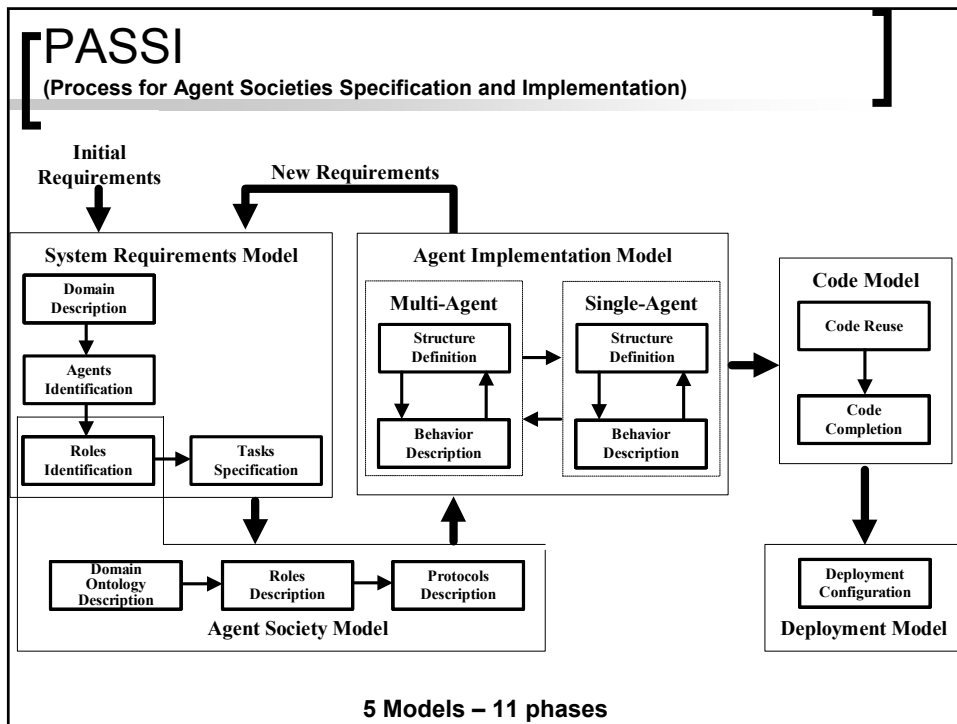
In order to simplify the organization of the pattern repository we introduce a structural and functional classification:

- Structural classification
 - **Action pattern.** A functionality of the system; it may be a method of either an agent class or a task class.
 - **Behavior pattern.** A specific behavior of an agent; we can look at it as a collection of actions (it represents a task in FIPA-OS and a behavior in JADE).
 - **Component pattern.** An agent pattern; it encompasses the entire structure of an agent together with its tasks.
 - **Service pattern.** A collaboration between two or more agents; it is an aggregation of components.

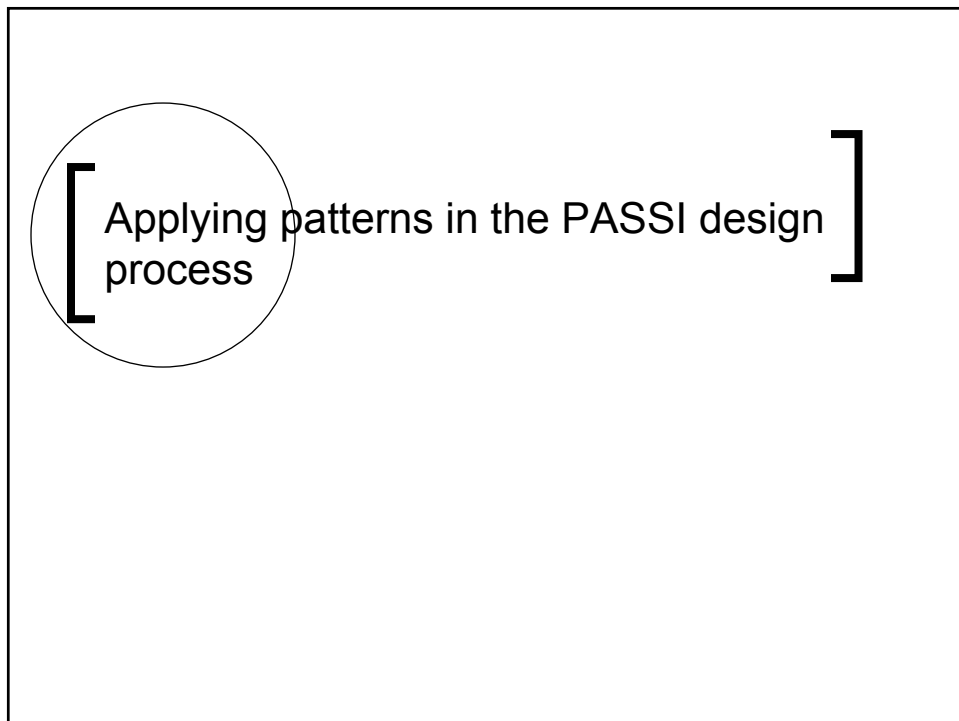
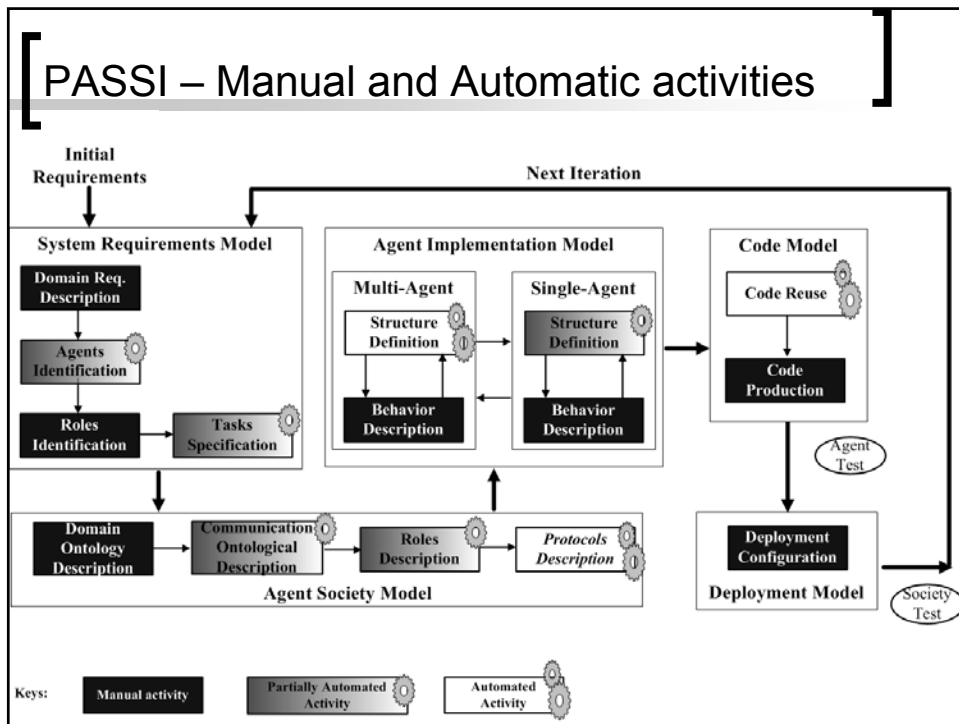
[Classification of Patterns (functional)]

- Looking at the functionality of the patterns, we can consider four categories:
 - **Mobility.** These patterns describe the possibility for an agent to move from a platform to another, maintaining its knowledge.
 - **Communication.** They represent the solution to the problem of making two agents communicate by a communication protocol.
 - **Elaboration.** They are used to deal with the agent's functionality devoted to perform some kind of elaboration on relevant amounts of data.
 - **Access to local resources.** They deal with information retrieval and manipulation of source data streams coming from hardware devices, such as cameras, sensors, etc.

[PASSI: Integrating the pattern reuse in the design methodology]



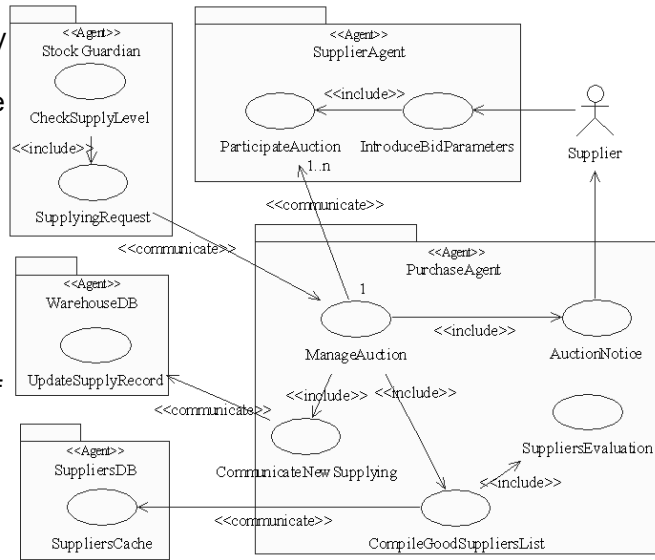
- PASSI**
- PASSI is conceived to be supported by PTK, an agent-oriented CASE tool
 - The functionalities of PTK include:
 - Automatic (total or partial) compilation of some diagrams
 - Automatic support to the execution of recurrent operations
 - Check of design consistency
 - Automatic compilation of reports and design documents
 - **Access to a database of patterns**
 - Generation of code and Reverse Engineering



A supply chain example

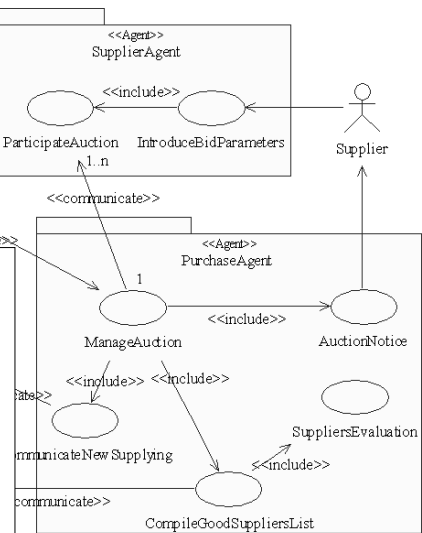
Let us consider a supply chain whose purpose is to ensure the availability of raw materials for the production chain of some manufacturing company.

On the right we can see a part of the Agent Identification diagram of this application



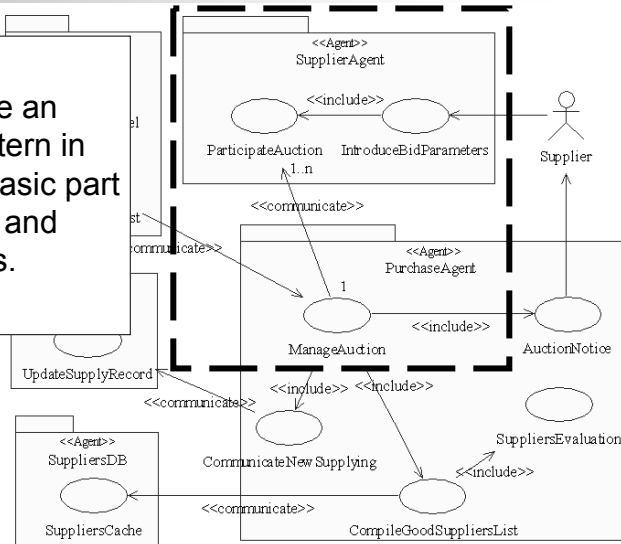
The supplying scenario

- The *PurchaseAgent* starts an auction to buy at the best price the raw materials requested by the *StockGuardian* agent.
- The possible suppliers are selected considering the previous experiences of the company with them
- Each selected supplier receives a notice and then, can post his own offer, interacting with an instance of the *SupplierAgent*.



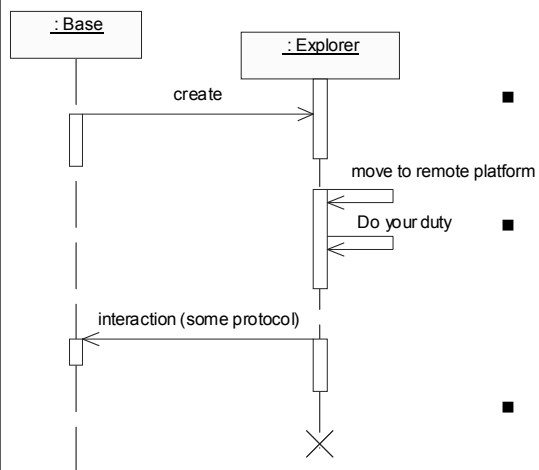
Where to apply the pattern

We can decide to use an *Explorer service** pattern in order to realize the basic part of the *SupplierAgent* and *PurchaseAgent* tasks.



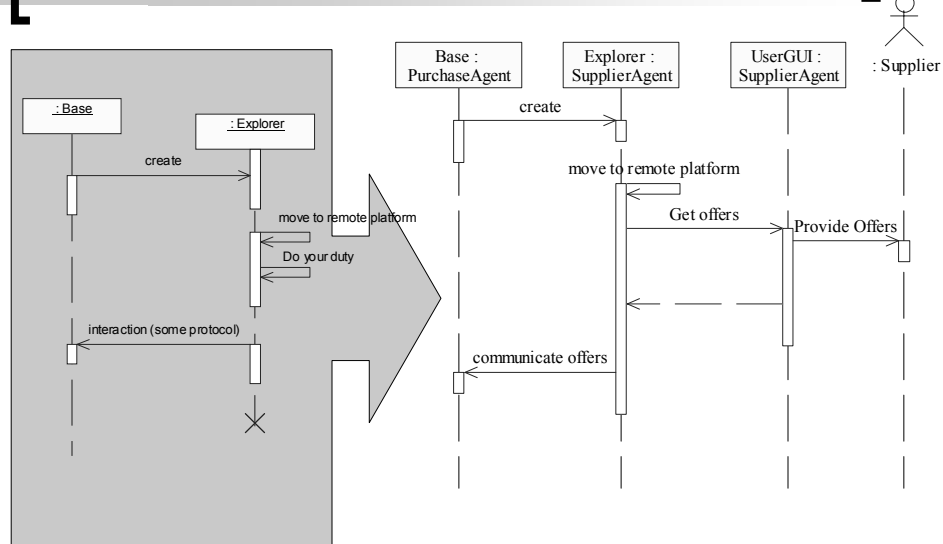
(*) A service pattern is composed of two interacting agents

The Explorer Pattern



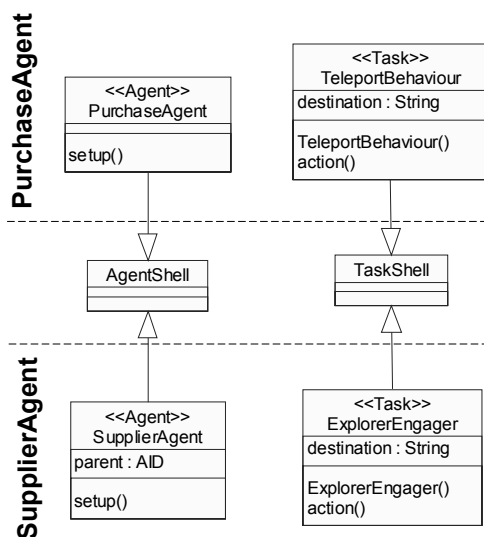
- The Explorer pattern allows the exploration of remote platforms with the intent to perform some kind of operation in them;
- it is composed of two agents: the **base agent** that will create an **explorer agent** and will send it to the other platform(s).
- The explorer agent will perform the required operation and then will inform its base agent.
- We can apply:
 - the *base agent* part of the pattern to the *PurchaseAgent*
 - the *explorer part* to the *SupplierAgent*

Pattern effect on the Roles Identification phase



Roles identified: **Base** (*PurchaseAgent*); **Explorer** and **UserGUI** (*SupplierAgent*)

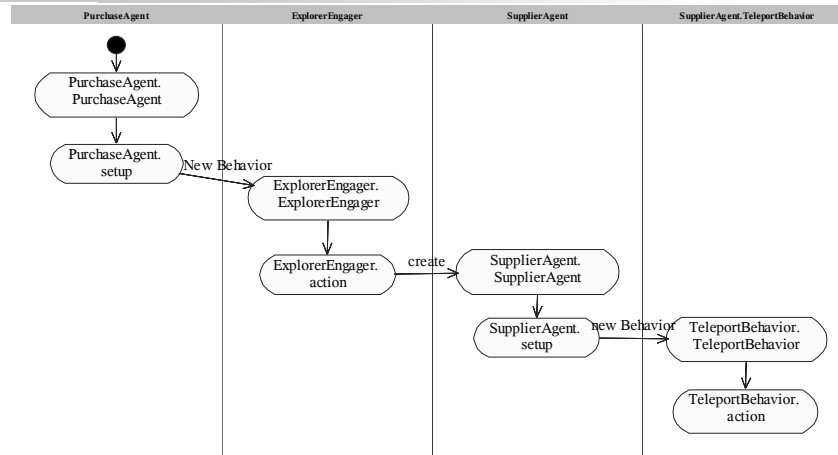
Pattern effect on the Multi-Agent Structure Diagram



These elements (agent and task classes) are the structure of the code that will be produced by PTK (PASSI ToolKit)



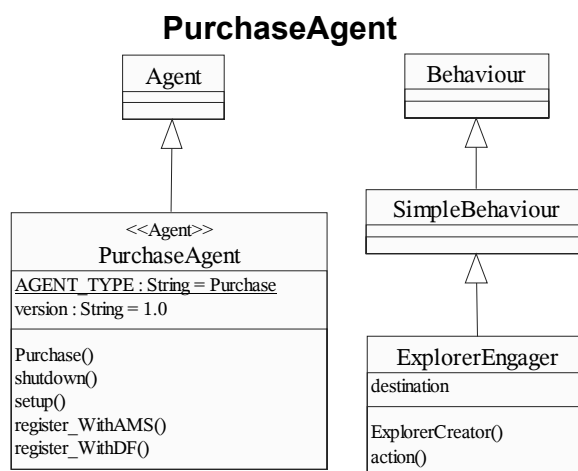
Pattern effect on the Multi-Agent Behavior Description Diagram



- Agents behavior is described in the MABD according to the pattern specification (*)

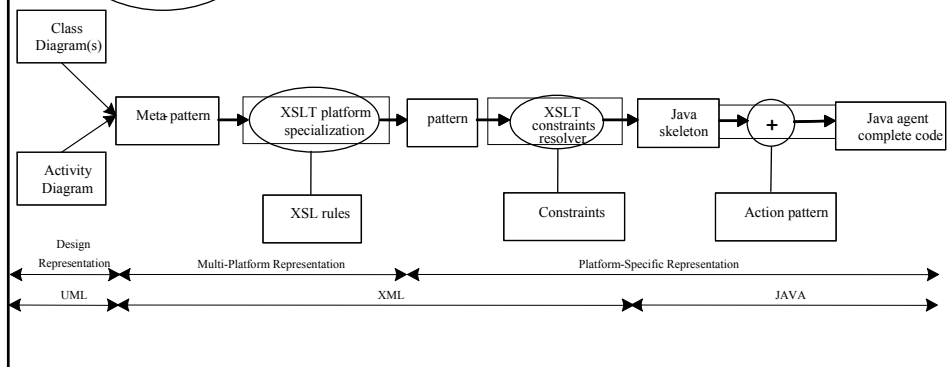
(*) With pattern engine AF 2.1 (alfa release only is available today)

Pattern effect on the Single-Agent Structure Diagram



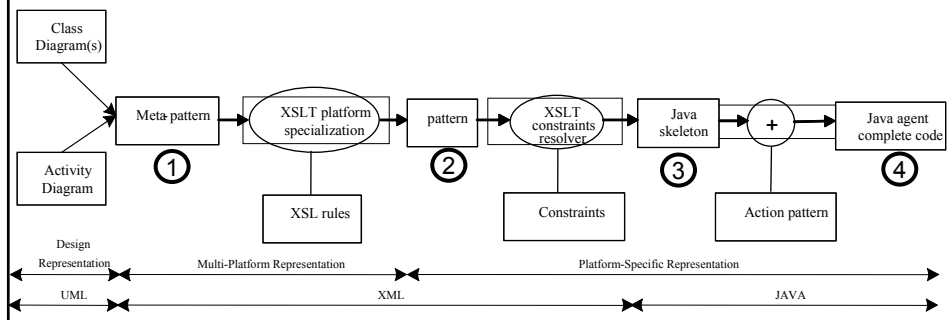
- The SASD diagram is automatically composed.
- Designer could add some implementation details when needed (e.g. parameters in operations)

The phases of the pattern production/reuse process



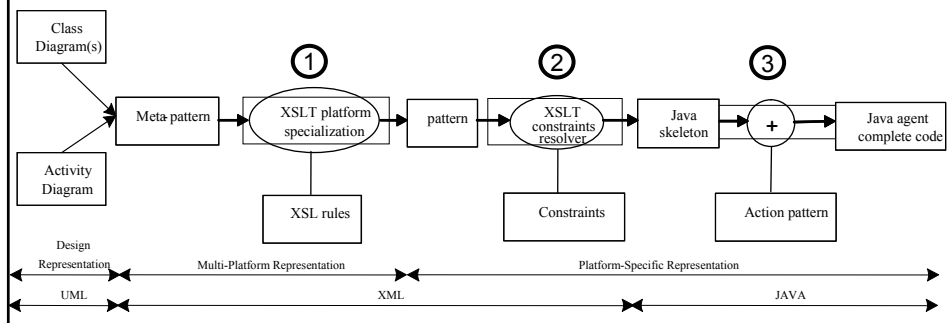
Code generation from models

- We use the Agent Factory code generation engine
- It adopts 4 different levels of abstraction in pattern representation:
 - Meta-pattern (platform independent) (1)
 - Pattern (platform specific, e.g. for FIPA-OS or JADE) (2)
 - JAVA skeletons (complete structure of the agent) (3)
 - JAVA complete code (structure+ inner parts of methods) (4)



[Code generation from models - 2]

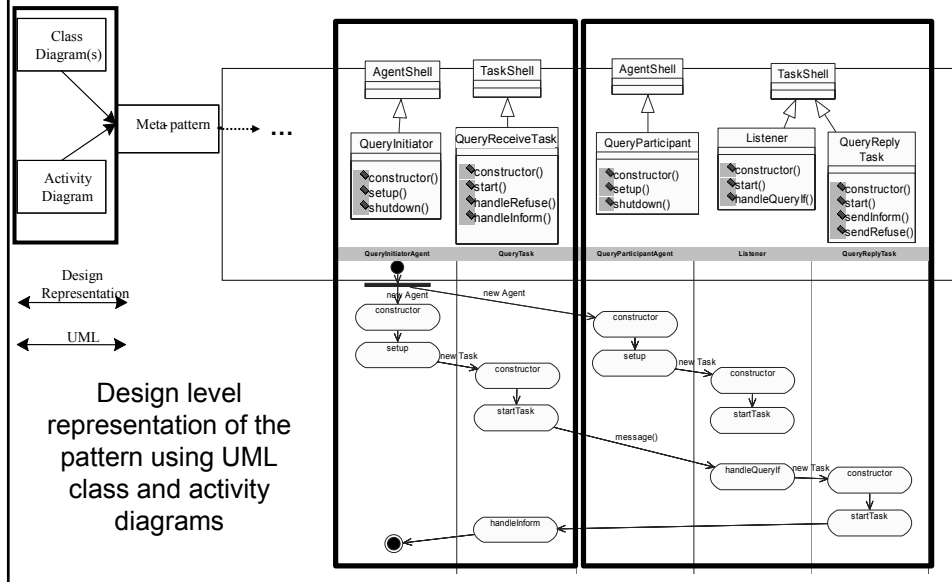
- Passage from one representation to another is obtained with 2 XML transformations (XSLT) **(1,2)** and the introduction of method codes from a DB **(3)**
- The overall architecture is similar to the OMG MDA (Model Driven Architecture)



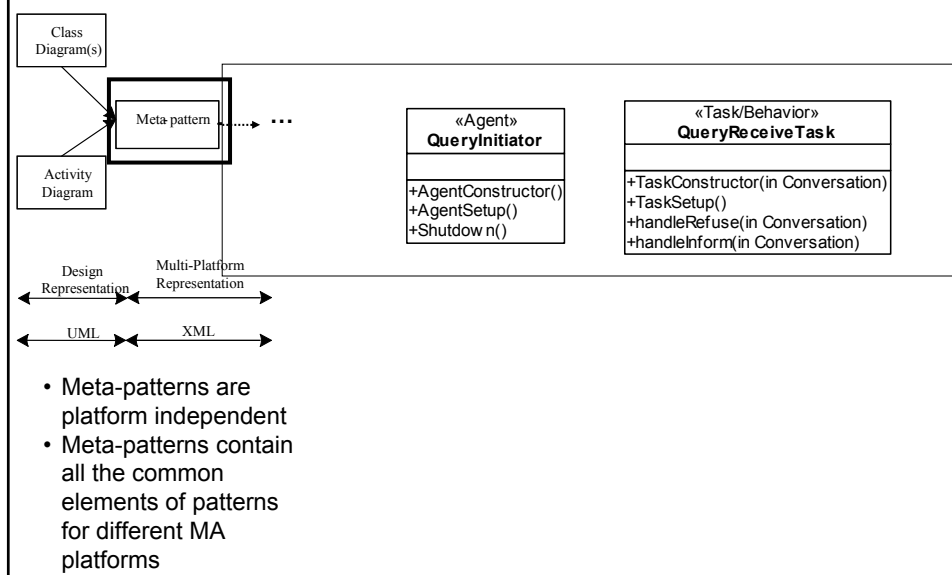
[Code generation phases]

From models to code

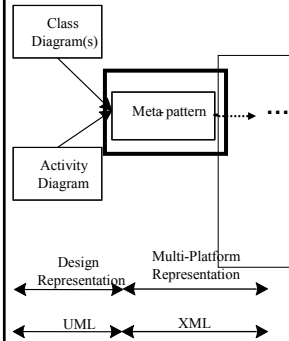
The design level representation of the pattern



The XML meta-pattern representation



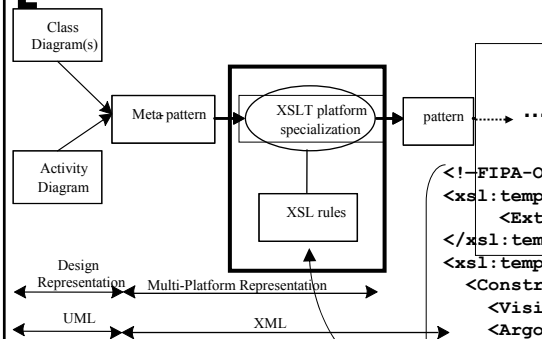
The XML meta-pattern representation



- Meta-patterns are platform independent
- Meta-patterns contain all the common elements of patterns for different MA platforms

```
<Agent name="QueryInitiator">
  <Visibility>public</Visibility>
  <ExtendsAgentShell/>
  <AgentConstructor>
    <Code>constructor@generic_agent</Code>
  </AgentConstructor>
  <AgentSetup>
    <Code>setup@generic_agent</Code>
  </AgentSetup>
  <Shutdown>
    <Code>shutdown@generic_agent</Code>
  </Shutdown>
  <Task name="QueryReceiveTask">
    <Visibility>public</Visibility>
    <ExtendsTaskShell/>
    <TaskConstructor>
      <Argument type="Conversation" name="conv"/>
      <Code>constructor@query_initiator_task</Code>
    </TaskConstructor>
    <TaskSetup>
      <Code>setup@query_initiator_task</Code>
    </TaskSetup>
    <Method name="handleRefuse" type="void">
      <Visibility>public</Visibility>
      <Argument type="Conversation" name="conv"/>
      <Code>handle_refuse@query_initiator_task</Code>
    </Method>
  </Task>
</Agent>
```

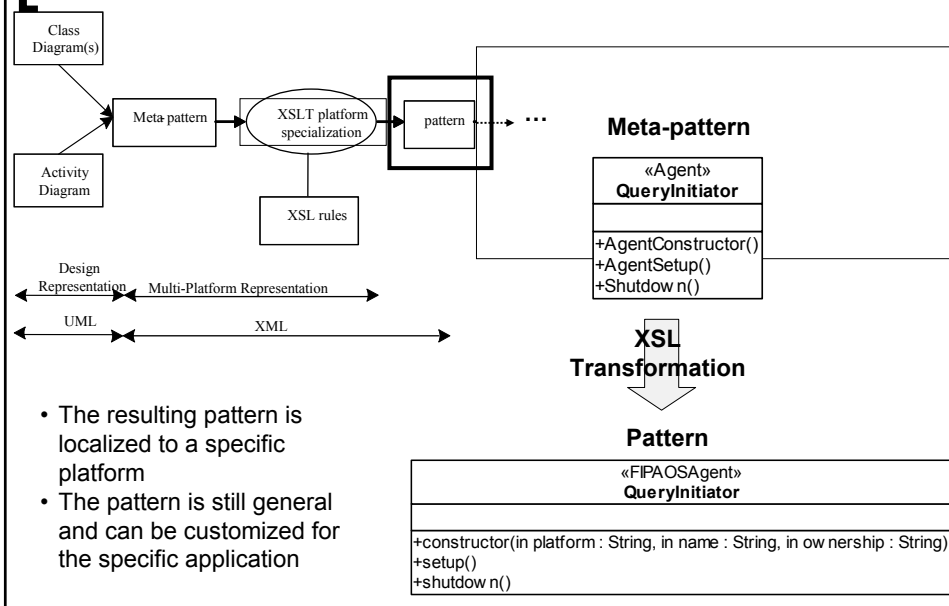
From meta-patterns to patterns – XSL rules



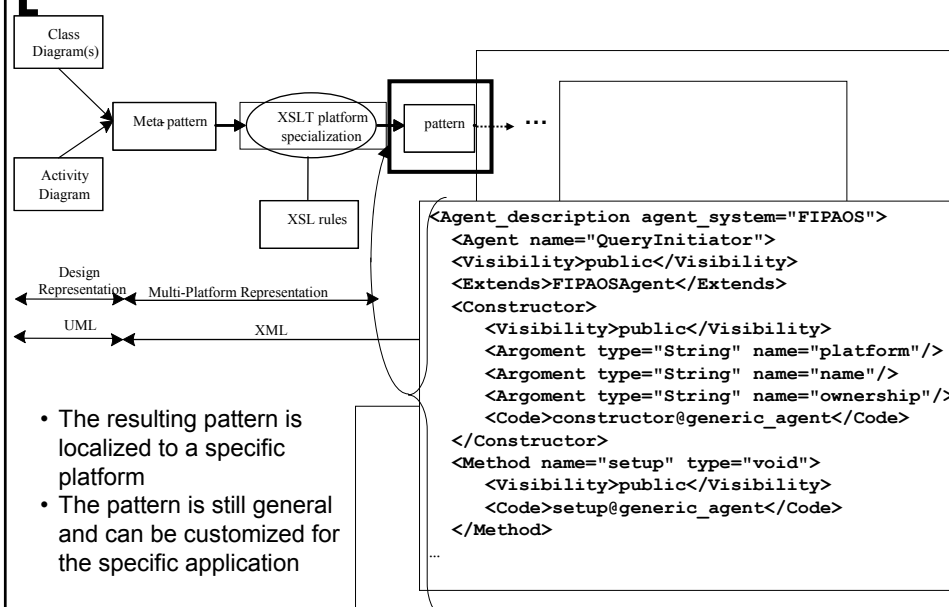
- Each pattern is specific for one of the selected platforms.
- In order to obtain a FIPA-OS pattern we apply to the meta-pattern the XSLT transformation shown on the right

```
<!--FIPA-OS SPECIFIC TRANSFORMATION-->
<xsl:template match="ExtendsAgentShell">
  <Extends>FIPAOSAgent</Extends>
</xsl:template>
<xsl:template match="AgentConstructor">
  <Constructor name="{parent::*/@name}">
    <Visibility>public</Visibility>
    <Argument type="String" name="platform">
    </Argument>
    <Argument type="String" name="name">
    </Argument>
    <Argument type="String" name="ownership">
    </Argument>
    <xsl:copy-of select="Code" />
  </Constructor>
</xsl:template>
<xsl:template match="AgentSetup">
  <Method name="setup" type="void">
    <Visibility>private</Visibility>
    <xsl:copy-of select="Code" />
  </Method>
</xsl:template>
```

From meta-patterns to patterns



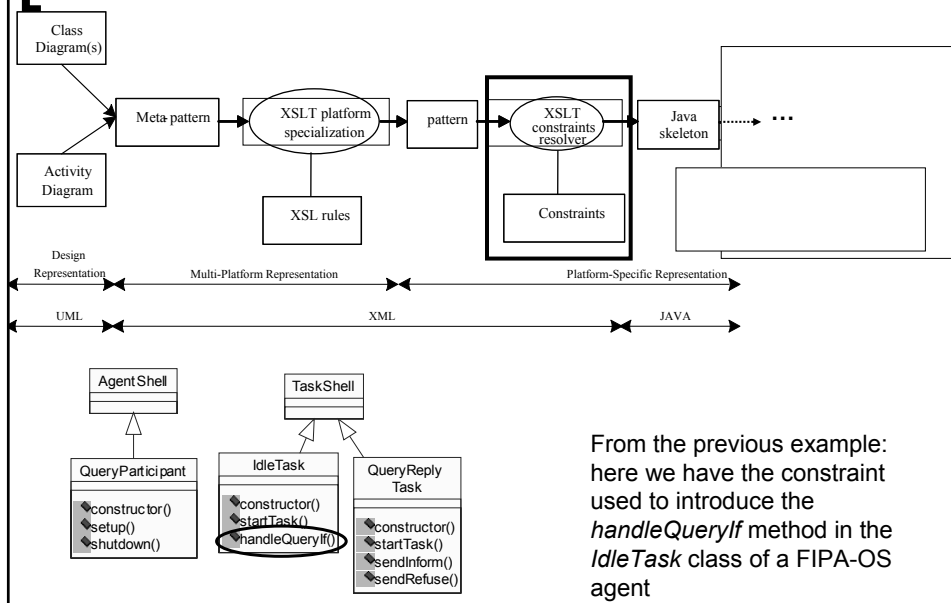
From meta-patterns to patterns



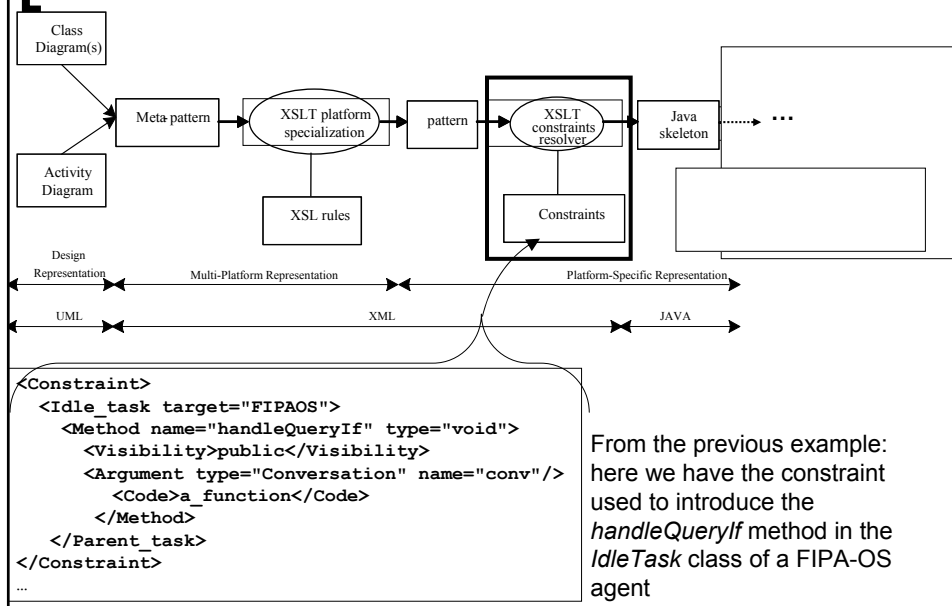
Patterns introduction in the project generates constraints

- When a pattern is applied to a project, it modifies the context in which it is placed (introducing new functionality into the system)
- The relationship between the pattern and existing elements could be expressed with a constraint.
- A **constraint** is a rule composed of two elements:
 - A **target** that specifies what agent/task will be influenced by the rule.
 - A **content** that expresses the changes to be applied when the pattern is inserted into the project (it could be an aggregation of attributes, constructors or methods).
- An example in FIPA-OS:
 - When we insert a communication task pattern into an existing agent, the listener task (*IdleTask*) should have a *handleX* method to catch performative acts of a particular type (i.e. QueryIf, Request, Inform, ...)

Constraints



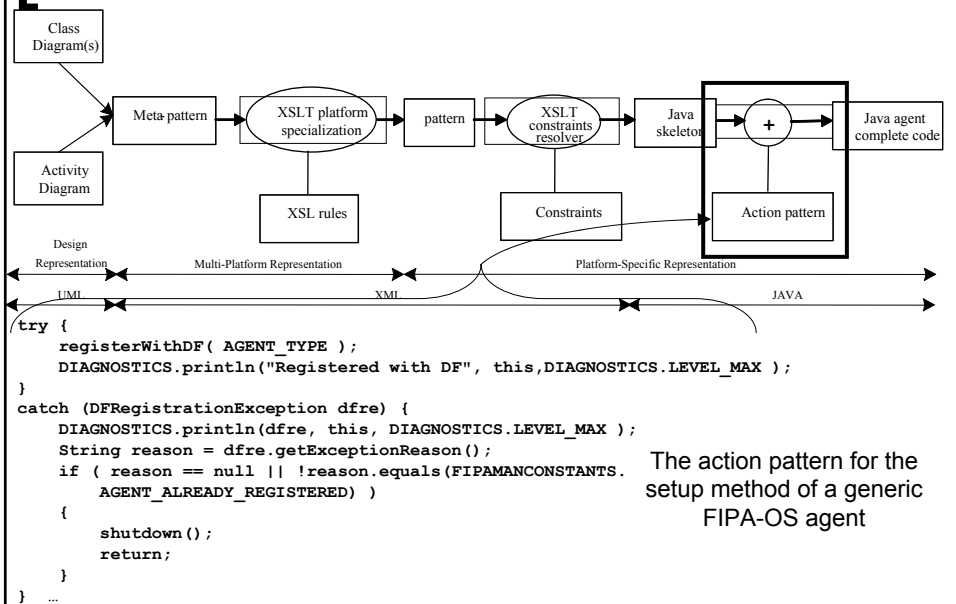
Constraints



From skeleton of classes to complete code

- With the previous steps we obtained a skeleton of the agent with its tasks/behaviors that is complete down to the method interfaces of each class
- **In order to (partially) fill the skeleton with the remaining code, action patterns are applied.**
- An **action pattern** is a portion of JAVA code realizing some kind of behavior. It is specific for each platform.
 - For example the registration to the DF service (Directory Facilitator, the yellow pages of the platform) it is part of the *setup@generic_agent* action pattern and it is introduced in the setup method of the agent
- Action patterns are stored in a database of pieces of code and the correct one for each method is selected referring to the value of the Code tag for the specific module (see the XML pattern representation)

Constraints



Experimental results

[Agents in a manufacturing company]

- The original application consists of more than 10 KLOCs
- It has been originally developed without pattern reuse
- Only a part of that application has been rebuilt in this experiment.
- The aim of our work was to evaluate the amount of code reused and estimate the amount of time saved in the design phase
- Types of Agents involved: 9
- Lines of code
 - > 2 thousands

[Experimental Results – lines of code]

Agents	Total LOC	Automatically Generated LOC	% of Total	Methods Body LOC	% of Autom. Generated
SuppliersDB	284	162	57	85	52
Wrapper	790	66	8	35	53
RawMaterialsDB	290	144	49	78	54
SuppliersAgent	124	43	34	26	60
PurchaseAgent	304	133	43	79	59
WarehouseDB	219	162	73	87	53
StockGuardian	109	98	89	49	50
RawMaterialsGUI	71	36	50	19	52
SuppliersGUI	69	36	52	19	52
Totale	2260	880	39	477	54

[Experimental Results – Design phase]

- We tried to evaluate the time difference in designing without and with pattern reuse
- No precise timetables were available for the original development phase therefore only an approximate estimation of the difference can be done
- We estimate that about 25-30% of the overall design time has been saved using patterns(*)

() We also considered a 'confidence' factor in evaluating it (the same project is designed quicker the second time you do it)*

[Experimental Results – Overall product quality]

- During the development phase with patterns we did some different choices (guided by available patterns) and the resulting architecture was considerably improved
- Software maintainability is increased because of the easier understandability of the agents structure (elementary bricks introduced by reused patterns are easy to identify)

Maintainability will be further improved by documenting patterns with PCL (pattern Comment Language) in the next release of Agent Factory

[Conclusions and future works]

[Conclusions and future works]

- Experimental results are encouraging but the amount of code automatically produced (and saving in design/coding time) depends on the dimension of the pattern repository
- It is difficult to synchronize the DB of patterns for many different implementation platforms
- In the Agent Factory project we created a web-based application that is available at: <http://mozart.csai.unipa.it/af/>