A Multi-Platform Architecture for Agent Patterns Representation and Reuse

Luca Sabatucci, Massimo Cossentino

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

Different use of the `done` method in FIPA-OS vs. JADE. Different name of methods.

Constructor and setup method in both the platforms.

Shutdown method in FIPA-OS only.
Pattern definition

We consider a pattern of agent as composed of its design level description and the corresponding JAVA code.

More in detail each pattern 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
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
(Process for Agent Societies Specification and Implementation)

5 Models – 11 phases
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
The phases of the pattern production/reuse process

- Meta-pattern
- XSLT platform specialization
- XSL rules
- Pattern
- XSLT constraints resolver
- Constraints
- Java skeleton
- Action pattern
- Java agent complete code

Class Diagram(s)
Activity Diagram

Design Representation
- UML

Multi-Platform Representation
- XML

Platform-Specific Representation
- JAVA
The design level representation of the pattern using UML class and activity diagrams.
The XML meta-pattern representation

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

```xml
<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>
```
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.
• The resulting pattern is localized to a specific platform
• The pattern is still general and can be customized for the specific application
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, …)
From the previous example: here we have the constraint used to introduce the `handleQueryIf` method in the `IdleTask` class of a FIPA-OS agent.
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).
The action pattern for the setup method of a generic FIPA-OS agent

```java
try {
    registerWithDF( AGENT_TYPE );
    DIAGNOSTICS.println("Registered with DF", this, DIAGNOSTICS.LEVEL_MAX );
} 

catch (DFRegistrationException dfre) {
    DIAGNOSTICS.println(dfre, this, DIAGNOSTICS.LEVEL_MAX );
    String reason = dfre.getExceptionReason();
    if ( reason == null || !reason.equals(FIPAMANCONSTANTS.AGENT_ALREADY_REGISTERED) )
    {
        shutdown();
        return;
    }
} ...
```
Experimental results
Mission of the robot:
- Surveillance of a building
  - reconnaissance of the building,
  - detection of new objects with the consequent update of the environmental knowledge and map description,
  - automatic detection of an intruder,
  - pursuit of the intruder

Types of Agents involved: 16

Lines of code
- > ten thousands
## Experimental Results

<table>
<thead>
<tr>
<th></th>
<th>WITHOUT patterns</th>
<th>WITH patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LOC</td>
<td>Months</td>
</tr>
<tr>
<td>Utility classes (drivers, algorithms)</td>
<td>4150</td>
<td>9</td>
</tr>
<tr>
<td>Study, Design, Implementation</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>Test and Tuning</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Agents/Ontology</td>
<td>4557/1903*</td>
<td>1461/1903*</td>
</tr>
</tbody>
</table>
Conclusions and future works
Conclusions and future works

- Experimental results are encouraging but the amount of code automatically produced depends on the dimension of the pattern repository.

- It is difficult to synchronize the DB of patterns for many different platforms.

- In the Agent Factory project (funded by Agentcities) we created a web-based application that is available at: [http://mozart.csai.unipa.it/af/](http://mozart.csai.unipa.it/af/)