Multiagent Systems
Agent Oriented Software Engineering

Viviana Mascardi
viviana.mascardi@unige.it

Dibris
Outline

1. Introduction
2. Agent-Oriented Architectures
3. AOSE methodologies

The features of agent-based systems are well suited to tackle the complexity of developing software in modern scenarios:

1. the autonomy of application components reflects the intrinsically decentralised nature of modern distributed systems and can be considered as the natural extension to the notions of modularity and encapsulation for systems that are owned by different stakeholders;
2 the flexible way in which agents operate and interact (both with each other and with the environment) is suited to the dynamic and unpredictable scenarios where software is expected to operate;

3 the concept of agency provides for a unified view of AI results and achievements, by making agents and MASs act as sound and manageable repositories of intelligent behaviours.
In the last few years, together with the increasing acceptance of agent-based computing as a novel software engineering paradigm, there has been a great deal of research related to the identification and definition of suitable models and techniques to support the development of complex software systems in terms of MASs.

This research, can be roughly grouped under the term Agent-Oriented Software Engineering (AOSE).
The Promise of AOSE for distributed systems engineering

- Today’s software engineering approaches are increasingly adopting abstractions approaching that of agent-based computing.

- This trend can be better understood by recognising that the vast majority of modern distributed systems scenarios are *intrinsically prone to be developed in terms of MASs*, and that modern distributed systems are already de facto MASs, i.e., they are indeed composed of *autonomous, situated, and social* components.
The Promise of AOSE for distributed systems engineering: Autonomy

- As far as autonomy is concerned, almost all of today’s software systems already integrate autonomous components.
- At its weakest, autonomy reduces to the ability of a component to react to and handle events, as in the case of graphical interfaces or simple embedded sensors.
- However, in many cases, autonomy implies that a component integrates an autonomous thread of execution, and can execute in a proactive way.
Today’s computing systems are also typically situated.

For example, control systems for physical domains and sensor networks, are built to explicitly manage data from the surrounding physical environment, and take into account the unpredictable dynamics of the environment.

Mobile and pervasive computing applications recognise (under the general term of *context-awareness*) the need for applications to model explicitly environmental characteristics and data.

Internet applications are also conceived for being aware of the features and the dynamics of the environment where they work, namely, the Web.
The Promise of AOSE for distributed systems engineering: Sociality

**Sociality** in modern distributed systems comes in different flavors:

1. the capability of components of supporting dynamic interactions, i.e., interaction established at run-time with previously unknown components;

2. the somewhat higher interaction level, overcoming the traditional client-server scheme;

3. the enforcement of some sorts of societal rules governing the interactions.
The Promise of AOSE for intelligent systems engineering

- Artificial Intelligence is mainly concerned with building intelligent systems: the very name of “Artificial Intelligence” literally suggests the notion of artifacts exhibiting intelligent behaviour.
- Therefore, AI can be considered as an engineering field (dealing with constructive concerns), rather than simply a scientific one (dealing with understanding and predicting intelligent systems behaviour).
The Promise of AOSE for intelligent systems engineering

- As already seen in the previous lessons, nearly 30 years of AI research were conducted by having research groups concentrate on single, isolated aspects of AI (like, say, artificial vision, knowledge representation, planning).

- There were few attempts of producing a reasonable set of conceptual and practical tools, which could promote the integration of such a vast amount of research findings into the mainstream practice of software development.

- This is where agents are actually becoming a key abstraction in today’s AI.
The very notion of agents provides a uniform conceptual space where all the findings of the AI field can be easily framed and related, and eventually find mainstream acceptance.
Mainstream research directions in AOSE:

1. Agent-Oriented Architectures
2. AOSE methodologies
3. Notation techniques
4. Agent-Oriented Programming Languages
5. Agent-based Modeling and Simulation
6. MAS infrastructures
Mainstream research directions in AOSE:

1. Agent-Oriented Architectures (in this lesson)
2. AOSE methodologies (in this lesson)
3. Notation techniques (after lessons on communication)
4. Agent-Oriented Programming Languages (lessons on Jason)
5. Agent-based Modeling and Simulation (lessons on NetLogo)
6. MAS infrastructures (lessons on JADE)
Outline

1. Introduction
2. Agent-Oriented Architectures
3. AOSE methodologies
For this part of the lesson we will use the slides by Antonio Moreno, in turn based on Rosenschein’s slides from Wooldridge book. They can be downloaded from https://www.slideshare.net/ToniMorenoURV/agent-architectures-3181662
But this wonderful picture is by myself!!!

The BDI architecture

- Plans
- Intentions
- Reasoner
- Goals
- Beliefs
- Environment
Outline

1. Introduction
2. Agent-Oriented Architectures
3. AOSE methodologies
Genealogy of some AOSE methodologies

AOSE methodologies

MAS, University of Genoa, DIBRIS

Agents and MASs

20 / 65

Reminder: what is a MAS

Agents and MASs

Organization

Inter-agent communication

Access to the environment

Environment
For this part of the lesson we will use the slides by Mehdi Dastani, “Multi-Agent Programming Software Development Methodologies for Multi-Agent Systems”. They can be downloaded from http://www.cs.uu.nl/docs/vakken/map/slides/gaia.pdf
Prometheus (further example, not part of the program)


- The Prometheus Design Tool:
  
The Prometheus methodology consists of three phases.

1. The *system specification phase* focuses on identifying the basic functionalities of the system, along with inputs (percepts), outputs (actions) and any important shared data sources.

2. The *architectural design phase* uses the outputs from the previous phase to determine which agents the system will contain and how they will interact.

3. The *detailed design phase* looks at the internals of each agent and how it will accomplish its tasks within the overall system.
Prometheus
This phase consists of the following steps, which are interleaved and iterated until the specification is considered sufficiently complete:

1. Identification of actors and their interactions with the system, in the form of percepts and actions;
2. Developing scenarios (by means of use cases) illustrating the system’s operation;
3. Identification of the system goals and sub-goals;
4. Identifying any external data;
5. Grouping goals and other items into the basic roles of the system.
This phase uses artefacts produced in the System Specification Phase to determine what agent types will be included in the system and the interaction between these agents. The steps in this phase are:

1. Determine the agent types
2. Develop the interaction protocols
3. Develop the system overview diagram
Prometheus: Detailed Design

This phase uses artefacts produced in the Architectural Design Phase to define the internals of every agent in the system and to specify how agents accomplish their overall tasks.

- Each agent is refined in terms of its capabilities, internal events, plans, and data structures.
- Each capability has a capability overview diagram that captures the structure of the plans within this capability and the events that are associated with these plans.
- The dynamic behaviour is described by process diagrams based on the interaction protocols identified in the previous phase.
Prometheus: a running example

We will go through the design and implementation of a small library management system. This system should be able to do the following things:

- Allow for checkout of books, providing a return date to the customer
- Allow for return of books
- Allow for reservation of unavailable books
- Allow for notification of overdue books
- Allow for notification of arrival of reserved books
System specification: Goals and subgoals

- **Checkout books**
  - Record book code to the user id checked out list.
  - Provide return date.

- **Return books**
  - Remove book code from the user id

- **Reserve unavailable books**
  - Record book code as reserved for user id
  - Show the current due date for the book
System specification: Goals and subgoals

- Give notification for overdue books
  - Access book record at the start of the day
  - Send email for overdue books

- Give notification of arrival of reserved books
  - Access the reserved list for user
  - Send email notification
System specification: Goals and subgoals

- **Checkout**
  - Record book code as checked out
  - Provide return date

- **Reserve**
  - Record book code as reserved
  - Show current due date
System specification: Goals and subgoals

- Return
- Remove book code from checkout list
- Find overdue books at the start of the day
- Notify overdue
- Send email for overdue books
System specification: Goals and subgoals

- Notify arrival
  - Find the reserved user
  - Send email mentioning arrival
Depending on the system we can identify 5 different scenarios:

1. When the user comes to checkout the book.
2. When the user returns the book.
4. When the user asks to reserve a book.
5. When the reserved book arrives.
System specification: Scenarios

- Checkout scenario
- Reserve scenario
- Return scenario
- Overdue scenario
- Arrival scenario
System specification: Checkout scenario

1. Request for checkout
2. Provide return date
3. Record book code as checked out
4. Provide book
System specification: Return scenario

1. Book returned
2. Remove book code from checkout list
System specification: Reserve scenario

1. Request for reservation
2. Record book code as reserved
3. Show current due date
4. Provide current due date
System specification: Arrival scenario

1. Reserved book arrives
2. Find the reserved user
3. Send arrival email
System specification: Overdue scenario

1. Start of the day
2. Find overdue books at the start of the day
3. Send overdue email
System specification: Role grouping

1. Checkout books
2. Return books
3. Overdue books
4. Reserve books
5. Send Arrival Notification
System specification: Checkout books role
System specification: Return and overdue books roles

1. **Book returned**
   - Return books
     - Return
     - Remove book code from checkout list

2. **Start of day**
   - Overdue books
     - Notify overdue
     - Find overdue books at the start of the day
     - Send email for overdue books
     - Send overdue email
System specification:
Reserve books and send arrival notification roles
Architectural design: Data coupling

In this phase we need to identify what type of data need to be stored in the system as beliefs. In our running example, we need to keep the information of all books that have been checked out and also the books that have been reserved.

- The checkout books role and return books role update the checkout belief and the overdue books role uses this to send overdue email.
- The reserve books role updates the reserve belief and the arrival notification uses it.
Architectural design:
Data coupling for checkout role
Architectural design:
Data coupling for reserve role

- Reserve database
- Reserve books
- Arrival notification
Now we group the roles and identify three agents to carry out this roles in the system. The agents identified are:

- Checkout Agent
- Reservation Agent
- Overdue Agent
Architectural design: Checkout agent-role coupling

- **CheckoutAgent**
  - Checkout books
  - Return books
Architectural design: Reservation agent-role coupling

![Diagram of OverdueAgent and Overdue books]
Architectural design: Overdue agent-role coupling

- ReservationAgent
  - Reserve books
  - Arrival notification
Architectural design: Agent acquaintance diagram

- CheckoutAgent
- ReservationAgent
- OverdueAgent
Architectural design: System overview diagram
Detailed design: Capabilities of the checkout agent

- Checkout Capability
- Return Capability
- Get Return Date Capability
Detailed design: Capabilities of the checkout agent
Detailed design: Checkout plan

Request for checkout → Checkout book → Provide book

Checkout book → Checkout database
Detailed design: Return plan
Detailed design: Get return date plan
Detailed design: Capabilities of the reservation agent

- Reservation Capability
- Arrival Notification Capability
Detailed design: Capabilities of the reservation agent

- Request for reservation
- Reserved book arrives

**ReservationCapability**
- Get return date
- Return date
- Provide current due date

**ArrivalNotificationCapability**
- Send arrival email

**Reserve database**
Detailed design: Reservation plan

Request for reservation

Get return date

Return date

Reserve book plan

Reserve database

Provide current due date
Detailed design: Arrival notification plan

- Reserved book arrives
- Send email plan
- Reserve database
- Send arrival email
Detailed design: Capabilities of the overdue agent

- Find overdue books Capability
Detailed design: Find overdue books plan

- Start of day
- Find overdue books
  - Get return date
  - Return date
- Send overdue email